Python 3

A Step-by-Step Guide to Learn, in an Easy Way, the Fundamentals of Python Programming Language

1ST EDITION

2020

By John Bach

Python3

A Step-by-Step Guide to Learn, in an Easy Way, the Fundamentals of Python Programming Language

1st Edition

2020

By

John Bach

For information contact :
(alabamamond@gmail.com, memlnc)
http://www.memlnc.com

First Edition: 2020

Python 3

"Programming isn't about what you know; it's about what you can figure out.” - Chris Pine

Install / Uninstall: Applications Supported by Canonical

When you first launch Add / Remove, a list of applications is displayed by category. Some are already installed, but most are not. The repository contains over 10,000 applications, so you can apply various filters to view smaller parts of the repository. The default filter , "Canonical-maintained applications" , shows a small subset of the total number of applications, only those officially supported by Canonical , which creates and maintains Ubuntu Linux.

Install / uninstall: all Open Source applications

Python 3 is not supported by Canonical, so first select "All Open Source applications" from the filter drop-down menu.

3.

Install / Uninstall: Search for "python 3"

After switching the filter to show all open applications, immediately use the search bar to find "python 3".

4.

Install / Uninstall: Select Python 3.0 Package

Now the list of applications has been reduced to those that match the query "python 3". There are two packages to note. The first is "Python (v3.0)". It contains the actual Python interpreter.

Install / Uninstall: Selecting the IDLE Package for Python 3.0

The second package you need is directly above the first one - "IDLE (using Python-3.0)". It is a graphical Python shell that you will use throughout this book.

After you check these two packages, click the “Apply Changes” button to continue.

6.

Install / uninstall: apply changes

The package manager will ask you to confirm that you want to install two packages - "IDLE (using Python-3.0)" and "Python (v3.0)".

Click the "Apply" button to continue.

7.

Install / uninstall: download progress bar

The package manager will show a progress bar while downloading the required packages from the Canonical online repository.

8.

Install / uninstall: installation progress bar

After downloading the packages, the package manager will automatically start installing them.

9.

Install / uninstall: new apps installed

If everything went well, the package manager will confirm that both packages were installed successfully. From here, you can start the Python shell by double-clicking on "IDLE" or by clicking "Close" to exit the package manager.

You can always start the Python shell from the Applications menu, Programming submenu, by choosing IDLE.

A graphical interactive Python shell for Linux

The Python shell is where you spend most of your time exploring

Python. All examples in this book assume that you know how to find the Python shell.

Go to using the Python shell .

Installation on other platforms

Python 3 is available on many different platforms. In particular, it is available on almost any Linux, BSD and Solaris distribution . For example, Re dHat Linux uses the yum package manager ; in FreeBSD its ports and packages collection ; Solaris has pkgadd with friends. A web search for "Python 3" + your operating system name will quickly show you if the appropriate Python 3 package is available and how to install it.

Using the Python shell

The Python Shell is where you can explore Python syntax, get online command help, and debug small programs. The Python graphical shell , IDLE , also includes a nice text editor that supports Python syntax highlighting. If you don't have your favorite text editor yet, IDLE is worth a try.

First things first, the Python shell itself is a wonderful interactive playground for playing with the language. Throughout the book, you'll see examples like this:

```
>>> 1 + 1 2
```

The first three angle brackets - >>> - denote a Python shell prompt. You do not need to enter it. This is just to show you that this example should run in the Python shell.

1 + 1 is what you enter. In the shell, you can enter any valid Python expression or command. Don't be shy, she won't bite! The worst thing that can happen is an error message. Commands are executed immediately (as soon as you press ↵ Enter), expressions are evaluated immediately too, and the shell prints the result.

2 - the result of evaluating this expression. As expected, 1 + 1 is a valid Python expression. The result is of course 2 .

Now let's try another example.

>>> print ('Hello world!') Hello world !

Pretty simple, right? But there are many more things you can do in the Python shell! If you get stuck somewhere, suddenly forget a command or what arguments you need to pass to a function, you can always call the online help in the Python shell. Just type help and press ↵ Enter .

Translating the shell message

>>> help

Type help () for interactive help, or help (object) for help about object.	Enter help () to enter online help mode or help (object) to get help about a specific object.

There are two modes of online help. You can get help on a specific object, it just prints the documentation and returns you to the Python shell prompt. You can also enter help mode, which does not evaluate Python expressions, but you simply enter keywords and command names, and everything that is known about this command is displayed in response.

To enter online help mode, type help () and press ↵ Enter .

Translating shell messages

>>> help ()

Welcome to Python 3.0! This is the online help utility.	Welcome to Python 3.0! You are in online help mode.
If this is your first time using Py thon, you should definitely check out the tutorial on the Internet at http://docs.python.org/tutorial/.	If this is your first time using Python, you should definitely check out the online tutorial at http://docs.python.org/tutorial/ [2] .
Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".	Enter a module name, keyword, or topic for help writing Python programs and using modules. To exit help mode and return to the interpreter, simply type quit .

To get a list of available modules, keywords, or topics, type "modules", "keywords", or "topics". Each module also comes wit h a one-line summary of what it does; to list the modules whose summaries contain a given word such as "spam", type "modules spam".

Type modules , keywords, or topics to see a list of available modules, keywords, and help topics . Each module has a short description of its purpose; for a list of modules that have a specific word in their descriptions, such as the word "spam", type modules spam .

help >

Note that the prompt has changed from >>> to help > . This means you are in online help mode. Here you can enter any keyword, command, module or function name - anything Python can understand - and read the documentation for it.

Translating shell messages

1
.

help > print

Help on built-in function print in module builtins:

print (...)

print (value, ..., sep = ', end = '\ n', file = sys.stdout)

Pri nts the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file: a file-like object

Reference for the built-in print function from the builtins module :

Print values to the specified stream or sys . stdout (default).
Optional named arguments:

(stream); defaults to the current sys.stdout.
sep: string inserted between values, default a space. end: string appended after the last value, default a newline.

- **file** - **file** -like object (stream), by default sys . stdout ;
- **sep** - the string inserted between the values, space by default;
- **end** - the string appended after the last value, by default a newline character .

help > PapayaWhip

no Python documentation found for 'PapayaWhip'

2.

no documentation for "PapayaWhip" found in Python [3]

help > quit

You a re now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the interpreter, you can type "help (object)". Executing "help ('string')" has the same effect as typing a particular string at the help> prompt.

3.

You leave help mode and return to the Python interpreter. If you want to get help about an object directly from the interpreter, you can type help (object) . Doing help ('string') works the same way as typing this string at the help > prompt .

>>>

4.

1. To get documentation on the print () function , simply type print and press ↵ Enter . Online help will show something like a man page: function name, short description, arguments, default values, and so on. If the documentation doesn't look very clear, don't be alarmed. In the

chapters ahead, you will get a better understanding of all of this.

2. Of course, the online help doesn't know everything. If you enter something that's not a Python command, module, function, or other built-in keyword, the online help just shrugs its virtual shoulders.
3. To exit online help, type quit and press ↵ Enter.
4. The prompt is again >>> to indicate that you have exited online help mode and returned to the Python shell.

IDLE, a graphical Python shell, also includes a text editor with Python code coloring.

Editors and IDEs for Python

IDLE is not the best option when it comes to writing Python programs. Since your programming is useful to begin with a study of development of the language itself, many developers prefer other text editors and IDEs (Integrated Development Environment, IDE). I won't go into detail on them here, but the Python community has a list of Python-enabled editors covering a wide range of platforms and licenses.

You can also take a look at the list of IDEs that support Python , although few support Python 3. One of them is PyDev , an Eclipse plugin [1] that turns it into a complete Python development environment. Both Eclipse and PyDev are cross-platform and open source .

On the commercial front, there is the Komodo IDE from ActiveState. It needs to be licensed for each user, but students are given discounts and the opportunity to try the product for free for a limited period.

I've been writing Python for nine years, doing it in GNU Emacs [2] , and debugging in the Python shell on the command line. There is no more correct or less correct way in Python development. Do what you think is right, what works for you.

Notes

1. It is written a little confused, but it is. On the topic, you can read w: en: PATH (variable) . - *Approx. per.*
2. Training course in English. Its translation into Russian is available in Wikibooks - Python 3.1 Tutorial . - *Approx. per.*
3. Papaya whip (English) - papaya mousse. - *Approx. per.*

Your first Python program

Don't run away from problems, judge yourself, or carry your burdens in righteous silence. Do you have a problem? Perfectly! It will do you good! Rejoice: dive into it and explore!

Venerable Henepola Gunarathan

Immersion

Typically books on programming start with a bunch of boring chapters on basic things, and gradually move on to creating something useful. Let's skip it all. Here's a complete, working Python program. Perhaps you will absolutely not understand anything about it. Don't worry about it, we'll be breaking it down line by line soon. But first read the code and see what you can learn from it.

[humansize.py]

SUFFIXES = { 1000 : ['KB' , 'MB' , 'GB' , 'TB' , 'PB' , 'EB' , 'ZB' , 'YB'] , 1024 : ['KiB' , 'MiB' , 'GiB' , 'TiB' , 'PiB' , 'EiB' , 'ZiB' , 'YiB']} def approximate_size (size , a_kilobyte_is_1024_bytes = True) : " 'Converts file size to human readable form. Key arguments: size - file size in bytes a_kilobyte_is_1024_bytes - if True (default), powers of 1024 are used if False, powers of 1000 are used Returns: text string (string) " ' if size < 0 : raise Value Error (' number must be non-negative ') multiple = 1024 if a_kilobyte_is_1024_bytes else 1000 for suffix in SUFFIXES [multiple] : size / = multiple if size < multiple: return ' {0: .1f} {1} ' . format (size , suffix) rais e ValueError ('the number is too large') if __name__ == '__main__' : print (approximate_size (1000000000000 , False)) print (approximate_size (1000000000000))

Now let's run this program from the command line . On Windows it will look something like this:

```
c: \ home \ diveintopython3 \ examples> c: \ python31 \ python.exe
humansize.py
1.0 TB
931.3 GiB
```

On Mac OS X and Linux, it will be much the same:

```
you @ localhost: ~ / diveintopython3 / examples $ python3 humansize.py
1.0 TB
931.3 GiB
```

What happened now? You have completed your first Python program. You invoked the Python interpreter on the command line and passed it the name of the script you wanted to execute. The script defines an approximate_size ()

function that takes an exact file size in bytes and calculates a "nice" (but approximate) size. (You may have seen this in Windows Explorer , in Mac OS X Finder , in Nautilus , Dolphin, or Thunar in Linux. If you display a folder with documents as a table, the file manager in each line will show an icon, document name, size, type, last modified date , etc. If there is a 1093 byte file named "TODO" in the folder, the file manager will not show "TODO 1093 bytes"; instead, it will say something like "TODO 1 KB". This is exactly what it does the approximate_size () function .)

Look at the last lines of the script, you will see two calls to print (approximate_size (arguments)) . These are function calls. First, approximate_size () is called , passed several arguments, then it takes its return value and passes it directly to print () . The print () function is built-in, you won't find its explicit declaration anywhere. It can only be used, anywhere, anytime. (There are many built-in functions, and many more functions that are separated into separate *modules* . Patience, fidget.)

So why is it always getting the same result when executing a script on the command line? We'll get to that. But first, let's take a look at the approximate_size () function .

Function declaration

Python has functions like most other languages, but no separate header files like C ++ or `interface` / `implementation` constructs like Pascal . When you need a function, just declare it like this:

```
def approximate_size ( size , a_kilobyte_is_1024_bytes = True ) :
```

Whenever you need a function, just declare it.

The declaration begins with the def keyword , followed by the name of the function, followed by the arguments in parentheses. If there are multiple arguments, they are separated by commas.

In addition, it is worth noting that the return type is not specified in the function declaration. Functions in Python do not define the type of values they return; they don't even indicate if the return value exists at all. (Actually, any function in Python returns a value; if a return statement is executed in a function , it returns the value specified in that statement; if not , it returns

None , a special null value.)

> In some programming languages, functions (returning a value) are declared with the `function` keyword , and subroutines (not returning values) are declared with the `sub` keyword . In Python, there are no subroutines. All functions return a value (even if it is None) and are always declared with the def keyword .

Function of the approximate_size () takes two arguments: `size bed` and `kilobyte_is_1024_bytes` , but none of them has a type. In Python, the type of variables is never explicitly set. Python calculates the type of a variable and keeps track of it itself.

> In Java and other statically typed languages, you must specify the type of the return value and each function argument. In Python, you don't need to explicitly specify the type for anything. Python itself keeps track of the types of variables based on the values assigned to them.

Optional and named arguments

In Python, function arguments can have default values; if the function is called without an argument, then it takes its default value. In addition, arguments can be specified in any order by specifying their names.

Let's take another look at the approximate_size () function declaration :

def approximate_size (size , a_kilobyte_is_1024_bytes = True) :

The second argument , `a_kilobyte_is_1024_bytes,` is written with the default value True . This means that this argument is optional; you can call the function without it, and Python will act as if it was called with True as the second parameter.

Now let's take a look at the last lines of the script:

if __name__ == '__main__' : print (approximate_size (1000000000000 , False)) ① print (approximate_size (1000000000000)) ②

① The approximate_size () function is called with two arguments. Inside the function the approximate_size () variable `a_kilobyte_is_1024_bytes` be

False , since the False transmitted explicitly in the second argument.

② The approximate_size () function is called with only one argument. But that's okay, because the second argument is optional! Since the second argument is not specified, it defaults to True , as defined in the function declaration.

You can also pass values to a function by name.

>>> from humansize import approximate_size
>>> approximate_size (4000, a_kilobyte_is_1024_bytes = False) ① '4.0 KB' >>> approximate_size (size = 4000, a_kilobyte_is_1024_bytes = False) ② '4.0 KB' >>> approximate_size (a_kilobyte_is_bytes = False , size = 4000) ③ '4.0 KB' >>> approximate_size (a_kilobyte_is_1024_bytes = False, 4000) ④ File "<stdin>", line 1 SyntaxError: non-keyword arg after keyword arg >>> approximate_size (size = 4000, False) ⑤ Fil e "<stdin>", line 1 SyntaxError: non-keyword arg after keyword arg

Translating shell messages:

File "<stdin>", line 1
SyntaxError: unnamed argument after named

① The approximate_size () function is called with 4000 as the first argument and False as an argument named `a_kilobyte_is_1024_bytes` . (He comes in second, but that doesn't matter, as you'll soon see.)

② The approximate_size () function is called with a value of 4000 for `size` and False for `a_kilobyte_is_1024_by tes` . (These named arguments appear in the same order as they appear in the function declaration, but

that doesn't matter either.)

③ The approximate_size () function is called with False for `a_kilobyte_is_1024_bytes` and 4000 for `size` . (See? I told you that order is not important.)

④ This call does not work because the named argument is followed by an unnamed (positional) one. If you read the list of arguments from left to right, then as soon as a named argument is encountered, all arguments following it must also be named.

⑤ This call doesn't work either, for the same reason as the previous one. Amazing? After all, first 4000 is passed in an argument named `size` , then, "obviously", you can expect False to become an argument named `a_k ilobyte_is_1024_bytes` . But it doesn't work in Python. Since there is a named argument, all arguments to the right of it must also be named.

Writing readable code

I will not spread my fingers out in front of you and torment you with a long lecture about the importance of documenting code. Just know that the code is written once, but read many times, and the most important reader of your code is yourself, six months after writing (that is , everything is already forgotten, and suddenly you need to fix something). It's easy to write readable code in Python. Use this advantage and in six months you will say “thank you” to me.

Docstrings

Functions in Python can be documented by providing them with documentation string (Eng. *Documentation : string* , abbreviated docstring). In our program, the approxi mate_size () function has a docstring :

def approximate_size (size , a_kilobyte_is_1024_bytes = True) : " 'Converts file size to human-readable form. Key arguments: size - file size in bytes a_kilobyte_is_1024_bytes - if True (default), powers of 1024 are used if False, powers of 1000 are used Returns: text string (string) " '

Every feature deserves good documentation.

Triple quotes [1] are used to specify strings [2] containing multiline text. Anything between the start and end quotes is part of the same data line, including line breaks, spaces at the beginning of each line of text, and other quotes. You can use them anywhere, but you will most often see them in docstring definitions.

> Triple quotes are also an easy way to define a string containing single (apostrophes) and double quotes, similar to qq /.../ in Perl 5 .

Everything in triple quotes is a function docstring describing what that function does. The docstring, if any, must begin the function body, that is, it is on the next line immediately below the function declaration. Strictly speaking, you are not required to write documentation for each of your functions, but it is always advisable to do so. I know you've been buzzing about documenting code by now, but Python gives you an added incentive - docstrings are available at runtime as a function attribute.

> Many Python IDEs use docstrings to display context-sensitive help, and when you type the name of a function, its documentation appears in a tooltip. This can be incredibly useful, but these are just docstrings that you write yourself.

Search Path operator `import`

Before going any further, I want to briefly talk about the library search paths. When you try to import a module (using an import statement), Python looks for it in several places. In particular, it searches all directories listed in sys . path . It is simply a list that can be easily viewed and modified using standard

list methods. (You will learn more about lists in the chapter Built-in Data Types.)

>>> import sys ① >>> sys . path ② ['' , '/usr/lib/python31.zip' , '/usr/lib/python3.1' , '/ us r / lib / python3.1 / plat-linux2 @ EXTRAMACHDEPPATH @' , '/ usr /lib/python3.1/lib-dynload ' , ' /usr/lib/python3.1/dist-packages' , '/usr/local/lib/python3.1/dist-packages'] >>> sys ③ < module 'the sys' (built- in) > >> > the sys . path . insert (0 , '/ home / mark / diveintopython3 / examples') ④ >>> sys . path ⑤ ['/ home / mark / diveintopython3 / examples' , ' ' , ' /usr/lib/python31.zip ' , ' /usr/lib/python3.1 ' , ' /usr/lib/python3.1/plat -linux2 @ E XTRAMACHDEPPATH @ ' , ' /usr/lib/python3.1/lib-dynload ' , ' /usr/lib/python3.1/dist-packages ' , ' /usr/local/lib/python3.1/ dist-packages']

① Importing the sys module makes all of its functions and attributes available.

② sys . path is a list of directory names specifying the current search

path. (Yours will look different, depending on your operating system, the version of Python you are using, and where it was installed.) Python will look in these directories (in the given order) for a file with a ".py" extension whose name matches what you are trying to import.

③ Actually, I deceived you; the actual state of affairs is a little more complicated, because not all modules are in files with a ".py" extension. Some of them, like the sys module , are built-in; they are soldered into Python itself. Built-in modules behave exactly like regular modules, but their source code is not available because they were not written in Python! (The sys module is written in C.)

④ You can add a new directory to the search path by adding the directory name to the sys . path while Python is running, and then Python will look at it along with the rest as soon as you try to import the module. The new search path will be valid for the entire Python session.

⑤ By running sys . path . in sert (0 , new_path) , you inserted the new directory at the top of the sys . path , and therefore at the beginning of the module search path. Almost always, this is exactly what you want. In the event of a name conflict (for example, if Python comes with version 2 of a library, and you want to use version 3), this trick ensures that your modules are found and used, and not those that come with Python.

Everything is an object

In case you missed it, I just said that functions in Python have attributes, and those attributes are available at runtime. A function, like everything else in Python, is an object.

Start an interactive Python shell and repeat after me:

>>> import humansize ① >>> print (humansize. approx imate_size (4096 , True)) ② 4.0 KiB >>> print (humansize. approximate_size .__ doc__) ③ Converts a file size to human-readable form. Key arguments: size - file size in bytes a_kilobyte_is_1024_bytes - if True (default) , powers of 1024 are used; if False , powers of 1000 are used Returns: text string (string)

① The first line imports the `humansize` program as a module , a piece of code that can be used interactively or from another Python program. After the module has been imported, you can access all of its public functions, classes, and attributes. Imports are used both in modules to access the functionality of other modules, and in the interactive Python shell. This is a very important idea, and you will come across it more than once in the pages of this book.

② When you want to use a function defined in an imported module, you need to add the module name to its name. So you can't just use approximate_size , it must be humansize. approximate_size . If you've used classes in Java , this should be familiar to you.

③ Instead of calling the function (as you might expect), you asked for one of its attributes - __doc__ .

> The import statement in Python is similar to the require in Perl . After importing in Python, you refer to module functions as `module.function` ; after require in Perl, the name `module :: function is` used to refer to `module functions` .

What is an object?

In Python, everything is an object, and any object can have attributes and methods. All functions have a standard __doc__ attribute that contains the docstring defined in the function's source code. The sys module is also an object with (among other things) an attribute called path . Etc.

But we have not received an answer to the main question: what is an object? Different programming languages define an "object" in different ways. Some

believe that all objects must have attributes and methods. In others, that objects can be subclassed. In Python, the definition is even less clear. Some objects do not have attributes or methods, although they might. Not all objects are subclassed. But everything is an object in the sense that it can be assigned to a variable or passed to a function as an argument.

You may have come across the term " first class object " in other programming books. In Python features - *first-class objects* . A function can be passed as an argument to another function. Modules - *First-class facilities* . The entire module can be passed as an argument to a function. Classes are first class objects, and their individual instances are also first class objects.

This is very important, so I'll repeat it in case you missed the first few times: *everything in Python is an object* . Strings are objects. Lists are objects. Functions are objects. Classes are objects. Instances of classes are objects. And even modules are objects.

Indentation

Functions in Python do not have explicit begin and end statements , nor curly braces to indicate where the function code begins and ends. Separators are only colon (:) and the indentation of the code itself.

def approximate_size (size , a_kilobyte_is_1024_bytes = True) : ① if size < 0 : ② raise ValueError ('the number must be non-negative') ③④ multiple = 1024 if a_kilobyte_is_1024_bytes else 1000 for suffix in SUFFIXES [multiple] : ⑤ size / = multiple if size < multiple: return '{0: .1f} {1 }' . format (size , suffix) raise ValueError ('number is too large')

① Code blocks are identified by their indentation. By "blocks of code" I mean functions, if blocks , for and while loops, and so on. Increasing

indentation starts a block, while decreasing indentation ends. No brackets, no keywords. This means that whitespace is important, and so is the number. In this example, the function code is punctuated with four spaces. It doesn't have to be exactly four spaces, it's just that their number must be constant. The first line encountered without indentation will mean the end of the function.

② The if statement must be followed by a block of code. If, as a result of evaluating the conditional expression, it turns out to be true, then the indented block will be executed, otherwise the transition to the else block (if any) will occur . Note that there are no parentheses around the expression.

③ This line is inside the if block . The raise statement raises an exception (of type ValueError), but only if size < 0 .

④ This is *not the* end of the function. Completely blank lines are not counted. They can improve the readability of your code, but they cannot serve as code block separators. The function code block continues on the next line.

⑤ The for loop also starts a block of code. Code blocks can contain multiple lines, namely , as many lines have the same indentation. This for loop contains three lines of code. There are no other syntactic constructs for describing multi-line code blocks. Just indent and you will be happy!

After you sort through the internal contradictions and draw a couple of snide analogies with Fortran , you will become friends with indents and begin to see their benefits. One of the main advantages is that all Python programs look roughly the same, since indentation is a language requirement, not a style issue. This makes it easier to read and understand Python code written by other people.

> Python uses carriage returns to separate statements, and colon and indentation to separate blocks of code. In C ++ and Java use the semicolons to separate operators and braces for code blocks.

Exceptions

Exceptions (English *exceptions* - abnormal, exceptional situations that

require special handling) are used throughout Python. Literally every module in the Python standard library uses them, and Python itself calls them in many situations. You will find them many times in the pages of this book.

What is an Exception? This is usually a mistake, a sign that something went wrong. (Not all exceptions are errors, but that doesn't matter for now.) In some programming languages, it is customary to return an error code, which you then *check* . It's common in Python to use the exceptions that you *handle* .

When an error occurs in the Python shell, it prints out some details about the exception and how it happened, and that's it. This is called an *unhandled* exception. When this exception was thrown, no code was found to notice it and handle it properly, so it popped up to the very top level , the Python shell, which dumped some debugging information and calmed down. This is not so bad in the shell, but if it happens while a real program is running, the whole program will crash with a crash unless the exception is handled. Maybe this is what you need, or maybe not.

> Unlike Java , functions in Python do not contain any declarations about what kind of exceptions they can throw. You decide which of the possible exceptions to catch.

The result of an exception is not always a complete crash of the program. Exceptions can be *handled* . Sometimes exceptions are thrown because of real bugs in your code (for example, accessing a variable that doesn't exist), but sometimes an exception is something that you can foresee. If you open a file, it may not exist. If you import a module, it may not be installed. If you are connecting to a database, it may not be accessible or you may not have sufficient rights to access it. If you know that a line of code might throw an exception, then you should handle it with a try ... except block .

> Python uses try ... except blocks to handle exceptions and a raise statement to raise them. Java and C ++ use try ... catch blocks to handle exceptions and the throw statement to throw them.

Function of the approximate_size () throws an exception in two different cases: if passed her size (size bed) more than the function can process, or if it is less than zero.

if size < 0 : raise ValueError ('the number must be non-negative')

The syntax for raising exceptions is fairly simple. You need to write a raise statement , followed by the name of the exception and, optionally, an explanatory line for debugging. The syntax is similar to a function call. (Actually, exceptions are implemented as classes, and the raise statement simply instantiates the ValueError class and passes the string 'the number must be non-negative' to its initialization method . But we're getting ahead of ourselves!)

It is not necessary to handle the exception in the function that called it. If one function does not handle it, the exception is passed to the function that called this one, then to the function that called the caller, and so on "up the stack." If an exception is nowhere to be processed, the program will fall, and Python will print "promotion stack" (Engl. *Traceback*) to standard error - and that's the end. Again, this may be exactly what you want - it depends on what your program is doing.

Catching import errors

One of Python's built-in exceptions is ImportError , which is thrown when a module cannot be imported. This can happen for several reasons, the simplest of which is the absence of a module in the search path, an import statement . What can be used to include optional features in the program. For example, the `chardet` library provides the ability to automatically detect character encoding. Suppose your program wants to use this library if it exists, or continue quietly if the user has not installed it. You can do this with a try ... except block .

```
try : import chardet except ImportError : chardet = None
```

After that, you can check for the presence of the `chardet` module `with a` simple if :

```
if chardet: # do something else : # continue further
```

Another common use of the ImportError exception is choosing between two modules that provide the same interface (API), and one of them is preferable to the other (it may be faster or requires less memory). To do this, you can try to import one module first, and if this fails, then import another. For example, the XML chapter talks about two modules that implement the same API, the so-called ElementTree API . The first is lxml , a third-party module that you need to download and install yourself. The second is xml . etree . ElementTree is slower but is part of the Python 3 standard library .

try : from lxml import etree except ImportError : import xml . etree . ElementTree as etree

Executing this try ... except block will import one of the two modules named etree . Since both modules implement the same API, there is no need in the following code to check which of these modules was imported. And since the imported module is named etree anyway , then you don't have to insert extra ifs to access different modules.

Unbound variables

Let's take another look at this line of the approximate_size () function :

multiple = 1024 if a_kilobyte_is_1024_bytes else 1000

We have not declared the multiple variable anywhere , we just assigned a value to it. It's okay, Python allows you to do that. What it won't let you do is access a variable that hasn't been assigned a value. If you try to do this, a NameError (name error) exception will be thrown .

>>> x
Traceback (most recent call last): File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined >>> x = 1 >>> x 1

Translating the shell message:

> Unrolling the stack (list of recent calls):
> File "<stdin>", line 1, <module>
>
> NameError: name 'x' is undefined

One day you will thank Python for this.

Everything is case sensitive

All names in Python are case sensitive - names of variables, functions, classes, modules, exceptions. Everything that can be read, written, invoked, created, or imported is case sensitive.

```
>>> an_integer = 1
>>> an_integer
1
>>> AN_INTEGER
```

T raceback (most recent call last): File "<stdin>", line 1, in <module> NameError: name 'AN_INTEGER' is not defined >>> An_Integer Traceback (most recent call last): File "<stdin>", line 1, in <module> NameError: name 'An_Integer' is not defined >>> an_inteGer Traceback (most recent call last): File "<stdin>" , line 1, in <module> NameError: name 'an_inteGer' is not defined

Translating shell messages:

> Unrolling the stack (list of recent calls):
> File "<stdin>", line 1, <module>
>
> NameError: name '<name>' is undefined

Etc.

Running scripts

Everything in Python is an object.

Python modules are objects that have several useful attributes. And this circumstance can be used for simple testing of modules, when writing them, by including a special block of code that will be executed when the file is run from the command line. Take a look at the last lines of `humansize.py` :

if __name__ == '__main__' : print (approximate_size (1000000000000 , False)) print (approximate_size (1000000000000))

Like C , Python uses the == operator to test for equality and the = operator for assignments. But unlike C, Python doesn't support assignment within another expression, so you can't accidentally assign a value instead of checking for equality.

So what makes this if block special? All modules, like objects, have a built-in __name__ (name) attribute . And the meaning of this attribute depends on how the module is used. If a module is imported, then __name__ is equal to the module file name, without the extension and directory path.

>>> import humansize >>> humansize .__ name__ 'humansize'

But the module can be run directly as an independent program, in this case __n ame__ will take on a special meaning - __main__ . Python will evaluate the value of the conditional expression in the if statement , determine if it is true, and execute the if block of code . In this case, two values will be printed.

c: \ home \ diveintopython3> c: \ python31 \ python.exe humansi ze.py
1.0 TB
931.3 GiB

And this is your first Python program!

Further reading

- PEP 257: Docstring Conventions explains how a good docstring is different from a great one.
- Python Tutorial: Documentation Strings also covers this question.
- PEP 8: Style Guide for Python Code discusses good style of indentation.
- *The Python Reference Manual* explains what " everything in Python is an object " means because some people are pedants who like lengthy discussions of this kind of thing.

Notes

1. In English, the apostrophes that frame the text are already single quotes. - *Approx. per.*
2. This means the data type string (string). - *Approx. per.*

Built-in data types

At the beginning of all philosophy lies wonder, study propels it forward, ignorance kills it.

Michel de Montaigne

Immersion

Put your first Python program aside for a moment and let's talk about data types. In Python, every value has a type, but there is no need to explicitly specify the types of variables. How it works? Based on the first assignment of a value to a variable, Python determines its type and then keeps track of it on its own.

Python has many built-in data types. The most important ones are:

1. **Boolean** , it can take one of two values - True (true) or False (false).
2. **Numbers** can be integers (1 and 2), floating point (1.1 and 1.2) fractional (1 / 2 and 2 / 3) and even complex .

3. **Strings** are sequences of Unicode characters , such as an HTML document.
4. **Bytes** and **byte arrays** , such as a JPEG image file .
5. **Lists** are ordered sequences of values.
6. **Tuples** are ordered, immutable sequences of values.
7. **Sets** are unordered collections of values.
8. **Dictionaries** are unordered collections of key-value pairs.

Of course, there are many other types of data as well. Everything in Python is an object, so it also contains types such as *module* , *function* , *class* , *method* , *file* , and even *compiled code* . Some of these you have already seen: modules have names, functions have docstrings, and so on. You will learn about classes in the chapter "Classes and Iterators"; with files - in the chapter "Files".

Strings and bytes are just as complex as they are important, which is why they have their own chapter. But first, let's get to know the rest of the types.

Boolean values

Almost any expression can be used in a logical context.

A boolean data type can take one of two values: true or false. In Python, there are two constants with friendly names True The (from Eng. *To true* - true) and False The (from Eng. *To false* - false) that can be used for the direct assignment of logical values. The result of evaluating expressions can also be a Boolean value. In certain places (for example, in an if statement), Python expects an expression to evaluate to a Boolean value. Such places are called *logical contexts* . Almost any expression can be used in a logical context, Python will try to determine if it is true anyway. For this, there are separate sets of rules, for different data types, indicating which of their values are considered true and which are false in a logical context. (This idea will become clearer as you go through specific examples later in this chapter.)

For example, consider the following excerpt from humansize.py:

if size < 0 : r aise ValueError ('the number must be non-negative')

Here the variable size and the value 0 are of type integer, and the < sign between them is a numeric operator. The result of evaluating the expression size < 0 will always be a boolean value. You can verify this for yourself using the interactive Python shell:

>>> size = 1 >>> size < 0 False >>> size = 0 >>> size < 0 False >>> size = -1 >>> size < 0 True

Due to some legacy circumstances from Python 2, booleans can be treated like numbers. True as a 1 , and False as a 0 .

```
>>> True + True
2
>>> True - False
1
>>> True * False
0
>>> True / False
Traceback (most recent call last): File "<stdin>", line 1, in <module>
ZeroDivisionError : int division or modulo by zer o
```

Translating the shell message:

> Unrolling the stack (list of recent calls):
> File "<stdin>", line 1, <module>
>
> ZeroDivisionError: integer division by zero or remainder modulo zero

Oh oh oh! Don't do that! Forget even that I mentioned it.

Numbers

Numbers are awesome. There are so many of them, there is always

something to choose from. Python supports both integers and floating point . And there is no need to declare a type to distinguish them; Python determines it by the presence or absence of a decimal point.

>>> type (1) ① < class 'int' > >>> isinstance (1 , int) ② True >>> 1 + 1 ③ 2 >>> 1 + 1.0 ④ 2.0 >>> type (2.0) < class 'float' >

① You can use the type () function to check the type of any value or variable. As expected, the number 1 is of type int (integer).

② The isinstance () function can also be used to check if a value or variable is of a certain type.

③ Adding two int values results in the same int .

④ Adding int and float values results in float . To perform the addition operation, Python converts an int to a float , and returns a float as a result .

Convert integers to decimals and vice versa

As you just saw, some operations (such as addition) convert integers to floating point numbers if necessary. You can do this conversion yourself.

>>> float (2) ① 2.0 >>> int (2.0) ② 2 >>> int (2.5) ③ 2 >>> int (-2.5) ④ - 2 >>> 1.12345678901234567890 ⑤ 1.1234567890123457 >>> type (1000000000000000) ⑥ < class 'int' >

① You can explicitly convert an int to a float by calling the float () function .

② It is also not surprising that you can convert a float value to an int value using the int () function .

③ The int () function discards the fractional part of the number, rather than rounding it.

④ The int () function "rounds" negative numbers upward. It returns the integer part as does the " floor " (Eng. *Floor*), but simply discards the fractional part.

⑤ The precision of floating point numbers is 15 decimal places in the fractional part.

⑥ Integers can be as large as you want.

> Python 2 had separate types of integers: int and long . The int type has been limited to sys . maxint , which varied from platform to platform, but was usually $2^{32}-1$. Python 3 has only one integer type, which in most cases behaves like a long type in Python 2. See PEP 237 .

Basic number operations

Various operations can be performed on numbers.

```
>>> 11 / 2  ①  5.5 >>> 11 // 2  ②  5 >>> - 11 // 2  ③  - 6 >>> 11.0 // 2  ④
5.0 >>> 11 ** 2  ⑤  121 >>> 11 % 2  ⑥  1
```

① The / operator performs floating-point division. It returns a float even if the dividend and divisor are both int .

② The // operator performs an unusual kind of integer division. When the result is positive, you can assume that it is simply discarding (not rounding) the fractional part, but be careful with this.

③ When integer division is performed on negative numbers, the // operator rounds up the result to the nearest integer up. From a mathematical point of view, this is of course rounding down, since −6 is less than −5; but this can be confusing and you will expect the result to be "rounded" to -5.

④ The // operator does not always return an integer. If at least one of the operands - dividend or divisor - is of type float , then although the result will be rounded to the nearest integer, in reality it will also be of type float .

⑤ The ** operator performs exponentiation. 11^{2} is 121.

⑥ The % operator returns the remainder of an integer division. 11 divided by 2 is 5 and remainder 1, so here the result is 1.

> In Python 2, the / operator usually means integer division, but by adding a special directive to your code you can force it to perform floating point division. In Python 3, the / operator always means floating point division. See PEP 238 .

Fractions

Python is not limited to integers and floating point numbers. It can also do all those funny things that you learned in school in math classes and then safely forgot.

>>> import fractions ① >>> x = fractions.Fraction (1, 3) ② >>> x Fraction (1, 3) >>> x * 2 ③ Fraction (2, 3) >>> fractions.Fraction (6, 4) ④

Fraction (3, 2) >>> fractions.Fraction (0, 0) ⑤ Traceback (most recent call last): File "<stdin>", line 1, in <module> File "fractions.py ", line 96, in __new__ r aise ZeroDivisionError ('Fraction (% s, 0)' % numerator) ZeroDivisionError: Fraction (0, 0)

Translating the shell message:

Unrolling the stack (list of recent calls):

File "<stdin>", line 1, in <module>
The file "fractions.py", line 96, at __new__

raise ZeroDivi sionError (' Fraction (% s, 0)' % numerator)
ZeroDivisionError: Fraction (0, 0)

① Before using fractions, import the fractions module .

② To define a fraction, create an object of the Fraction class and pass the numerator and denominator to it.

③ All normal mathematical operations can be performed with fractions.

They all return a new object of the Fraction class .

④

The Fraction object will automatically reduce fractions.

⑤ Python has enough common sense not to create fractions with a zero denominator.

Trigonometry

You can also work with basic trigonometric functions in Python .

>>> import math >>> math . pi ① 3.1415926535897931 >>> ma th . sin (math . pi / 2) ② 1.0 >>> math . tan (math . pi / 4) ③ 0.99999999999999989

① The math module contains a constant π - the ratio of the circumference of a circle to its diameter.

② The math module contains all the basic trigonometric functions, including sin () , cos () , tan () , and their variants like asin () .

③ Note, however, that the precision of calculations in Python is not infinite. The expression must return 1.0 , not 0.99999999999999989 .

Numbers in a logical context

Null values are false, non-null values are true.

You can use numbers in a logical context such as an if statement . Null values are false, non-null values are true.

>>> def is_it_true (anythin g) : ① ... if anything: ... print (" yes , that's true ") ... else : ... print (" no , that's false ") ... > >> is_it_true (1) ② yes , this is true >>> is_it_true (- 1) yes , this is true >>> is_it_true (0) no , this is false >>> is_it_true (0.1) ③ yes , this is true >>> is_it_true (0.0) no ,

that's false >>> import fractions >>> is_it_true (fractions. Fraction (1 , 2)) ④ yes , that's true >>> is_it_true (fractions. Fraction (0 , 1)) no , that's false

① Did you know that you can define your own functions in the Python interactive shell? Just press the ↵ Enter key at the end of each line, and to finish typing, press the ↵ Enter key on a blank line.

② In a logical context, nonzero integers are true; value 0 is false.

③ Nonzero floating point numbers are true; value 0.0 is false. Be careful with this! If there is the slightest rounding error (as you may have seen in the previous section, this is quite possible), then Python will check the value 0.0000000000001 instead of 0.0 and accordingly return the Boolean value True .

④ Fractions can also be used in a logical context. Fraction (0 , n) is false for all values of n . All other fractions are true.

Lists

Lists are the workhorse of Python. When I say “list,” you’re probably thinking, “this is an array, whose size I have to specify in advance, and which can only store elements of one type,” and so on , but this is not so. Lists are much more interesting.

Lists in Python are like arrays in Perl 5 . There, variables containing arrays always start with the @ symbol ; in Python, variables can be named whatever you want, the language keeps track of the type itself.

In Python, a list is more than an array in Java (although a list can also be used as an array, if that's what you really want in life). More precisely, there will be an analogy with the Java ArrayList class , which can store arbitrary objects and dynamically expand as new elements are added.

List creation

It's easy to create a list: enter all values, separated by commas, in square brackets.

>>> a_list = ['a' , 'b' , 'mpilgrim' , 'z' , 'example'] ① >>> a_list ['a' , 'b' , 'mpilgrim' , 'z' , 'example '] >>> a_list [0] ② ' a ' >>> a_list [4] ③ ' example ' >>> a_list [- 1] ④ ' example ' >>> a_list [- 3] ⑤ ' mpilgrim '

① First, you defined a list of five items. Please note they retain their original order. This is no coincidence. A list is an ordered collection of items.

② The list can be used as a zero-based array. The first element of a non-empty list will always be a_list [0] .

③ The last element of this five-element list will be a_list [4] , because the numbering of elements in the list always starts at zero.

④ Using a negative index, you can refer to items by their number from the end of the list. The last element of a non-empty list will always be a_list [- 1] .

⑤ If negative indices confuse you, just think of them like this: a_list [-n] == a_list [len (a_list) - n] . In our example, a_list [- 3] == a_list [5 - 3] == a_list [2] .

Slitting the list

a_list [0] - the first element of the a_list .

After the list is created, you can get any part of it as a list. This is called *"slicing" - a slice of the list* .

>>> a_list
['a' , 'b' , 'mpilgrim' , 'z' , 'example'] >>> a_list [1 : 3] ① ['b' , 'mpilgrim'] >>> a_list [1 : - 1] ② ['b' , 'mpilgrim' , 'z'] >>> a_list [0 : 3] ③ ['a' , 'b' , 'mpilgrim'] >>> a_list [: 3] ④ ['a' , 'b' , 'mpilgrim'] >>> a_list [3 :] ⑤ ['z' , 'example'] >>> a_list [:] ⑥ ['a' , 'b' , 'mpilgrim ' , ' z ' , ' example ']

① You can get a slice of a list by specifying two indices. The result is a new list that includes the elements of the original in the same order, starting at the first index of the slice (in this case a_list [1]) up to the last, but not including it (in this case a_list [3]).

② The slice works even if one or both of the indices are negative. If this helps you, you can think of it this way: the list reads from left to right, the first slice index identifies the first element you want, and the second index identifies the first element you don't need. The return value is always in between.

③ Lists are numbered starting at zero, so a_list [0 : 3] returns the first three elements of the list, starting at a_list [0] and ending at (but not including) a_list [3] .

④ If the left slice index is 0, you can omit it, 0 will be implied. So, a_list [: 3] is the same as a_list [0 : 3] , because the leading 0 is implied.

⑤ Likewise, if the right slice index is the length of the list, you can omit it. So, a_list [3 :] is the same as a_list [3 : 5] , because the list contains five elements. There is a clear symmetry here. In this five-element list, a_list [: 3] returns the first 3 elements, and a_list [3 :] returns the last two elements. In fact, a_list [: n] will always return the first n items, and a_list [n:] will return all the others, regardless of the length of the list.

⑥ If both list indices are omitted, all list items are included. But this is not the same as the original a_list variable . This is a new list that includes all the items in the original. The a_list [:] entry is the simplest way to get a complete copy of a list.

Adding items to the list

There are four ways to add items to the list.

>>> a_list = [' a'] >>> a_list = a_list + [2.0 , 3] ① >>> a_list ② ['a' , 2.0 , 3] >>> a_list. append (True) ③ >>> a_list ['a' , 2.0 , 3 , True] >>> a_list. extend (['four' , ' Ω ']) ④ >>> a_list ['a' , 2.0 , 3 , True , 'four' , ' Ω '] >>> a_list. insert (0 , ' Ω ') ⑤ >>> a_list [' Ω ' , 'a' , 2.0 , 3 , True , 'four' , ' Ω ']

① The + operator concatenates lists, creating a new list. The list can contain any number of items; there is no size limit (as long as there is available memory). However, if you care about memory, know that adding lists creates another list in memory. In this case, this new list is immediately assigned to the existing variable a_list . So this line of code actually implements a two-step process - addition and then assignment - which can (temporarily) take up a lot of memory if you're dealing with large lists.

② A list can contain any type of element, and the elements of the same list do not have to be of the same type. Here we see a list containing a string, a float, and an integer.

③ The append () method adds one element to the end of the list. (We now have *four* different data types listed !)

④ Lists are implemented as classes. "Creating" a list is actually creating an instance of a class. Thus , the list has methods that work with it. The extend () method takes one argument , a list, and adds each of its elements to the original list.

⑤ The insert () method inserts an item into the list. The first argument is the index of the first item in the list to be shifted from its position by the new item. List items do not have to be unique; for example, we now have two different items with the value 'Ω' : the first item a_list [0] and the last item a_list [6] .

In Python, the a_list. insert (0 , value) acts like the unshift () function in Perl . It adds an item to the beginning of the list, and all other items increment their index by one to free up space.

Let's take a closer look at the difference between append () and extend () .

>>> a_list = ['a' , 'b' , 'c'] >>> a_list. extend (['d' , 'e' , 'f']) ① >>> a_list ['a' , 'b' , 'c' , 'd' , 'e' , 'f'] >>> len (a_list) ② 6 >>> a_list [- 1] 'f' >>> a_list. append (['g' , 'h' , 'i']) ③ >>> a_list ['a' , 'b' , 'c' , 'd' , 'e' , 'f' , ['g' , 'h' , 'i']] >>> len (a_list) ④ 7 >>> a_list [- 1] ['g' , 'h' , 'i']

① The extend () method takes one argument, which is always a list, and adds each element of that list to a_list .

② If you take a list of three items and expand it with a list of three more items, you end up with a list of six items.

③ On the other hand, the append () method takes a single argument, which can be of any type. Here we call the append () method , passing in a list of three elements.

④ If you take a list of six items and add a list to it, you end up with ... a list of seven items. Why seven? Because the last item (which we just added) is a list. Lists can contain any type of data, including other lists. Perhaps this is what you need, perhaps not. But this is what you asked for and this is what you received.

Finding values in a list

>>> a_list = ['a', 'b', 'new', 'mpilgrim', 'new']
>>> a_list.count ('new') ① 2 >>> 'new' in a_list ② True > >> 'c' in a_list False >>> a_list.index ('mp ilgrim') ③ 3 >>> a_list.index ('new') ④ 2 >>> a_list.index ('c') ⑤ Traceback (innermost last): File "<interactive input>", line 1, in ? ValueError: list.index (x): x not in list

Translating the shell message :

Unrolling the stack (from external to internal):
File "<interactive input>", line 1, position ?

ValueError: list.index (x): x is not in the list

① As you might expect, the count method returns the number of occurrences of the specified value in the list.

② If all you need to know is whether a value is in the list or not, then the in operator is much faster than the count () method . The in operator always returns True or False ; it doesn't tell you exactly how many values are in the list of data.

③ If you need to know exactly where some value is in the list, then use the index () method . By default, it scans the entire list, but you can specify with the second argument the index (based on zero) at which to start the search, and even the third argument , the index at which to stop the search.

④ The index () method only finds the first occurrence of a value in the list. In this case, 'new' appears twice in the list: in a_list [2] and a_list [4] , but the index () method will only return the index of the first occurrence.

⑤ Contrary to your expectations, if the value is not found in the list, the index () method will throw an exception. [wa p- rob i n . with om]

Wait, what? Yes, that's right: the index () method throws an exception if it can't find a value in the list. You may have noticed unlike most other languages that return some invalid index (for example, - 1). If at first glance it may seem a little annoying, then I think in the future you will adopt this approach. This means that your program will crash where the problem was, instead of quietly stopping somewhere else. Remember - 1 is also a good index for lists . If the index () method returned - 1 , you could have had some unhappy evenings spent looking for bugs in your code!

Removing items from a list

Lists never contain breaks.

Lists can grow and shrink automatically. You have already seen how they can grow. There are also several different ways to remove items from the list.

>>> a_list = ['a' , 'b' , 'new' , 'mpilgrim' , 'new'] >>> a_list [1] 'b' >>> del a_list [1] ① >>> a_list ['a' , 'new' , 'mpilgrim' , 'new'] >>> a_list [1] ② 'new'

① You can use the del expression to remove a specific item from the list.

② If, after deleting the element with index 1, again try to read the value of the list with index 1, this will not cause an error. All elements after deletion shift their indices to "fill in the gap" that occurs after the element is deleted.

Don't know the index? It doesn't matter - you can delete an element by value.

>>> a_list.remove ('new') ① >>> a_list ['a', 'mpilgrim', 'new'] >>> a_list.remove ('new') ② >>> a_list ['a', 'mpilgrim'] >>> a_list.remove ('new') Traceback (most recent call last): File "<stdin>", line 1, in <module> ValueError: list.remove (x): x not in list

Translating shell messages:

Unrolling the stack (list of recent calls):
File "<stdin>", line 1, <module>

ValueError: list.remove (x): x is not in the list

① You can remove an item from the list using the remove () method . The remove () method takes a value as a parameter and removes the first occurrence of that value from the list. In addition, the indices of all elements following the deleted one will be shifted to "fill in the gap". Lists never contain breaks.

② You can call remove () as much as you like, but if you try to remove a value that is not in the list, an exception will be thrown.

Removing items from the list: extra round

Another interesting list method is pop () . The pop () method is another way to remove items from a list , but with one twist.

>>> a_list = ['a', 'b', 'new', 'mpilgrim']
>>> a_list.pop () ① 'mpilgrim' >>> a_list ['a', 'b', 'new'] >>> a_list.po p (1) ② 'b' >>> a_list ['a', 'new'] >>> a_list.pop () 'new' >>> a_list.pop () 'a' >>> a_list.pop () ③ Traceback (most recent call last): File "<stdin>", line 1, in <module> IndexError: pop from empty list

Translating the shell message:

Unrolling the stack (list of recent calls):
File "<stdin>", line 1, <module>

IndexError: pop from empty list

① If you call pop () with no arguments, it will remove the last item in the list and return the removed value.

② Using the pop () method, you can remove any item in the list. Just call the method with the index of the element. This element will be removed, and all elements after it will be shifted to "fill in the gap". The method returns the value removed from the list.

③ The pop () method throws an exception on an empty list.

Calling pop () with no arguments is equivalent to calling pop () in Perl . It removes the last item from the list and returns the removed value. The Perl programming language also has a shift () function that removes the first element and returns its value. In Python, this is equivalent to a_list. pop (0) .

Lists in a logical context

Empty lists are false, all others are true.

You can also use a list in a logical context , for example in an if statement :

>>> def is_it_true (anything) : ... if anything: ... print (" yes , that's true ") ... else : ... print (" no , that's false ") ... >>> is_it_true ([]) ① no , this is false >>> is_it_true (['a']) ② yes , this is true >>> is_it_true ([False]) ③ yes , this is true

① In a logical context, an empty list is false.

② Any list with at least one element is true.

③ Any list with at least one element is true. The element values are not important.

Tuples

A tuple is an immutable list. A tuple cannot be modified in any way after it has been created.

>>> a_tuple = ("a" , "b" , "mpilgrim" , "z" , "example") ① >>> a_tuple ('a' , 'b' , 'mpilgrim' , 'z' , 'example ') >>> a_tuple [0] ② ' a ' >>> a_tuple [- 1] ③ ' example ' >>> a_tuple [1 : 3] ④ (' b ' , ' mpilgrim ')

① A tuple is defined in the same way as a list, except that the set of elements is enclosed in parentheses rather than squares.

② The elements of a tuple are specified in a specific order, just like in a list. The elements of a tuple are zero-indexed, like the elements of a list, so the first element of a non-empty tuple is always a_tuple [0] .

③ Negative index values are counted from the end of the tuple, as in a list.

④ Slicing a tuple is similar to slicing a list. When a list is sliced, a new list is obtained; when a slice of a tuple is created, a new tuple is produced.

The main difference between tuples and lists is that tuples cannot be modified. Technically speaking, a tuple is an immutable object . In practice, this means that they do not have methods to change them. Lists have methods such as append () , extend () , insert () , remove () , and pop () . Tuples have none of these methods. You can take a slice from a tuple (since this will create a new tuple), you can check if the tuple contains an element with a specific value (since this action will not change the tuple), and ... that's all .

continuation of the previous example >>> a_tuple ('a', 'b', 'mpilgrim', 'z', 'example') >>> a_tuple.append ("new") ① Traceback (innermost last): File " < interactive input> ", line 1, in ? AttributeError: ' tuple' object has no attribute 'append' >>> a_tuple.remove ("z") ② Traceback (innermost last): File "<interactive input>", line 1, in ? AttributeError: 'tuple' object has no attribute 'remove' >>> a_tuple.index ("example") ③ 4 >>> "z" in a_tuple ④ True

Translating shell messages:

Unrolling the stack (from external to internal):
File "<interactive input>", line 1, position ?

AttributeError: 'tuple' object has no attribute '<attribute>'

① You cannot add elements to a tuple. Tuples do not have append () or extend () methods .

② You cannot remove items from a tuple. Tuples have no remove () or pop () methods .

③ You can search for elements in tuples as this does not change the tuple.

④ You can also use the in operator to check if an element exists in a tuple.

So where do tuples come in handy?

- Tuples are faster than lists. If you are defining an immutable set of values and all you want to do with it is iterate over it, use a tuple instead of a list.
- Tuples make your code safer if you have "write-protected" data that shouldn't change. Using tuples instead of lists saves you the hassle of using the assert statement to make it clear that data is immutable and that special effort (and special function) is needed to get around it.
- Some tuples can be used as dictionary keys (specifically, tuples containing *immutable* values, such as strings, numbers, and other tuples). Lists can never be used as dictionary keys, because lists are mutable objects.

Tuples can be converted to lists and vice versa. The built-in tuple () function takes a list and returns a tuple of all its elements, the list () function takes a tuple and returns a list. Basically, tuple () freezes the list, while list () unfreezes the tuple.

Tuples in a logical context

You can use tuples in a logical context such as an if statement .

>>> def is_it_true (anything) : ... if anything: ... print (" yes , that's true ") ... else : ... print (" no , that's false ") ... >>> is_it_true (()) ① no , this is false >>> is_it_true (('a' , 'b')) ② yes , this is true >>> is_it_true ((False ,)) ③ yes , this is true >>> type ((False)) ④ < class 'bool' > >>> type ((False ,)) < class 'tuple' >

① In a logical context, an empty tuple is false.

② Any tuple with at least one element is true.

③ Any tuple with at least one element is true. The element values are not important. But what is this comma doing here?

④ To create a tuple of one element, you must put a comma after it. Without the comma, Python assumes you just added another pair of parentheses, which doesn't do anything wrong, but it doesn't create a

tuple either.

Assigning multiple values at once

Here's a cool programming trick: in Python, you can use tuples to assign a value to multiple variables at once.

>>> v = ('a' , 2 , True) >>> (x , y , z) = v ① >>> x 'a' >>> y 2 >>> z True

① v is a tuple of three elements and (x , y , z) is a tuple of three variables. Assigning one to another results in the assignment of each value from v to each variable in that order.

This is not the only way to use it. Suppose you want to name a range of values. You can use the built-in range () function to quickly assign multiple consecutive values at once.

>>> (MONDAY , TUESDAY , WEDNESDAY , THURSDAY , FRIDAY , SATURDAY , SUNDAY) = range (7) ① >>> MONDAY ② 0 >>> TUESDAY 1 >>> SUNDAY 6

① The built-in function range () creates a sequence of integers. (Strictly speaking, range () returns an iterator, not a list or tuple, but you will learn the difference a little later.) MONDAY , TUESDAY , WEDNESDAY , THUR SDAY , FRIDAY , SATURDAY , and SUNDAY are definable variables . (This example is taken from the module -in calendar , a little funny module that displays a calendar about the program cal of the UNIX . Constants defined integer type for days

of the week in this module.)

② Now each variable is assigned a specific value: MONDAY is 0, TUESDAY is 1, and so on.

You can also use multiple variable assignments to create functions that return multiple values by simply returning a tuple containing those values. At the point in the program where the function was called, the return value can be used as a tuple in its entirety, or assigned the values of several separate variables. This technique is used in many standard Python libraries, including the os module , which you will learn about in the next chapter .

The sets

A set is a bag of unordered unique values. One set can contain values of any type. If you have two sets, you can perform any of the standard operations on them, such as union, intersection, and difference.

Creating a set

Let's start from the very beginning. It is very easy to create a set .

>>> a_set = { 1 } ① >>> a_set { 1 } >>> type (a_set) ② < class 'se t' > >>> a_set = { 1 , 2 } ③ >>> a_set { 1 , 2 }

① To create a set with one value, place it in curly braces (`{}`).

② Sets are actually implemented as classes, but don't worry about that for now.

③ To create a set with multiple values, separate them from each other with commas and place them inside curly braces.

You can also create a set from the list .

>>> a_list = ['a' , 'b' , 'mpilgr im' , True , False , 42] >>> a_set = set (a_list) ① >>> a_set ② { 'a' , False , 'b' , True , 'mpilgrim' , 42 } >>> a_list ③ ['a' , 'b' , 'mpilgrim' , True , False , 42]

① To create a set from a list, use the set () function . (Pedants who know how sets are implemented will note that this is actually an instantiation of a class, not a function call. I *promise* you will learn the difference later in this book. For now, just know that set () behaves like a function. and returns a set.)

② As I mentioned earlier, a set can contain values of any type. And, as I mentioned earlier, the sets are *unordered* . This set does not remember the original order of the list from which it was created. If you add items to a set, it doesn't remember the order in which they were added.

③ The original list has not changed.

Don't have values yet? No problems. You can create an empty set.

>>> a_set = set () ① >>> a_ set ② set () >>> type (a_set) ③ < class 'set' > >>> len (a_set) ④ 0 >>> not_sure = {} ⑤ > >> type (not_sure) < class 'dict' >

① To create an empty set, call set () with no arguments.

② The printed representation of an empty set looks a little odd. You probably expected to see `{}` ? This would mean an empty dictionary, not an empty set. You will learn more about dictionaries later in this chapter.

③ Despite the odd print performance, there are *indeed* many ...

④ ... and this set does not contain any element.

⑤ Due to historical quirks from Python 2, you cannot create an empty set with two curly braces. In fact, they create an empty dictionary, not a set.

Change set

There are two ways to add items to an existing set: the add () method and the update () method .

>>> a_set = { 1 , 2 } >>> a_set. add (4) ① >>> a_set { 1 , 2 , 4 } >>> len (a_set) ② 3 >>> a_set. add (1) ③ >>> a_set { 1 , 2 , 4 } >>> len (a_set) ④ 3

① The add () method takes one argument, which can be of any type, and adds the given value to the set.

② The set now contains 3 elements.

③ Sets are bags of *unique values* . If you try to add a value that is already in the set, nothing happens. This will not result in an error; just zero action.

④ This set *still has* 3 elements.

>>> a_set = { 1 , 2 , 3 } >>> a_set { 1 , 2 , 3 } >>> a_set. update ({ 2 , 4 , 6 }) ① >>> a_set ② { 1 , 2 , 3 , 4 , 6 } >>> a_set. update ({ 3 , 6 , 9 } , { 1 , 2 , 3 , 5 , 8 , 13 }) ③ >>> a_set { 1 , 2 , 3 , 4 , 5 , 6 , 8 , 9 , 13 } >> > a_set. update ([10 , 20 , 30]) ④ >>> a_set { 1 , 2 , 3 , 4 , 5 , 6 , 8 , 9 , 10 , 13 , 20

, 30 }

① The update () method takes one argument , a set, and adds all of its elements to the original set. It is as if you called the add () method and passed all the elements of the set to it in turn.

② Duplicate values are ignored because the set cannot contain duplicates.

③ Actually, you can call the update () method with any number of parameters. When called with two sets, the update () method adds all the elements of both sets to the original set (skipping duplicates).

④ The update () method can accept objects of various types, including lists. When a list is passed to it, it adds all of its elements to the original set.

Removing elements from a set

There are three ways to remove individual values from a set. The first two , discard () and remove () , are slightly different .

>>> a_ set = {1, 3, 6, 10, 15, 21, 28, 36, 45}
>>> a_set
{1, 3, 36, 6, 10, 45, 15, 21, 28}
>> > a_set.discard (10) ① >>> a_set {1, 3, 36, 6, 45, 15, 21, 28} >>> a_set.discard (10) ② >>> a_set {1, 3, 36, 6, 45, 15, 21, 28 } >>> a_set.remove (21) ③ >>> a_set {1, 3, 36, 6, 45, 15, 28} >>> a_set.remove (21) ④ Traceback (most recent call last): File "<stdin>", line 1, in <module> KeyError: 21

Translating the shell message:

Unrolling the stack (list of recent calls):
File "<stdin>", line 1, <module>

KeyError: 21

① The discard () method takes a single value as an argument and removes that value from the set.

② If you call the discard () method and pass it a value that is not in the set, nothing happens, just a null action.

③ The remove () method also takes a single value as an argument, and also removes it from the set.

④ Here's the difference: if the value is not in the set, the remove () method will throw a KeyError exception .

Like lists, sets have a pop () method .

>>> a_set = {1, 3, 6, 10, 15, 21, 28, 36, 45}
>>> a_set.pop () ① 1 >>> a_set.pop () 3 >>> a_set.pop ( ) 36 >>> a_set {6, 10, 45, 15, 21, 28} >>> a_set.clear () ② >>> a_set set () >>> a_set.pop () ③ Traceback (most recent call last): File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'

Translating the shell message:

Unrolling the stack (list of recent calls):
File "<stdin>", line 1, <module>

KeyError: 'pop from empty set'

① The pop () method removes one element from the set and returns its value. However, since the sets are unordered, this is not the "last" item in the set, so it is impossible to control which value has been removed. An arbitrary element is removed.

② The clear () method removes all elements of the set, leaving you with an empty set. This is equivalent to writing a_set = set () , which will create a new empty set and overwrite the previous value of a_set .

③ An attempt to pop an element from an empty set will throw a KeyError exception .

Basic set operations

The set type in Python supports several basic set operations.

>>> a_set = { 2 , 4 , 5 , 9 , 12 , 21 , 30 , 51 , 76 , 127 , 195 } >>> 30 in a_set ① True >>> 31 in a_set False >>> b_set = { 1 , 2 , 3 , 5 , 6 , 8 , 9 , 12 , 15 , 17 , 18 , 21 } >>> a_set. union (b_set) ② { 1 , 2 , 195 , 4 , 5 , 6 , 8 , 12 , 76 , 15 , 17 , 18 , 3 , 21 , 30 , 51 , 9 , 127 } >>> a_set. intersection (b_set) ③ { 9 , 2 , 12 , 5 , 21 } >>> a_set. difference (b_set) ④ { 195 , 4 , 76 , 51 , 30 , 127 } >>> a_set. symmetric_difference (b_set) ⑤ { 1 , 3 , 4 , 6 , 8 , 76 , 15 , 17 , 18 , 195 , 127 , 30 , 51 }

① To check if a value is in a set, use the in operator . It works the same way as it does for lists.

② The union () method returns a new set containing all the elements of each set.

③ The intersection () method returns a new set containing all the elements in both the first set and the second.

④ The difference () method returns a new set containing all the elements that are in a_set but not in b_set .

⑤ The symmetric_difference () method (symmetric difference) returns a new set that contains only the unique elements of both sets.

Three of these methods are symmetrical.

continuation of the previous example >>> b_set. symmetric_difference (a_set) ① { 3 , 1 , 195 , 4 , 6 , 8 , 76 , 15 , 17 , 18 , 51 , 30 , 127 } >>> b_set. symmetric_difference (a_set) == a_set. symmet ric_difference (b_set) ② True >>> b_set. union (a_set) == a_set. union (b_set) ③ True >>> b_set. intersection (a_set) == a_set. intersection (b_set) ④ True >>> b_set. difference (a_set) == a_set. difference (b_set) ⑤ False

① The symmetric difference between a_set and b_set *does* not *look* like the symmetric difference between b_set and a_set , but remember, the sets are unordered. Any two sets, all (without exception) values of which are the same, are considered equal.

② This is exactly what happened here. Looking at the python shell's printable representation of these sets, don't be fooled. The values of the elements of these sets are the same, so they are equal.

③ The union of the two sets is also symmetric.

④ The intersection of the two sets is also symmetric.

⑤ The difference between the two sets is asymmetric. In essence, this operation is similar to subtracting one number from another. The order of the operands matters.

Finally, there are a few more set questions you can ask.

>>> a_set = { 1 , 2 , 3 } >>> b_set = { 1 , 2 , 3 , 4 } >>> a_set. issubset (b_set) ① True >>> b_set. issuperset (a_set) ② True >>> a_set. add (5) ③ >>> a_set. issubset (b_set) False >>> b_set. issuperset (a_set) False

① A plurality a_set a subset b_set - all elements a_set are also elements b_set .

② Conversely, b_set is a superset of a_set because all elements of a_set are also elements of b_set .

③ Since you added an item to a_set but did not add to b_set , both checks will return False .

Sets in a logical context

You can use sets in a logical context , such as in an if statement .

>>> def is_it_true (anything) : ... if anything: ... print (" yes , that's true ") ... else : ... print (" no , that's false ") ... >>> is_it_true (set ()) ① no , this is false >>> is_it _true ({ 'a' }) ② yes , this is true >>> is_it_true ({ False }) ③ yes , this is true

① In a logical context, an empty set is false.

② Any set containing at least one element is true.

③ Any set containing at least one element is true. The element values are not important.

Dictionaries

A dictionary is an unordered set of key-value pairs. When you add a key to a dictionary, you must also add a value for that key. (The value can always be changed later.) Dictionaries in Python are optimized to retrieve a value from a known key, but not for other purposes.

A dictionary in Python is similar to a hash in Perl 5 . In Perl 5, hash variables always start with a % character . In Python, variables can

be named whatever you want, the language itself keeps track of data types.

Dictionary creation

It is very easy to create a dictionary. The syntax is similar to the syntax for creating sets, but instead of elements, key-value pairs are used. If you have a dictionary, you can view the values by their key.

>>> a_dict = {'server': 'db.diveintopython3.org', 'database': 'mysql'} ① >>> a_dict {'server': 'db. diveintopython3.org ',' database ':' mysql '} >>> a_dict [' server '] ② ' db.diveintopython3.org ' >>> a_dict [' database '] ③ ' mysql ' >>> a_dict [' db .diveintopython3.o rg '] ④ Traceback (most recent call last): File "<stdin>", line 1, in <module> KeyError:' db.diveintopython3.org '

Translating the shell message:

Unrolling the stack (list of recent calls):
File "<stdin>", line 1, <module>

KeyError: 'db.diveintopython3.org'

① First, you create a new dictionary with two elements and assign it to the variable a_dict . Each element is a key-value pair, and the entire set of elements is enclosed in curly braces.

② 'server' is a key and it is associated with a value that a_dict ['server'] will access to give us 'db.diveintopython3.org' .

③ 'database' is a key and it is associated with a value that a_dict ['database'] will access to give us 'mysql' .

You can get a value by key, but you cannot get keys by value. So

④ a_dict ['server'] is 'db.diveintopython3.org' , but a_dict ['diveintopython3.org'] will throw an exception because 'db.diveintopython3.org' is not a key.

Changing the dictionary

Dictionaries do not have any predefined size limit. At any time, you can add new key-value pairs to the dictionary, or change the value corresponding to an existing key. Let's continue with the previous example:

>>> a_dict
{ 'server' : 'db.diveintopython3.org' , 'data base' : 'mysql' } >>> a_dict ['database'] = 'blog' ① >>> a_dict { 'server' : ' db.diveintopython3.org ' , ' database ' : ' blog ' } >>> a_dict [' user '] = ' mark ' ② >>> a_dict ③ { ' server ' : ' db.diveintopython3.org ' , ' user ' : 'mark' , 'database' : 'b log' } >>> a_dict ['user'] = 'dora' ④ >>> a_dict { 'server' : 'db.diveintopython3.org' , 'user' : ' dora ' , ' database ' : ' blog ' } >>> a_dict [' User '] = ' mark ' ⑤ >>> a_dict { ' User ' : ' mark ' , ' server ' : ' db.diveintopython3.org ' , 'user' : 'dora' , 'database' : 'blo g' }

① Your dictionary cannot contain the same keys. Assigning a value to an existing key will destroy the old value.

② You can add new key-value pairs at any time. This syntax is identical to the syntax for modifying existing values.

③ It seems that the new dictionary item (key 'user' , value 'mark') has hit the middle. In fact, it's just a coincidence that the items appear to be in

order in the first example; the same coincidence that they now appear out of order.

④ Assigning a value to an existing key simply replaces the old value with the new one.

⑤ Will the value for the 'user' key change back to "mark" ? Not! Look at it more closely - the "User" key is capitalized. Dictionary keys are case-sensitive, so this expression will create a new key-value pair rather than overwrite the existing one. It seems to you that the keys are similar, but from the point of view of Python they are completely different.

Dictionaries with mixed meanings

Dictionaries can be more than just strings. Dictionary values can be of any type, including integers, logical objects, arbitrary objects, or even other dictionaries. And the values in the same dictionary do not have to be of the same type; you can mix and match them as you need. Dictionary keys are more limited, but they can be strings, integers, and some other types. Keys of different types can also be mixed and combined in one dictionary.

In fact, you've already seen a dictionary with non-string keys and values in your first Python program.

SUFFI XES = { 1000 : ['KB' , 'MB' , 'GB' , 'TB' , 'PB' , 'EB' , 'ZB' , 'YB'] , 1024 : ['KiB' , 'MiB' , 'GiB' , 'TiB' , 'PiB' , 'EiB' , 'ZiB' , 'YiB']}

Let's pull this variable out of our program and work with it in an interactive Python shell.

>>> SUFFIXES = { 1000 : ['KB' , 'MB' , 'GB' , 'TB' , 'PB' , 'EB' , 'ZB' , 'YB'] , ... 1024 : ['KiB ' , ' MiB ' , ' GiB ' , ' TiB ' , ' PiB ' , ' EiB ' , ' ZiB ' , ' YiB ']} >>> len (SUFFIXES) ① 2 >>> 1000 in SUFFIXES ② True >> > SUFFIXES [1000] ③ ['KB' , 'MB' , 'GB' , 'TB' , 'PB' , 'EB' , 'ZB' , 'YB'] >>> SUFFIXES [1024] ④ ['KiB ' , ' MiB ' , ' GiB ' , ' TiB ' , ' PiB ' , ' EiB ' , ' ZiB ' , ' YiB '] >>> SUFFIXES [1000] [3] ⑤ ' TB '

① Just like for lists and sets , len () returns the number of elements in a dictionary.

② And just like with lists and sets, you can use the in operator to check if a particular key is defined in the dictionary.

③ 1000 *is the* key in the SUFFIXES dictionary ; its value is a list of eight elements (eight lines to be precise).

④ Similarly, 1024 is the key of the SUFFIXES dictionary ; and its value is also a list of eight elements.

⑤ Since SUFFIXES [1000] is a list, you can refer to the individual elements of the list by their ordinal numbers, which are indexed from zero.

Dictionaries in a logical context

Empty dictionaries are false, all others are true.

You can use dictionaries in a logical context such as an if statement .

>>> def is_it_true (anything) : ... if anything: ... print (" yes , that's true ") ... else : ... print ("no, that's false") ... >>> is_it_true ({}) ① no , this is a lie >>> is_it_true ({ 'a' : 1 }) ② yes , this is true

① In a logical context, an empty dictionary is false.

② Any dictionary with at least one key-value pair is true.

Constant None

None is a special constant in Python. It denotes an empty value. None is not the same as False . None is also not 0. None is not even an empty string. If you compare None with other data types, the result will always be False .

None is just empty. None has its own type (NoneType). You can assign None to any variable, but you cannot create other objects of type NoneType . All variables whose value is None are equal to each other .

>>> type (None) < class 'NoneType' > >>> None == False False >>> None == 0 False >>> None == '' False >>> None == None True >>> x = None >>> x == None True >>> y = None >>> x == y True

None in a boolean context

In a logical context, None is always false and not None is true.

>>> def is_it_true (anything) : ... if anything: ... print (" yes , that's true ")

... else : ... print (" no , that's false ") ... >>> is_it_true (None) no , this is false >>> is_it_true (not None) yes , this is true

Further reading

- Logical operations
  - Numeric types
  - Sequence types
    - Set types
  - Display types
  - Fractions module
  - Math module
- PEP 237: Unifying Long Wholes and Wholes
  - PEP 238: Modifying the Division Operator

Generators

We have to strain our imagination more strongly, not in order, as in fiction, to imagine what does not exist in reality, but in order to comprehend what is really happening.

Richard Feynman

Immersion

Each programming language has one such feature, a complex, but specially simplified thing. If you've written in a different language before, you may not want to pay attention to this, because your old language didn't simplify this thing that much (because it was busy simplifying something else a lot). In this chapter, you will explore list, dictionary, and set generators - three related concepts centered around one very powerful technology. But first, I want to deviate a little from our story to tell you about two modules that will help you navigate your local file system.

Working with files and directories

Python 3 comes with the os module , which stands for operating system. The os module contains many functions for obtaining information about (and in some cases, manipulating) local directories, files, processes and environment variables. Python offers a very good unified programming interface for all supported operating systems, so your programs can run on any computer with a minimum of platform-specific code.

Current working directory

When you're just getting started with Python, you spend a lot of time in the interactive Python shell. Throughout this book, you will see examples that look like this:

1. Importing any module from the examples folder
2. Calling a function from this module
3. Explanation of the result

There is always a current working directory.

If you don't know anything about the current working directory, then step 1 may fail and an ImportError will be thrown . Why? Because Python will look for the specified module in the search path of the import statement , but it won't find it because the `examples` directory is not in the search paths. To fix this, you can do one of two things:

- either add the `examples` folder to the search path of the import statement ;
- or make the `examples` folder the current working directory .

The current working directory is an implicit parameter that Python stores permanently in memory. The current working directory is always there when you are working in an interactive Python shell, running your scripts from the command line, or a CGI script somewhere on a web server .

The os module contains two functions for working with the current working directory.

>>> import os [1]

>>> print (os . getcwd ()) C: \ Python31 [2]

>>> os . chdir ('/ Users / pilgrim / diveintopython3 / examples') [3]

>>> print (os . getcwd ()) C: \ Users \ pilgrim \ diveintopython3 \ examples [4]

1. ↑ The os module comes with Python; you can import it anytime, anywhere.
2. ↑ Use the os function . getcwd () to get the value of the current working directory. When you are using the Python graphical shell, the current working directory is the directory from which it was started. On Windows, it depends on where you installed Python; the default directory is `c: \ Python31` . If the Python shell is started from the command line, the current working directory is the one you were in when you started it.
3. ↑ Use the os function . chdir () to change the current working directory.
4. ↑ When I called os . chdir () , I used a Linux -style path (forward slash , no drive letter) even though it actually worked on Windows . This is one of those places where Python tries to blur the differences between operating systems.

Working with file and directory names

Speaking of directories, I would like to draw your attention to the os module . path . It contains functions for working with file and directory names.

>>> import os

>>> print (os . path . join ('/ Users / pilgrim / diveintopython3 / examples /' [1] , 'humansize.py')) / Users / pilgrim / diveintopython3 / examples /

humansize. py

>>> print (os . path . join ('/ Users / pilgrim / diveintopython3 / examples' , 'humansize.py')) / Users / pilgrim / diveintopython3 / examples \ humansize. py [2]

>>> print (os . path . expanduser ('~')) c: \ Users \ pilgrim [3]

>>> print (os . path . join (os . path . expanduser ('~') , 'diveintopython3' , 'examples' , 'humansize.py')) c: \ Users \ pilgrim \ diveintopython3 \ examples \ humansize. py [4]

1. ↑ Function os . path . join () composes a directory path from one or more partial paths. In this case, it just concatenates the strings.
2. ↑ This is already a less trivial case. The join function will add an extra slash to the folder name before appending the file name. In this case, Python adds a backslash instead of a backslash, because I ran this example on Windows . If you enter this command on Linux or Mac OS X , you will see a simple forward slash. Python can refer to a file regardless of what delimiter is used in the file path.
3. ↑ Function os . path . expanduser () expands the path that uses the `~` character to indicate the home directory of the current user. The feature works on any platform where the user has a home directory, including Linux, Mac OS X, and Windows. The function returns the path without the trailing slash, but for the os . path . join () it doesn't matter.
4. ↑ By combining the two, you can easily build file paths for folders and files in the user's home directory. Function os . path . join () takes any number of arguments. I really enjoyed discovering this because in other languages, when developing tools, I had to constantly write a silly little addSlashIfNecessary () function . In the Python programming language, smart people have

already taken care of this.

The os module . path also contains functions for splitting file paths, folder and file names into their constituent parts.

>>> pathname = '/ Users / pilgrim / diveintopython3 / ex amples / humansize.py'

>>> os . path . split (pathname) ('/ Users / pilgrim / diveintopython3 / examples' , 'humansize.py') [1]

>>> (dirname , filename) = os . path . split (pathname) [2]

>>> dirname [3]
'/ Users / pilgrim / diveintopython3 / examples'

>>> filename [4]
'humansize.py'

>>> (shortname , extension) = os . path . splitext (filename) >>> shortname 'humansize' >>> extension '.py' [five]

1. ↑ The split function splits the full path and returns a tuple containing the directory path and file name separately.
2. ↑ Remember, I talked about how to assign several values at once and how to return several values from a function at the same time? Function os . path . split () does exactly that. You can assign the return value from the split function to a tuple of two variables. Each of the variables will take on the value of the corresponding element of the resulting tuple.
3. ↑ The first variable , dirname , will receive the value of the first element of the tuple returned by the os function . path . split () , namely the path to the directory.
4. ↑ The second variable , filename , will take the value of the second element of the tuple returned by the os function . path . split () , namely the filename.

5. ↑ The os module . path also contains the os function . path . splitext () , which splits the filename and returns a tuple containing the filename and filename separately. You can use the same technique as before to assign each of the values of interest to separate variables.

Retrieving directory contents

The glob module understands the wildcard characters used in shells.

The glob module is another tool from the Python standard library. This is an easy way to programmatically retrieve the contents of a folder, and it also knows how to use wildcards , which you are probably familiar with if you worked on the command line.

>>> os . chdir ('/ Users / pilgrim / diveintopython3 /') >>> import glob

>>> glob . glob ('examples / *. xml') ['examples \\ feed-broke n.xml' , 'examples \\ feed-ns0.xml' , 'examples \\ feed.xml'] [1]

>>> os . chdir ('examples /') [2]

>>> glob . glob ('* test * .py') ['alphameticstest.py' , 'pluraltest1.py' , 'pluraltest2.py' , 'pluraltest3.py' , 'pluraltest4.py' , 'pluraltest5.py' , 'pluraltest6. py ' , ' romantest1.py ' , ' romantest10.py ' , ' romantest2.py ' , ' romantest3.py ' , ' romantest4.py ' , ' rom antest5.py ' , ' romantest6.py ' , ' romantest7.py ' , ' romantest8.py ' , ' romantest9.py '] [3]

1. ↑ The glob module takes a wildcard pattern and returns the paths of all files and directories that match it. In this example, the template contains a directory path and `"* .xml"` that will match all `xml` files in the `examples` directory .
2. ↑ Now let's make the current working directory `examples` . Function os . chdir () can accept relative paths as well.
3. ↑ You can use multiple jokers in your template. This example finds all files in the current working directory that end in `.py` and contain the word `test` anywhere in the filename.

Retrieving file information

Any modern operating system stores information about each file (metadata): creation date, last modified date, file size, and so on. Python provides a single programming interface for accessing this metadata. You don't need to open the file; all that is required is a filename.

>>> import os

>>> print (os . getcwd ()) c: \ Users \ pilgrim \ diveintopython3 \ examples [1]

>>> metadata = os . stat ('feed.xml') [2]

>>> metadata. st_mtime [3]
1247520344.9537716

>>> import time [4]

>>> time . localtime (metadata. st_mtime) time . struct_time (tm_year = [five]
2009 , tm_mon = 7 , tm_mday = 13 , tm_hour = 17 , tm_min = 25 , tm_sec

= 44 , tm_wday = 0 , tm_yday = 194 , tm_isdst = 1)

1. ↑ Current working directory - folder with examples.
2. ↑ feed.xml - file in the folder with examples. The os . stat () returns an object containing various metadata about the file.
3. ↑ st_mtime is the file modification time, but it is written in a terribly inconvenient format. (This is actually the number of seconds since the beginning of the " UNIX era ", which began in the first second on January 1 , 1970. Seriously.)
4. ↑ The time module is part of the Python standard library. It contains functions for converting between different time formats and time zones, for converting them to strings (str), etc.
5. ↑ The time function . localtime () converts the time of the format "seconds from the beginning of the era" (field st_mtime , returned by os . the stat ()) into a more convenient structure containing the year, month, day, hour, minute, second, and so on. d. This file last modified on July 13, 2009, at approximately 17 hours, 25 minutes.

```
# continuation of the previous example
>>> metadata. st_size                                    [1]
3070
>>> import humansize
>>> humansize. approximate_size ( metadata. st_size )    [2]
'3.0 KiB'
```

1. ↑ Function os . stat () also returns the file size in the st_size property . The size of the feed.xml file is 3070 bytes.
2. ↑ You can pass the st_size property to the approximate_size () function .

Getting absolute paths

In the previous section, the glob . glob () returned a list of relative paths. In the first example, the paths were of the form 'examples \ feed.xml' , and in the second, the relative paths were even shorter, for example, 'romantest1.py' . As long as you stay in the current working directory, you can use these relative

paths to open files or get their metadata. But if you want an absolute path - that is, one that includes all directory names up to the root or up to a drive letter, you need the os . path . realpath () .

>>> import os >>> print (os . getcwd ()) c: \ Users \ pilgrim \ diveintopython3 \ examples >>> print (os . path . realpath ('feed.xml'))

c: \ Users \ pilgrim \ diveintopython3 \ examples \ feed.xml

List generators

In any Python expressions can use generators lists.

With list generators, you can easily map one list to another by applying some function to each item.

>>> a_list = [1 , 9 , 8 , 4] >>> [elem * 2 for elem in a_list] [2 , 18 , 16 , 8] [1]

>>> a_list [2]
[1 , 9 , 8 , 4]

>>> a_list = [elem * 2 for elem in a_list] >>> a_list [2 , 18 , 16 , 8] [3]

1. ↑ To understand what's going on here, read the generator from right to left. a_list is the list to display. Python loops through the elements of a_list sequentially , temporarily assigning the value of each element to the variable elem . Then it uses the elem * 2 function and adds the result to the returned list.
2. ↑ The generator creates a new list without changing the original one.
3. ↑ You can assign the result of the list generator to a displayed variable. Python will create a new list in memory and, when the generator has output, assign it to the original variable.

The list comprehensions can use any Python expression, including the

functions of the module os , used to work with files and directories.

>>> import os , glob >>> glob . glob ('* .xml') ['feed-broken.xml' , 'feed-ns0.xml' , 'feed.xml'] [1]

>>> [os . path . realpath (f) for f in glob . glob ('* .xml')] ['c: \\ Users \\ pilgrim \\ diveintopython3 \\ examples \\ feed-broken.xml' , 'c: \\ Users \\ pilgrim \\ diveintopython3 \\ examples \ \ feed-ns0.xml ' , ' c: \\ Users \\ pilgrim \\ diveintopython3 \\ examples \\ feed.xml '] [2]

1. ↑ This expression returns a list of all .xml files in the current working directory.
2. ↑ This generator takes a list of all .xml files and converts it to a list of full paths.

When generating lists, you can also filter items to discard some of the values.

>>> import os , glob

>>> [f for f in glob . glob ('* .py') if os . stat (f) . st_size > 6000] ['plural test6.py' , 'romantest10.py' , 'romantest6.py' , 'romantest7.py' , 'romantest8.py' , 'romantest9.py'] [1]

1. ↑ To filter the list, add an if statement at the end of the list generator. The expression after the if statement will be evaluated for each item in the list. If this expression is true, this element will be processed and included in the generated list. In this line generates a list of all .py-files in the current directory, and the operator if filter this list, leaving only files larger than 6000 bytes.

There are only six such files, so a list of six file names will be generated .

All the examples of list generators discussed above used simple expressions: multiplying a number by a constant, calling one function, or simply returning a list item unchanged (after filtering). But when generating lists, you can use expressions of any complexity.

>>> import os , glob

>>> [(os . stat (f) . st_size , os . path . realpath (f)) for f in glob . glob ('* [1] .xml')] [(3074 , 'c: \\ Users \\ pilgrim \\ diveintopython3 \\ examples \\ feed-broken.xml') , (3386 , 'c: \\ Users \\ pilgrim \\ diveintopython3 \\ examples \\ feed-ns0.xml ') , (3070 , ' c: \\ Users \\ pilgrim \\ diveintopython3 \\ examples \\ feed.xml ')] >>> import humansize

>>> [(humansize. approximate_size (os . stat (f) . st_size) , f) for f in [2] glob . glob ('* .xml')] [('3.0 KiB' , 'feed-broken.xml') , ('3.3 KiB' , 'feed-ns0.xml') , ('3.0 KiB' , 'feed.xml ')]

1. ↑ This generator looks for all .xml files in the current working directory, gets the size of each file (by calling os . Stat ()), and creates a tuple from the file size and absolute path of each file (by calling os . Path . Realpath ()) ...
2. ↑ This generator, based on the previous one, calls the approximate_size () function passing in the size of each .xml file.

Dictionary generators

A dictionary generator is like a list generator, but instead of a list, it creates a dictionary.

>>> import os , glob

>>> metadata = [(f , os . stat (f)) for f in glob . glob ('* test * .py')] [1]

>>> metadata [0] [2]
('alphameticstest.py' , nt. Stat_result (st_mode = 33206 , st_ino = 0 , st_dev = 0 , st_nlink = 0 , st_uid = 0 , st_gid = 0 , st_size = 2509 , st_atime = 1247520344 , st_mtime = 1247520344 , st_ctime = 1247520344))

>>> metadata_dict = { f: os . stat (f) for f in glob . glob ('* test * .py')} [3]

>>> type (metadata_dict) < class 'dict' > [4]

>>> list (metadata_dict. keys ()) ['romantest8.py' , 'pluraltest1.py' , 'pluraltest2.py' , 'pluraltest5.py' , 'pluraltest6.py' , 'romantest7.py' , 'romantest10 .py ' , ' romantest4.py ' , ' romantest9.py ' , ' pluraltest3.py ' , ' romantest1.py ' , ' romantest2.py ' , ' romantest3.py ' , ' romantest5.py ' , ' romantest6.py ' , ' alphameticstest.py ' , ' pluraltest4.py '] [five]

>>> metadata_dict ['alphameticstest.py'] . st_size [6]
2509

1. ↑ This is not a dictionary generator, it is a list generator. It finds all files with a .py extension, checks their names, and then creates a tuple from the file name and file metadata (by calling the os . Stat () function).
2. ↑ Each element of the resulting list is a tuple.
3. ↑ This is a dictionary generator. The syntax is similar to the list generator syntax, with two differences. Firstly, it is enclosed in braces, and not in the square. Second, instead of one expression for each element, it contains two, separated by a colon. The expression to the left of the colon (f in our example) is the dictionary key; the expression to the right of the colon (os . stat (f) in our example) is the value.
4. ↑ Dictionary generator returns a dictionary.

5. ↑ keys of the dictionary - it's just the file names obtained using glob . glob ('* test * .py') .
6. ↑ The value associated with each key is obtained using the os function . stat () . This means that in this dictionary we can get its metadata by the file name. One of the metadata elements (st_size) is the file size. The alphameticstest.py file is 2509 bytes in size .

As well as in generators lists, you can include generators dictionaries condition of the if , to filter the input sequence with the expression-conditions, is calculated for each element.

```
>>> import os , glob , humansize
>>> metadata_dict = { f: os . stat ( f ) for f in glob . glob ( '*' )}
>>> humansize_dict = { os . path . splitext ( f ) [ 0 ] : humansize.
approximate_size ( meta. st_size ) \ ... for f , meta in metadata_dict. items ()
if meta. st_size > 6000 }
>>> list ( humansize_dict. keys ()) [ 'romantest9' , 'romantest8' ,
'romantest7' , 'romantest6' , 'romantest10' , 'pluraltest6' ]
>>> humansize_dict [ 'romantest9' ]
'6.5 KiB'
```

1. ↑ In this expression is taken a list of files in the current directory (glob . Glob ('*')), for each file are defined by its metadata (os . The stat (f)) and built a dictionary whose keys are the names of the files, and values - the metadata of each file.
2. ↑ This generator builds on the previous one. Filters out files smaller than 6000 bytes (the if the meta. St_size > 6000). Selected elements are used to build a dictionary, the keys of which are file names without extension (os . Path . Splitext (f) [0]), and the values - the approximate size of each file (humansize. Approximate_size (meta. St_size)).
3. ↑ As you already know from the previous example, there are six such files in total, therefore there are six elements in this

dictionary.

4. ↑ The value for each key is a string obtained by calling the `approximate_size()` function .

Other cool things you can do with dictionary generators

Here's a trick with dictionary generators you might find useful someday: swapping the keys and values of a dictionary.

>>> a_dict = { 'a' : 1 , 'b' : 2 , 'c' : 3 } >>> { value: key for key , value in a_dict. items ()} { 1 : 'a' , 2 : 'b' , 3 : 'c' }

Of course , this will only work if the values of the dictionary elements are immutable, such as strings or tuples.

>>> a_dict = { 'a' : [1 , 2 , 3] , 'b' : 4 , 'c' : 5 } >>> { value: key for key , value in a_dict. items ()} Traceback (most recent call last) : File "<stdin>" , line 1 , in < module > File "<stdin>" , line 1 , in < dictcomp > TypeError : unhashable type : 'list'

Set generators

It can not be left to the board and the sets they can also be created with the help of generators. The only difference is that instead of key: value pairs, they are based on the same values.

>>> a_set = set (range (10)) >>> a_set { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 }

>>> { x ** 2 for x in a_set } { 0 , 1 , 4 , 81 , 64 , 9 , 16 , 49 , 25 , 36 } [1]

>>> { x for x in a_set if x % 2 == 0 } { 0 , 8 , 2 , 4 , 6 } [2]

>>> { 2 ** x for x in range (10)} { 32 , 1 , 2 , 4 , 8 , 64 , 128 , 256 , 16 , [3]

512 }

1. ↑ In an input generator sets can receive other sets. This generator calculates the squares of a set of numbers ranging from 0 to 9.
2. ↑ Like list and dictionary generators, set generators can contain an if condition to test each item before including it in the result set.
3. ↑ On entry generator sets can receive not only a plurality, but and any other sequences.

Further reading

- Os module
- os - access to special features of operating systems
- Os.path module
- os.path - platform independent file name manipulation
- Glob module
- glob - compare filenames with patterns
- Time module
- time - Functions for manipulating time
- List generators
- Nested List Generators
- Cycle technique

Strings

A few boring things you need to know before diving

Did you know that the people of Bougainville have the shortest alphabet in the world? The Rotokas alphabet consists of only 12 letters: A, E, G, I, K, O, P, R, S, T, U, and V. At the other end of this peculiar number axis there are languages such as Chinese, Japanese and Korean with thousands of

characters. English, of course, contains only 26 letters - 52 if you count both upper and lower case letters - plus a handful of punctuation marks ! @ # $% &.

When people say text, they mean letters and symbols on a computer screen. But computers don't work with letters and symbols; they work with bits and bytes. Every piece of text that you've ever seen on a computer screen is actually stored in a specific encoding. Roughly speaking, character encoding ensures that what you see on the screen matches what is actually stored in memory or on disk. There are many different character encodings, some of them are optimized for specific languages, for example Russian, Chinese or English, others can be used for several languages at once.

In reality, everything is much more complicated. Many characters are common to several encodings, but each encoding can use a different sequence of bytes to store them in memory or on disk. You can think of character encoding as a kind of cryptographic key. Whenever you are given a sequence of bytes - file, web page, whatever - and claimed to be "text", you need to understand what encoding was used. Knowing the encoding, you can decode the bytes to characters. If you are given the wrong key or no key at all, you have no choice but to try to crack the code yourself. Most likely, as a result, you will get a bunch of kryakozyabrov (gibberish - gibberish, slurred speech, but it's clearer, approx. Transl.). Everything you knew about strings is wrong.

Everything you knew about strings is wrong.

Surely you've seen webpages like this, with strange question marks in place of apostrophes. This usually means that the author of the page incorrectly specified their encoding, your browser just needs to guess it, and the result is a mixture of expected and completely unexpected characters. In English, this is just annoying; in other languages, the result may become completely unreadable.

There are encodings for all major world languages. But since languages differ significantly from each other, and memory and disk space used to be expensive, each encoding is optimized for a specific language. By this I mean

that to represent the characters of their language, all encodings use the same range of numbers (0-255). For example, you are probably familiar with the ASCII encoding, which stores English characters as numbers from 0 to 127. (65 is an uppercase "A", 97 is a lowercase "a", etc.) The English alphabet is very simple and can be represented less than 128 numbers. If you know the binary number system, you understand that only 7 out of 8 bits are involved in a byte.

Western European languages such as French, Spanish and German contain more characters than English. More precisely, they contain characters with different diacritics, such as the Spanish character ñ. The most common encoding for these languages is CP-1252, also known as “windows-1252” due to its widespread use in the Microsoft Windows operating system. In CP-1252, characters 0 through 127 are the same as in ASCII, and the rest of the range is used for characters such as n-with-tilde-top (241), u-with-two-dots-top (252) etc. However, this is still a single-byte encoding; the maximum possible number 255 still fits in one byte.

And then there are languages like Chinese, Japanese, and Korean that have so many characters that they require multibyte encodings. This means that each “character” is represented by a two-byte number between 0 and 65535. But different multibyte encodings still have the same problem as different single-byte encodings: each encoding uses the same numbers to mean different things. The only difference is that the range of numbers is larger, because there are many more characters to encode.

This was quite normal in the offline world where you would type "text" for yourself and sometimes print it out. In this world there was nothing but “plain text” (I don’t know, there might be no translation of “plain text” at all - approx. Transl.). The source code was ASCII, and for everything else, word processors were used, which defined their own (non-text) formats in which, along with formatting information, encoding information was tracked. People read these documents with the same word processor that was used to create them, so everything more or less worked.

Now think about the proliferation of global networks such as e-mail and the web. A lot of "plain text" is moved around the planet, created on one computer, transmitted through a second, and received and displayed by a third computer. Computers only see numbers, but numbers can have different meanings. Oh no! What to do? The systems were designed to convey the

encoding information along with each piece of "plain text". Remember, this is a cryptographic key that establishes a correspondence between numbers and human-readable symbols. A lost key means garbled text or krakozyabry, if not worse.

Now think about the task of storing different snippets of text in one place, for example, in one database table that stores all the email messages you ever receive. You still need to store the encoding along with each piece of text to be able to read it. Do you think this is difficult? Try to search this database, converting on the fly between multiple encodings. Isn't it funny?

Now think about the possibility of multilingual documents, where characters from multiple languages are side by side in the same document. (Hint: programs that try to do this usually use alphabet change codes to switch “modes.” Wham, and you're in Russian KOI8-R mode , and 241 means I; bam, and now you're in Greek mode for Macintosh, and 241 means ώ.) And of course you also want to search these documents.

Now cry, because everything you know about strings is wrong, and there is no such thing as plain text.

Unicode

Introduction to Unicode.

Unicode is designed for the system to represent every character in any language. Unicode represents each letter, character, or ideography as a 4-byte number, each number representing a unique character used in at least one of the world's languages. (more than 65535 of them are used, so 2 bytes would not be sufficient.) Characters that are used in different languages have the same code unless there is a good etymological reason. Regardless of everything, there is exactly 1 code corresponding to a character, and exactly 1 character corresponding to a numeric code. Each code always means only one character; there are no "modes". U + 0041 always matches 'A', even if your language does not have an 'A'.

At first glance, this is a great idea. One encoding for everything. Many languages in one document. No more "mode switching" is needed to change encodings. But you should have an obvious question. Four bytes? For each character? This seems terribly wasteful, especially for languages like English or Spanish, which need less than one byte (256 numbers) to represent any possible character. In fact, this is wasteful even for hieroglyphic languages

(such as Chinese), which never need more than two bytes per character.

There is a Unicode encoding that uses four bytes per character. It is called UTF-32 because 32 bits = 4 bytes. UTF-32 - straightforward encoding; each Unicode character (4-byte number) corresponds to a character with a specific number. This has its advantages, the most important of which is that you can find the Nth character in a string in constant time, since the Nth character starts at 4 * N bytes. But this encoding also has disadvantages, the most obvious of which is that it takes four bytes to store each character.

Although there are a huge number of characters in Unicode, in reality most people never use those with numbers higher than 65535 (2 ^ 16). So there is another Unicode encoding called UTF-16 (obviously 16 bits = 2 bytes). UTF-16 encodes each character in numbers 0–65535; to represent rarely used "outrageous" symbols with numbers above 65535, you have to resort to some tricks. The most obvious advantage: UTF-16 is twice as efficient in memory consumption than UTF-32, since each character requires 2 bytes instead of 4 (except for the cases with those "beyond" characters). And, as in the case of UTF-32, you can easily find the required N-th character in a string in constant time, if you are sure that the text does not contain "out-of-bounds" characters; and everything is fine if it really is.

However, there are subtle disadvantages to both UTF-32 and UTF-16. Individual bytes are stored differently on different computer platforms. This means that the character U + 4E2D can be stored in UTF-16 as either 4E 2D or 2D 4E (backwards), depending on the byte order used: big-endian or little-engian. (For UTF-32 there are even more kinds of orders.) As long as you keep your documents exclusively on your computer, you are safe - different applications on the same computer always use the same order. But the moment you want to transfer documents to another computer on the Internet or another network, you need a way to mark the document in which byte order you are using. Otherwise, the receiving system has no idea what the "4E 2D" byte sequence represents: U + 4E2D or U + 2D4E.

To solve this problem, Unicode multibyte encodings have a “byte order mark” (BOM), which is a special non-printable character that you can include at the beginning of a document to preserve information about the byte order being used. For UTF-16, this mark is numbered U + FEFF. If you receive a UTF-16 document that begins with the FF FE byte, this is a forward-order message uniquely; if it starts with FE FF bytes, then the order is reversed.

In fact, UTF-16 is not ideal, especially if you are dealing with a lot of ASCII characters. You didn't think that even a Chinese web page can contain a large number of ASCII characters - all the elements and attributes that surround printable Chinese characters (and they also spend 2 bytes, although they fit into one). The ability to search for the Nth character in a string in constant time is tempting, but there is still a problem with those "outlandish" characters, which is annoying for everyone, which is that you cannot guarantee that each character is stored in exactly two bytes, as a result of which searching in constant time also becomes impossible (unless you have a separate character index). Let me tell you a secret: there are still a huge number of ASCII texts in the world ...

Someone before you also thought about this problem and came to this solution:

UTF-8

UTF-8 is a variable byte Unicode encoding. This means that different characters take up different number of bytes. For ASCII characters (AZ, numbers, etc.), UTF-8 uses only 1 byte per character (really, and no longer required). Moreover, in fact, exactly the same numbers are reserved for them as in ASCII; the first 128 characters (0–127) of the UTF-8 table are indistinguishable from the same ASCII part. “Extended” characters such as ñ and ö are two bytes. (bytes are not simply the Unicode code point like they would be in UTF-16; there is some serious bit-twiddling involved.) Chinese characters such as 中 are three bytes. The most rarely used symbols are four.

Disadvantages: Since each character takes up a different number of bytes, searching for the Nth character is O (N), which means that the search time is proportional to the length of the string. In addition, bit-twiddling, which is used to encode characters into bytes, also increases the search time. (approx. transl. in encoding with a fixed number of bytes per character, the search time is O (1), that is, it does not depend on the length of the string).

Benefits: Extremely efficient encoding of the most commonly used ASCII characters. No worse than storing wide characters in UTF-16. Better than UTF-32 for Chinese characters. Also (I don't want to burden you with math, so you have to take my word for it), due to the very nature of bit twiddling, there is simply no problem with byte ordering. A document encoded in UTF-8 uses the same byte order on any computer!

Immersion

In the Python 3 programming language, all strings are a sequence of Unicode characters. In Python, there is no such thing as a UTF-8 encoded string or a CP-1251 encoded string. The question is incorrect: "Is this a string in UTF-8?" UTF-8 is a way to encode characters into a sequence of bytes. If you want to take a string and turn it into a sequence of bytes in some encoding, then Python 3 can help you with that. If you want to turn a sequence of bytes into a string, then Python 3 will come in handy here. Bytes are not characters, bytes are bytes. Symbols are an abstraction. And a string is a sequence of such abstractions.

>>> s = ' 深入 Python' ① >>> len (s) ② 9 >>> s [0] ③ ' 深 ' >>> s + '3' ④ ' 深入 Python 3'

- ① To create a string, surround it with quotes. The Python lines can be created with either single (`'`) and double quotes (`"`).
- ② The standard function len () returns the length of a string, i.e. the number of characters in it. The same function is used to determine the length of lists, tuples, sets and dictionaries. In this case, a string is like a tuple of characters.
- ③ Just like with lists, you can get an arbitrary character from a string, knowing its index.
- ④ As with lists, you can concatenate strings using the `+` operator .

Formatting strings

Strings can be created using either single or double quotes.

Let's take another look at `humansize.py` :

SUFFIXES = { 1000 : ['KB' , 'MB' , 'GB' , 'TB' , 'PB' , 'EB' , 'ZB' , 'YB'] , ① 1024 : ['KiB' , 'MiB' , 'GiB' , 'TiB' , 'PiB' , 'EiB' , 'ZiB' , 'YiB']} def approximate_size (si ze , a_kilobyte_is_1024_bytes = True) : " 'Convert a

file size to human-readable form. ② Keyword arguments: size - file size in bytes a_kilobyte_is_1024_bytes - if True (default), use multiples of 1024 if False, use multiples of 1000 Returns: string " ' ③ if size < 0 : raise ValueError (' number must be non-negative ') ④ multip le = 1024 if a_kilobyte_is_1024_bytes else 1000 for suffix in SUFFIXES [multiple] : size / = multiple if size < multiple: return ' {0: .1f} {1} ' . format (size , suffix) ⑤ raise ValueError ('number too large ')

- ① 'KB', 'MB', 'GB' ... are all lines.
- ② Function comments are also a string. Function comments can be multi-line, so triple quotes are used at the beginning and end of the line.
- ③ These triple quotes end function comments.

- ④ Here's another line that is passed to the exception constructor as human readable error text.
- ⑤ Here ... wow, what is this?

Python 3 supports formatting values to strings. Formatting can involve very complex expressions. The simplest use is to insert a value into a string substitution field.

>>> username = 'mark' >>> password = 'PapayaWhip' ① >>> "{0} 's password is {1}" . format (username , password) ② "mark's password is PapayaWhip"

- ① You don't think my password is really `PapayaWhip`
- ② There is a lot going on here. First, the format (...) method is called on the string. Strings are objects, and objects have methods. Second, the value of the entire expression will be a string. Third, `{0}` and `{1}` are fields that are replaced by arguments passed to the format () method

Composite field names

The previous example showed the simplest way to format strings: fields in a string are integers. These numbers in curly braces indicate the ordinal numbers of the arguments in the parameter list of the `format ()` method . This means that `{0}` is replaced with the first argument (in this case `username`), and `{1}` is replaced with the second argument (`password`), & c. You can have as many field numbers as there are arguments to the `format ()` method . And there can be any number of arguments. But field names are much more powerful than they might seem at first glance.

>>> import humansize >>> si_suffixes = humansize. SUFFIXES [1000] ① >>> si_suffixes ['KB' , 'MB' , 'GB' , 'TB' , 'PB' , 'EB' , 'ZB ' , 'YB'] >>> '1000 {0 [0]} = 1 {0 [1]} ' . format (si_suffixes) ② '1000KB = 1MB'

- ① Instead of calling any functions of the `humansize` module , you simply use one of the vocabularies defined in this module: list of SI suffixes (powers of 1000)
- ② This piece looks complicated, although it is not. `{0}` refers to the first argument passed to the `format ()` method (`si_suffixes` variable). But `si_suffixes` is a list. Therefore `{0 [0]}` refers to the first element of this list: 'KB' . At the same time `{0 [1]}` refers to the second element of the same list: 'MB' . Anything outside the curly braces — including 1000, the equal sign, and spaces — is left untouched. As a result, we get the line '1000KB = 1MB' .

{0} is replaced by the 1st format () argument. {1} is replaced by t he 2nd.

This example shows that when formatting in field names, you can access the elements and properties of data structures using (almost) Python syntax. This is called "compound field names". The following compound field names just work:

- Pass the list and access the list item by its index (as in the previous example);
- Pass the dictionary and access the dictionary value by its key;
- Pass a module and access its variables and functions knowing their names;
- Pass an instance of a class and access its properties and methods by their names;
- Any combination of the above.

And to blow your mind, here's an example that uses all of the above possibilities:

>>> import humansize >>> import sys >>> '1MB = 1000 {0.modu les [humansize] .SUFFIXES [1000] [0]}' . format (sys) '1MB = 1000KB'

This is how it works:

The `sys` module contains information about a running Python interpreter. Since you imported it, you can use it as an argument to the `format()` method . That is, field `{0}` refers to the `sys` module . sys . modules is a dictionary with all the modules that are currently imported by the Python interpreter. The keys of this dictionary are strings with module names; values are objects representing imported modules. Thus, the `{0.modules}` field refers to a dictionary of imported modules. sys . modules ['humansize'] is an object representing the `humansize` module that you just imported. Thus, the `{0.modules [humansize]}` composite field refers to the `humansize` module . Note that the syntax is different here. In Python syntax, the keys of the sys . modules are strings, and you need to quote the module name (eg 'humansize') to access the dictionary values . Here is a quote from PEP 3101: Advanced String Formatting: "The rules for parsing keys are very simple. If it starts with a digit, it must be interpreted as a number. Otherwise, it's a string." sys . modules ['humansize'] . SUFFIXES is a dictionary defined at the very beginning of the `humansize` module . The field `{0.modules [humansize] .SUFFIXES}` refers to this dictionary.

sys . modules ['humansize'] . SUFFIXES [1000] is a list of SI suffixes: ['KB' , 'MB' , 'GB' , 'TB' , 'PB' , 'EB' , 'ZB' , 'YB'] . Therefore, the field `{0.modules [humansize] .SUFFIXES [1000] }` refers to this list. And sys . modules ['humansize'] . SUFFIXES [1000] [0] is the first element of the suffix list: 'KB' . This `replaces the` final compound field `{0.modules [humansize] .SUFFIXES [1000] [0]}` with a two-character string KB.

Format specifiers

Wait! There is something else. Let's take a look at another weird line from `humansize.py` :

```
if size < multiple: return '{0: .1f} {1}' . format ( size , suffix )
```

`{1}` is replaced with the second argument of the format () method, that is, with the value of the `suffix` variable . But what does `{0: .1f} mean` ? There are two things here: `{0}` , which you already know and `: .1f` , which you haven't heard of yet. The second half (colon and everything after it) describes the format, which specifies how the placeholder value should be formatted.

☞ The format specifier allows you to modify placeholder text in many useful ways, like the `printf()` function in the C programming language. You

can add zero or space padding, horizontal text alignment, decimal precision control, and even hex conversion.

Within a field to be replaced, a colon (`:)` character and everything after it means a format specifier. The format specifier `".1"` means "round to tenths" (that is, show only one decimal place). The format specifier `"f"` means "fixed point number" (**f** ixed-point number) (in contrast to the exponential or any other representation of decimal numbers). Thus, if `size` is 698.24 and `suffix` is 'GB' , the formatted string will be '698.2 GB' , because 698.24 is rounded to one decimal place and suffix is added to it.

>>> '{0: .1f} {1}' . format (698.24 , 'GB') '698.2 GB'

For full details of format specifiers, see the "Format Specification Mini-Language" section of the official Python 3 documentation.

Other common string methods

In addition to formatting, strings allow you to do many useful tricks.

>>> s = " 'Finished files are the re- ① ... sult of years of scientif- ... ic study combined with the ... experience of years.' " >>> s. splitlines () ② ['Finished files are the re-' , 'sult of years of scientif-' , 'ic study combined with the' , 'experience of years.'] >>> print (s. lower ()) ③ finished files are the re- s ult of years of scientific study combined with the experience of years. >>> s. lower () . count ('f') ④ 6

- ① In the Python interactive shell, you can enter multi-line text. Such text begins with a triple quotation mark. And when you press ENTER, the interactive shell will prompt you to continue entering text. Multiline text must also end with triple quotation marks. When you press ENTER, the Python interactive shell will execute the command (write the text to the `s` variable).
- ② The splitlines () method takes multiline text and returns a list of lines, one for each line of the original text. Note that line feeds are not added to the resulting strings.
- ③ The lower () method converts all characters in the string to lowercase. (Similarly, the upper () method converts a string to uppercase.)
- ④ The count () method counts the number of occurrences of a substring. Yes, this sentence has 6 letters "f".

Here's another common case. Suppose you have a list of key-value pairs in the form `key1 = val ue1 & key2 = value2` , and you want to separate them and get a dictionary in the form { key1: value1 , key2: value2 } .

>>> query = 'user = pilgrim & database = master & password = PapayaWhip' >>> a_list = query. split ('&') ① >>> a_list ['user = pilgrim' , 'databas e = master' , 'password = PapayaWhip'] >>> a_list_of_lists = [v. split ('=' , 1) for v in a_list] ② >>> a_list_of_lists [['user' , 'pilgrim'] , ['database' , 'master'] , ['password' , 'PapayaWhip']] > >> a_dict = dict (a_list_of_lists) ③ >>> a_dict { 'password' : 'PapayaWhip' , 'user' : 'pilgrim' , 'database' : 'master' }

- ① The `split()` method takes one argument, a delimiter, and splits the delimited string into a list of strings. In this case, the delimiter is the apersand (`&`), but the delimiter can be anything you like.
- ② Now you have a list of strings, each of which consists of a key, an `=` sign and a value. We can use list generators to go through the entire list and split each line at the first `=` into two lines: a key and a value. (In theory, a value can also contain an equal sign. If we just do 'key = value = foo' . Split ('=')), we get a list of three elements ['key' , 'value' , 'foo'] .)
- ③ Finally Python can turn this list into a dictionary using the dict () function.

☞ The previous example is similar to parsing the parameters in a URL, in real life this parsing is much more difficult. If you need to work with URL parameters, it is better to use the urllib function . parse . parse_qs () , which can handle some non-obvious specific cases.

Slitting strings

Once you have created a line, you can get any part of it as a new line. This is called string slicing. Slicing works the same way as slicing lists, which makes sense since strings are the same sequences of characters.

>>> a_string = 'My alphabet starts where your alphabet ends.' >>> a_string [3 : 11] ① 'alphabet' >>> a_string [3 : - 3] ② 'alphabet starts where your alphabet en' >>> a_string [0 : 2] ③ 'My' >>> a_string [: 1 8] ④ 'My alphabet starts' >>> a_string [18 :] ⑤ 'where your alphabet ends.'

- ① You can get any part of the string, the so-called "slice", by specifying two indices. The return value is a new string containing

all the characters in the original string in the same order, starting at the first specified index.

- ② As with list slices, indexes for string slices can be negative.
- ③ The character indexing in the string starts from zero, so a_string [0 : 2] will return the first two elements of the string, starting with a_string [0] (inclusive) and ending (not inclusive) a_string [2] .
- ④ If the slice starts at index 0, then this index can be omitted. Thus a_string [: 18] is the same as a_string [0 : 18] .
- ⑤ Similarly, if the last index is the length of the string, then it can be omitted. That is, a_string [18 :] means the same as a_string [18 : 44] , since there are 44 characters in a string. There is a pleasant symmetry here. In our example, the string contains 44 characters, a_string [: 18] returns the first 18 characters, and a_string [18 :] returns everything but the first 18 characters. In fact, a_string [: n] always returns the first n characters, and a_string [n:] returns the rest, regardless of the length of the string.

Strings versus byte sequence

Bytes are bytes; symbols are an abstraction. An immutable sequence of Unicode characters is called a string (: string). An immutable sequence of numbers - from -0 to -255 called object bytes .

>>> by = b 'abcd \ x 65' ① >>> by b 'abcde' >>> type (by) ② < class 'bytes' > >>> len (by) ③ 5 >>> by + = b ' \ x ff' ④ >>> by b 'abcde \ x ff' >>> len (by) ⑤ 6 >>> by [0] ⑥ 97 >>> by [0] = 102 ⑦ Traceback (most recent call last) : File "<stdin>" , line 1 , in < module > TypeError : ' bytes' object does not support item assignment

- ① Use the "byte string" b " syntax to create a `bytes` object . Each byte in a byte string can be either an ASCII character or an encoded hexadecimal number from `\x00` to `\xff` (0-255).
- ② Byte string type - bytes .
- ③ Similar to lists and strings, you can determine the length of a byte string using the built-in len () function .
- ④ Similar to lists and strings, you can concatenate byte strings using the `+` operator . The result will be a new object of type bytes .
- ⑤ Combining a 5-byte and a 1-byte object will result in a 6-byte object.
- ⑥ Similar to lists and strings, you can get a specific byte from a byte string by its index. The elements of an ordinary string are strings, and the elements of a byte string are integers. Specifically, numbers from 0 to 255.
- ⑦ The byte string is immutable. You cannot change any bytes in it. If you need to change individual bytes, you can either use the concatenation operator (`+`), which works the same as with strings, or convert a bytes object to a bytearray object .

>>> by = b 'abcd \ x 65'
>>> barr = bytearray (by) ① >>> barr bytearray (b 'abcde') >>> len (barr) ② 5 >>> barr [0] = 102 ③ >>> barr bytearray (b 'fbcde')

- ① Use the built-in bytearray () function to convert a bytes object to a mutable bytearray object .
- ② All the methods and operators that you have used with bytes objects also work with bytearray objects .
- ③ The only difference is that you can change the value of an individual byte when working with a bytearray object . The value to write must be an integer from 0 to 255.

The only thing you cannot do is mix bytes and strings.

>>> by = b 'd'
>>> s = 'abcde' >>> by + s ① Traceback (most recent call last) : File " <stdin>" , line 1 , in < module > TypeError : can ' t concat bytes to str >>> s.count (by) ② Traceback (most recent call last): File "<stdin>", line 1, in <module> TypeError: Can ' t convert ' bytes' object to str implicitly >>> s. count (by. decode ('ascii')) ③ 1

- ① you cannot concatenate bytes and a string. These are two different types of data.
- ② You cannot calculate the frequency of occurrence of a sequence of bytes in a string, because there are no bytes in the string at all. A string is a sequence of characters. Perhaps you mean "count the number of occurrences of a string obtained by encoding a sequence of bytes from a specific encoding"? Then it must be specified exactly. Python 3 will not automatically convert bytes to

strings or strings to bytes.

- ③ Coincidentally, this line of code means "count the number of occurrences of a string obtained by decoding a sequence of bytes from a particular encoding".

This is where the relationship between strings and bytes comes in: an object of type bytes has a decode () method that takes an encoding as an argument and returns a string. In turn, the string has an encode () method that takes an encoding as an argument and returns a bytes object . In the previous example, decoding was relatively simple: a sequence of ASCII bytes was converted to a string. But this process is fine for any encoding that supports string characters, even legacy (non-Unicode) encodings.

>>> a_string = ' 深入 Python' ① >>> len (a_string) 9 >>> by = a_string. encode ('utf-8') ② >>> by b ' \ x e6 \ x b7 \ x b1 \ x e5 \ x 85 \ x a5 Python' >>> len (by) 13 >>> by = a_string. encode ('gb18030') ③ >>> by b ' \ x c9 \ x ee \ x c8 \ x eb Python' >>> len (by) 11 >>> by = a_string. encode ('big5') ④ >>> by b ' \ x b2` \ x a4J Python' >>> len (by) 11 >>> roundtrip = by. decode ('big5') ⑤ >>> roundtrip ' 深入 Python' >>> a_string == roundtrip True

- ① These are strings. It has 9 characters.
- ② This is a bytes object . It has 13 bytes. This is a sequence of bytes obtained by encoding a_string in UTF-8 encoding.
- ③ This is a bytes object . It has 11 bytes. This is a sequence of bytes obtained by encoding a_string in GB18030 encoding.
- ④ This is a bytes object . It has 11 bytes. This is a sequence of bytes obtained by encoding a_string in Big5 encoding.
- ⑤ This is a string. It consists of nine characters. It is a sequence of characters that you get after decoding by using the Big5 encoding algorithm. The resulting string is the same as the original.

PS Encoding in Python source code

Python 3 assumes that your source code is - i.e. each .py file is written in UTF-8 encoding.

☞ In Python 2, the default encoding for .py files was ASCII. In Python 3, the default encoding is UTF-8.

If you want to use a different encoding in your code, you can place an encoding declaration on the first line of each file. For example, for the windows-1252 encoding, the declaration looks like this:

```
# - * - coding: windows-1252 - * -
```

The encoding declaration can also appear on the second line of the file if the first line is the path to the Python interpreter.

```
#! / usr / bin / python3
# - * - coding: windows-1252 - * -
```

For more information refer to PEP 263: Defining Python Source Code Encodings .

Further reading

- On Unicode in Python:

- Python Unicode HOWTO
- What's New In Python 3: Text vs. Data Instead Of Unicod e vs. 8-bit
- PEP 261 explains how Python handles astral characters outside of the Basic Multilingual Plane (ie characters whose ordinal value is greater than 65535)
- On Unicode in general:
- The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and * Character Sets (No Excuses!)
- On the Goodness of Unicode
- On Character Strings
- Characters vs. Bytes
- On character encoding in other formats:
- Character encoding in XML
- Character encoding in HTML
- On strings and string formatting:
- string - Common string operations
- Format String Syntax
- Format Specification Mini-Language
- PEP 3101: Advanced String Formatting

Regular expressions

❝ Some people, while solving one problem, think: "I know, I will use regular expressions." Now they have two problems ... ❞ - Jamie Zawinski

Immersion

Every new programming language has built-in functions for working with strings. In Python, strings have methods for finding and replacing: `index()`, `find()`, `split()`, `count()`, `replace()`, etc. But these methods are limited for the simplest cases. For example, the `index()` method looks for a simple hard-coded portion of a string, and the search is always case sensitive. To

perform a case-insensitive search on string *s* , you must call s.lower () or s.upper () to make sure the string is case-sensitive to search. The replace () and split () methods have the same limitations.

If your problem can be solved using these methods, it is better to use them. They are simple and fast, easy to read, a lot can be said about fast, simple and readable code. But if you find yourself using a large number of string functions with if conditions to handle special cases, or using multiple sequential calls to split () and join () to slice your strings, then you need regular expressions.

Regular expressions are a powerful and (for the most part) standardized way to find, replace and parse text using complex patterns. Although the syntax of regular expressions is quite complex and looks unlike normal code, the end result is often *more* readable than a set of sequential functions for strings. There is even a way to put comments inside the regex, so you can include a little documentation in the regex.

> If you've used regular expressions in other languages (such as Perl, JavaScript, or PHP), the Python syntax will be fairly familiar to you. Read the re module overview for the available functions and their arguments.

***

Case Study: Street Address

This series of examples is based on real-life problems that appeared in my work a few years ago, when I had to process and standardize street addresses exported from the legacy system before importing into the new system. (Please note: this is not a fictional example, you can still use it). This example shows how I approached the problem:

>>> s = '100 NORTH MAIN ROAD' >>> s. replace ('ROAD' , 'RD.') ① '100 NORTH MAIN RD.' >>> s = '100 NORTH BROAD ROAD' >>> s. replace ('ROAD' , 'RD.') ② '100 NORTH BRD. RD. ' >>> UNIQae610d7ca506639d-nowiki- 0000 0003 -QINU ③ '100 NORTH BROAD RD.' >>> import re ④ >>> re . sub ('ROAD $' , 'RD.' , s) ⑤ '100 NORTH BROAD RD.'

- ① My task is to standardize the street address, for example 'ROAD' is always abbreviated as 'RD.' ... At first glance, it seemed to me that this is simple enough that I can use the replace () method . After all, all the data is already in uppercase and case mismatch is not a problem. The search string 'ROAD' was a constant and deceptively simple example of s . replace () probably works.
- ② Life, on the other hand, is full of conflicting examples, and I quickly discovered one of them. The problem was that 'ROAD' appeared in the address twice, once as 'ROAD' and the other as part of the street name 'BROAD' . The r eplace () method found 2 occurrences and blindly replaced both, thus destroying the correct address.
- ③ To solve this problem of occurrence of more than one substring of 'ROAD' , you need to resort to the following: search and replace 'ROAD' in the last four characters of the address (s [- 4 :]), leaving the string alone (s [: - 4]) ... As you can see, this is already getting cumbersome. For example, the pattern depends on the length of the string being replaced. (If you replaced 'STREET' with 'ST.' , You will have to use s [: - 6] and s [- 6 :] . Replace (...) .) Would you like to come back to this code six months later for debugging ? I would not want to.
- ④ It's time to move on to regular expressions. In Python, all regex related functions are contained in the re module .
- ⑤ Let's take a look at the first parameter: 'ROAD $' . This is a simple regular expression that only finds 'ROAD' at the end of a line. The $ sign means "end of line". (There is also a ^ , which stands for "start of line".) Using the re . sub () you look for the

regular expression 'ROAD $' in string s and replace it with 'RD .' ...
It matches 'ROAD' at the end of s, but *does not* match 'ROAD' , which is part of the name 'BROAD' , since it is in the middle of s.

Continuing the story of address processing, I soon discovered that the previous example of matching 'ROAD' at the end of an address was not good enough, since not all addresses included street definitions. Some addresses just ended with a street name. I avoided this most of the time, but if the street name was 'BROAD' then the regex matched the 'ROAD' at the end of the 'BROAD' line , which I didn't want at all.

>>> s = '100 BROAD' >>> re . sub ('ROAD $' , 'RD.' , s) '100 BRD.' >>> re . sub (' \\ bROAD $' , 'RD.' , s) ① '100 BROAD' >>> re . sub (r ' \ b ROAD $' , 'RD.' , s) ② '100 BROAD' >>> s = '100 BROAD ROAD APT. 3 ' > >> re . sub (r ' \ b ROAD $' , 'RD.' , s) ③ '100 BROAD ROAD APT. 3 ' >>> re . sub (r ' \ b ROAD \ b ' , 'RD.' , s) ④ '100 BROAD RD. APT 3 '

- ① In *fact*, I would like to match 'ROAD' when it is at the end of the line , *and* is an independent word (and not part of a larger). To describe this in a regular expression, you need to use ' \ b ' , which means "the word should go right here." This is tricky in Python, since the ' \' character in a string must be escaped. This is sometimes referred to as the "backslash disaster" and is one of the reasons why regular expressions are easier in Perl than in Python. However, the disadvantage of Perl is that regular expressions are mixed with other syntax, if you have an error it is difficult to

determine where it is, in the syntax or in the regular expression.

- ② To get around the problem of "disaster backslash" you can use what is called *an unformatted string* (*the raw: string)* , by applying a string prefix with the symbol 'r' . This tells Python that nothing on that line should be escaped; '\t' is a tab, but r'\t' is a backslash character '\' followed by the letter 't' . I recommend that you always use an unformatted string when dealing with regular expressions; on the other hand, things are getting quite confusing (even though our regex is already confusing enough).
- ③ * *sigh* * Bad luck, I soon found more reasons to contradict my logic. In this case, the street address contained a single single word 'ROAD' and it was not at the end of the line, since the address contained the apartment number after the street was determined. Since the word 'ROAD' is not at the end of the line, the regular expression re . sub () skipped it and we got the same string as input, which is what you don't want.
- ④ To solve this problem I removed the '$' character and added another '\b' . The regular expression now matches 'ROAD' if it was a whole word anywhere in the string, at the end, in the middle, and at the beginning.

⁂

Case Study: Roman Numerals

You have most likely seen Roman numerals, even if you don't understand them. You may have seen them on copyrighted old films and TV shows ("Copyright MCMXLVI " instead of "Copyright 1946 "), or on walls in university libraries ("established by MDCCCLXXXVIII " instead of " established 1888 "). You could see them in the structure of bibliographic references. This number display system dates back to the ancient Roman Empire (hence the name).

Roman numerals have seven characters that are repeated in various combinations to represent numbers.

- I = 1
- V = 5
- X = 10
- L = 50

- C = 100
- D = 500
- M = 1000

The following rules allow you to construct Roman numerals:

- Sometimes the symbols add up. I is 1 , II is 2 , and III is 3 . VI is 6 (character by character, " 5 and 1 "), VII is 7 , and VIII is 8 .
- Decimal characters (I , X , C , and M) can be repeated up to 3 times. To form 4, you need to subtract 5 from the next highest symbol. You cannot write 4 as IIII ; instead, it is written as IV (" 1 less than 5 "). 40 is written as XL (“ 10 less than 50 ”), 41 as XLI , 42 as XLII , 43 as XLIII , and 44 as XLIV (“ 10 less than 50 , and 1 less than 5 ”).
- Sometimes symbols ... are the opposite of addition. By placing certain characters before others, you subtract them from the final value. For example, 9 , you need to subtract ten from the next highest character: 8 is VIII , but 9 is IX (" 1 less than 10 "), not VI III (since I cannot be repeated 4 times). 90 is XC , 900 is CM .
- Fives cannot be repeated. 10 is always displayed as X , never as VV . 100 is always C , never LL .
- Roman numerals are read from left to right, so the position of the character matters a lot. DC is 600 ; CD is a completely different figure (400 , " 100 less than 500 "). CI is 101 ; IC is not even a valid Roman number (since you cannot subtract 1 directly from 100 ; you need to write this as XCIX , " 10 less than 100 , and 1 less than 10 ").

Check for thousands

What should be done to check that an arbitrary string is a valid Roman number? Let's take one character at a time. Since Roman numbers are always written from highest to lowest, let's start with the highest: the thousandth position. For numbers 1000 and above, the symbols M are used .

>>> import re >>> pattern = '^ M? M? M? $' ① >>> re . search (pattern ,

'M') ② UNIQae610d7ca506639d-nowiki- 00000039 -QINU >>> re . search (pattern , 'MM') ③ <_sre.SRE_Match ob ject at 0106C290> >>> re . search (pattern , 'MMM') ④ <_sre.SRE_Match object at 0106AA38> >>> re . search (pattern , 'MMMM') ⑤ >>> re . search (pattern , '') ⑥ <_sre.SRE_Match object at 0106F4A8>

- ① This pattern has three parts. ^ matches the beginning of the line. If you do not specify it, the pattern will match M without taking into account the position in the string, which is not what we want. You must make sure that the M characters, if present, are at the beginning of the line. M? Optionally matches one M character . Since this is repeated three times, the pattern will match from zero to three times with the M character in the string. And the $ character will match the end of the line. When combined with the ^ at the beginning, it means the pattern must match the entire string, with no other characters before or after the M characters.
- ② The essence of the re module is the search () function , which uses the regular expression *pattern* (*pattern*) and the string ('M') and searches for matches against the regular expression. If a match is found, search () returns an object that has various methods for describing the match; if no match is found, search () returns None , in Python the value is *null* . All we care about at the moment, whether the pattern matches, you can tell by looking at the value returned by the search () function . 'M' matches this regex, as the first optional M matches, and the second optional M and the third are ignored.
- ③ 'MM' matches as the first and second optional M match and

the third is ignored

- ④ 'MMM' matches exactly as all three Ms match
- ⑤ 'MMMM' does not match. All three Ms match, but the regular expression insists on the end of the line (since it requires the $ character), and the line hasn't ended yet (because of the fourth M). Therefore search () returns None .
- ⑥ Interestingly, the empty string also matches the regular expression, since all M characters are optional.

Checking for hundreds

? makes the pattern optional

The location of hundreds is more complex than thousands because there are multiple mutually exclusive write paths and depends on the value.

- 100 = C
- 200 = CC
- 300 = CCC
- 400 = CD
- 500 = D
- 600 = DC
- 700 = DCC
- 800 = DCCC
- 900 = CM

Thus, there are four possible patterns:

- CM
- CD
- Zero to three C characters (zero if revenge hundreds is empty)
- D , from subsequent zeros to three C characters

The last two patterns are combined:

- an optional D followed by zero to three C characters

This example shows how to check the position of a hundred in a Roman number.

>>> import re >>> pattern = '^ M? M? M? (CM | CD | D? C? C? C?) $' ①

>>> re . search (pattern , 'MCM') ② UNIQae610d7ca506639d-n owiki-00000040 -QINU >>> re . search (pattern , 'MD') ③ UNIQae610d7ca506639d-nowiki- 00000041 -QINU >>> re . search (pattern , 'MMMCCC') ④ UNIQae610d7ca506639d-nowiki- 00000042 -QINU >>> re . search (pattern , 'MCMC') ⑤ >>> re . search (p attern , '') ⑥ UNIQae610d7ca506639d-nowiki- 00000043 -QINU

- ① This pattern starts the same way as the previous one, checking first lines (^), then thousands (M? M? M?). This is followed by a new part in brackets that describes three mutually exclusive patterns separated by a vertical line: CM , CD and D? C? C? C? (which is an optional D followed by zero to three optional C characters). The regular expression parser checks each of these patterns in sequence from left to right, picking the first one that matches and ignoring the next.
- ② 'MCM' matches because the first M matches, the second and third M characters are ignored, the CM characters match (and the CD and D? C? C? C? Patterns are not parsed after that). MCM is the Roman representation of 1900 .
- ③ 'MD' matches because the first M matches, the second and third M are ignored, and the pattern D? C? C? C? Matches D (three C characters are optional and ignored). MD is the Roman representation of 1500 .
- ④ 'MMMCCC' matches because the first M matches, and the pattern D? C? C? C? matches CCC (D is optional and ignored). MMMCCC i is the Roman representation of 3300 .

- ⑤ 'MCMC' does not match. The first M matches, the second and third Ms are ignored, and CM also matches , but the $ pattern does not match because you are not at the end of the line yet (you still have a mismatch C). Symbol C *do not* coincide as part Paterna D? C? C? C? since the excluding CM pattern already matched.
- ⑥ Interestingly, the empty string still matches the regular expression, since all M characters are optional and ignored and the empty string matches the D? C? C? C? where all characters are optional and ignored.

Shit! Have you noticed how quickly regular expressions get nasty? And so far, we have processed only the positions of thousands and hundreds in the Roman representation of numbers. But if you follow along, you will find that tens and ones will be easier to describe, since they have the same pattern. In the meantime, let's look at another way to describe this pattern.

⁂

Using the syntax {n, m}

modifier {1,4} matches 1 to 4 occurrences of the pattern

In the previous section, we dealt with a pattern where the same symbol can be repeated up to three times. There is another way to write this regular expression that many people will find more readable. First, let's take a look at the method we already used in the previous example.

>>> import re >>> pattern = '^ M? M? M? $' >>> re . search (pattern , 'M') ① UNIQae610d7ca506639d-nowiki- 00000045 -QINU >>> pattern = '^ M? M? M? $' >>> re . search (pattern , 'MM') ② UNIQae610d7ca506 639d-nowiki- 00000046 -QINU >>> pattern = '^ M? M? M? $' >>> re . search (pattern , 'MMM') ③ UNIQae610d7ca506639d-nowiki- 00000047 -QINU >>> re . search (pattern , 'MMMM') ④ >>>

- ① Here, the pattern matches the beginning of the line and the first optional M, but not the second and third (but this is normal since they are optional), as well as the end of the line.
- ② Here the pattern matches the beginning of the line, with the first and second optional M characters, but not with the third (this is normal since it is optional) and the end of the line.
- ③ Here the pattern matches the beginning of the line and all three optional M characters and also the end of the line.
- ④ Here the pattern matches the beginning of the line and all three optional M characters, but does not match the end of the line (since there is another M), so the pattern does not match and returns **None** .

>>> pattern = '^ M {0,3} $' ① >>> re . search (pattern , 'M') ② UNIQae610d7ca506639d-nowiki- 00000049 -QINU >>> re . search (pattern , 'MM') ③ <_sre.SRE_Match object at 0x008EE090> >>> re . search (pattern , 'MMM') ④ <_sre.SRE _Match object at 0x008EEDA8> >>> re . search (pattern , 'MMMM') ⑤ >>>

- ① This pattern says: "Match the beginning of the line, then from zero to three M characters anywhere, then the end of the line." Characters 0 and 3 can be any digits, if you need to match 1 or more

M characters, you must write M {1,3}.

- ② Here the pattern matches the beginning of the line, then with one of the possible three characters M, then with the end of the line.
- ③ Here the pattern matches the beginning of the line, then with two of the possible three characters M, then with the end of the line.
- ④ Here the pattern matches the beginning of the line, then with three of the possible three characters M, then with the end of the line.
- ⑤ Here the pattern matches the beginning of the line, then with two of the possible three characters M, but *does not match* the end of the line.

The regular expression allows up to three M characters to the end of the line, but you have four and the pattern returns **None** .

Checking for tens and ones

Now, let's expand the regular expression to include tens and ones. This example shows checking for tens.

>>> pattern = '^ M? M? M? (CM | CD | D? C? C? C?) (XC | XL | L? X? X? X?) $' >>> re . search (pattern , 'MCMXL') ① <_sre.SRE_Match object at 0x008EEB48> >>> re . search (pattern , 'MCML') ② <_sre.SRE_Match object at 0x008EEB48> >>> re . search (pattern , 'MCMLX') ③ UNIQae610d7c a506639d-nowiki- 0000004F -QINU >>> re . search (pattern , 'MCMLXXX') ④ UNIQae610d7ca506639d-nowiki- 00000050 -QINU >>> re . search (pattern , 'MCMLXXXX') ⑤ >>>

- ① Here the pattern matches the beginning of the line, then the first optional character M, then CM, then XL, then the end of the line. Remember that the syntax (A | B | C) means "match only one oz of characters A, B, or C" We match XL and we ignore XC and L? X? X? X? and then go to the end of the line. MCMXL is the Roman representation of 1940.
- ② Here the pattern matches the beginning of the line, then the first optional character M, then CM, then L? X? X? X ?. Of L? X? X? X? Matches L and skips three optional Xs. Then, it goes to the end of the line. MCML is the Roman representation of 1950.
- ③ Here the pattern matches the beginning of the line, then with the first optional character M, then CM, then with the optional L and the first optional X, skipping the second and third optional characters X, then goes to the end of the line. MCMLX is the Roman representation of 1960.
- ④ Here the pattern matches the beginning of the line, then with the first optional character M, then CM, then with the optional L and all three optional characters X, then goes to the end of the line. MCMLXXX is the Roman representation of 1980 .
- ⑤ Here the pattern matches the beginning of the line, then with the first optional character M, then CM, then with the optional L and all three optional characters X, after that it *does not match* the end of the line, since there is one more X character, so the pattern does not work and returns **None** . MCMLXXXX is not a valid Roman number.

(A | B) coincides with either A or B.

The same pattern is suitable for describing units. I'll reduce the detail and show the final result.

>>> pattern = '^ M? M? M? (CM | CD | D? C? C? C?) (XC | XL | L? X? X? X?) (IX | IV | V? I? I? I?) $ '

So what would it look like using the alternative syntax {n, m} ? This example shows the new syntax.

>>> pattern = '^ M {0,3} (CM | CD | D? C {0,3}) (XC | XL | L? X {0,3}) (IX | IV | V? I { 0,3}) $ ' >>> re . search (pattern , 'MDLV') ①

UNIQae610d7ca506639d-nowiki- 0000005 3 -QINU >>> re . search (pattern , 'MMDCLXVI') ② UNIQae610d7ca506639d-nowiki- 00000054 -QINU >>> re . search (pattern , 'MMMDCCCLXXXVIII') ③ UNIQae610d7ca506639d-nowiki- 00000055 -QINU >>> re . search (pattern , 'I') ④ UNIQae610d7ca506639d-nowiki- 00000056 -QINU

- ① Here the pattern matches the beginning of the string, then one of three possible characters M, then D? C {0,3} . Of these, only the optional D and none of the optional Cs match. Further, the optional L from L? X {0.3} and none of the three optional X matches. After that, it matches V from V? I {0.3} and neither one of three optional I's and finally with the end of the line. MDLV is the Roman representation of 1555 .
- ② Here the pattern matches the beginning of the string, then two of the three possible characters M, then D and one optional C from D? C {0,3} . Then L? X {0,3} with L and one of three possible X, then V? I {0,3} with V and one of the three I, then with the end of the line. MMDCLXVI is the Roman representation of 2666 .
- ③ Here the pattern coincides with the beginning of the string, then with three of three M, then D and C from D? C {0,3}, then L? X {0,3} with L and three out of three X, then V? I {0,3} with V and three out of three I's, then end of line. MMMDCCCLXXXVIII is the Roman representation of 3888 , and it is the longest Roman number that can be written without extended syntax.
- ④ Look carefully. (I feel like a magician, "Look carefully kids, now the rabbit will come out of my hat ;)" Here the beginning of the line matches, none of the three Ms, then D? C {0,3} skips an optional D and three optional Cs, then L ? X {0,3} omitting the optional L and three optional Xs, then V? I {0,3} omitting the

optional V and one of the three optional I's . Then the end of the line. Stop, fuf.

If you followed everything and understood on the first try, then you are doing better than me. Now imagine that you are trying to figure out someone's regular expressions in an important function in a huge program. Or, for example, imagine that you are returning to your own program in a few months. I've done it and it's not a very pleasant sight.

For now, let's explore an alternative syntax that will make it easier to maintain your expressions.

⁂

Verbose Regular Expressions

So far, you've dealt with what I call "compact" regular expressions. As you can see, they are difficult to read, even if you know what they are doing. There is no guarantee that you will be able to figure them out after six months. What you really need is the nested documentation

Python allows you to do this with *verbose regular expressions* . Detailed regular expressions differ from compact ones in two ways:

- Blank lines are ignored, spaces, tabs, and carriage returns do not match, respectively. They don't match at all. (If you want to match a space in a verbose regular expression, you need to put a backslash in front of it.)
- Comments are ignored. A comment in a verbose regular expression is the same as a comment in Python code: it starts with a # and continues to the end of the line. In this case, this comment is a comment within a multi-line line, but it works the same way as a simple one.

An example will make it clearer. Let's double-check the compact regex we worked with and create a verbose regex. This example is shown below.

>>> pattern = " ' ^ # beginning of line M {0,3} # thousands - 0 to 3 M (CM | CD | D? C {0,3}) # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 C), # or 500-800 (D, followed by 0 to 3 C) (XC | XL | L? X {0.3}) # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 X), # or 50-80 (L, followed by 0 to 3 X) (IX | IV | V? I

{0.3}) # units - 9 (IX), 4 (IV), 0-3 (0 to 3 I), # or 5-8 (V, followed by 0 to 3 I) $ # end of line " ' >>> re . search (pattern , 'M' , re . VERBOSE) ① UNIQae610d7ca506639d-nowiki- 00000058 -QINU >>> re . search (pattern , 'MCMLXXXIX' , re . VERBOSE) ② UNIQae610d7ca506639d-nowiki-00000059 -QINU >>> re . search (pattern , 'MMMDCCCLXXXVIII' , re . VERBOSE) ③ <_sre.SRE_Match object at 0x008EEB48> >>> re . search (pattern , 'M') ④

- ① The main thing to remember is that you need to add extra arguments to work with them: re.VERBOSE is a constant defined in the re module that serves as a signal that the pattern should be used as a verbose regular expression. As you can see, this pattern contains a large number of blank lines. (and they are all ignored) as well as a few comments (which are also ignored). If we ignore comments and blank lines, we get the same regular expression as in the previous example, but in a much more readable form.
- ② Here the beginning of the line matches, then one and three possible M , then CM , then L and three of the possible X , then IX , then the end of the line.

- ③ There coincides beginning of the line, then three of the three possible M , then D and three of the possible three C , then L and three of the possible three X , then V and three out of three possible I , then the end of the line.
- ④ It doesn't match here. Why? Since the re.VERBOSE flag is missing and the re.search function treats the pattern as a compact regular expression, with significant spaces and # symbols. Python cannot automatically determine if a regular expression is verbose or not. Python treats every regexp as compact unless you specify that it is verbose.

*
**

Case Study: Handling Phone Numbers

\ d matches any digits (0-9). \ D matches everything except digits

So far, you have been concentrating on complete patterns. Matches or doesn't match, but regular expressions can be much more powerful than that. When a regular expression matches something, you can get a specially highlighted part of the match. You can find out what matched and where.

This example emerged from another real problem I experienced in my previous job. The problem was handling American phone numbers. The client wants to enter a phone number in a simple field (no delimiters), but then also wants to store the postcode, trunk, number and optionally additional information in the company database. I searched the internet and found many examples of a regular expression that should do this, but unfortunately none of the solutions worked.

Here are the phone numbers that I had to process:

- 800-555-1212
- 800 555 1212
- 800.555.1212
- (800) 555-1212
- 1-800-555-1212

- 800-555-1212-1234
- 800-555-1212x1234
- 800-555-1212 ext. 1234
- work 1- (800) 555.1212 # 1234

There are enough options in any of these examples. I needed to know code 800 , highway 555 and the rest of the number 1212 . For those with extensions, I needed to know that extension 1234

Let's start developing a solution for handling a phone number. This example shows the first step :

>>> phonePattern = re . compile (r '^ (\ d {3}) - (\ d {3}) - (\ d {4}) $') ① >>> phonePattern. search ('800-555-1212') . groups () ② ('800' , '555' , '1212') >>> phonePattern. search ('800-555-1212-1234') ③ >>> phonePattern. search ('800-555-1212-1234') . groups () ④ Traceback (most recent call last) : File "<stdin>", line 1, in <module> AttributeError : 'NoneType' object has no attribute 'groups'

- ① Always read the regular expression from left to right. The expression matches the beginning of the line and then (\ d {3}) . What is \ d {3} ? So, \ d means "any digit" (0 to 9). {3} means "match specifically with three digits"; these are variations on the {n, m} syntax you saw earlier. If you enclose this expression in parentheses, it means "exactly three numbers must match and then *remember them as a group that I will request later* ." Then the expression must match the hyphen. Then match with another group of three digits, then again a hyphen. Then another group of four numbers. And at the end it matches the end of the line.

- ② To access the groups that the regular expression handler remembered, use the g roups () method on the object that returns the search () method . It should return a tuple of as many groups as specified in the regular expression. In our case, three groups are defined, one with three digits, another with three digits, and the third with four digits.
- ③ This regex is not a definitive answer as it does not handle the extension after the phone number. To do this, you have to extend the regular expression.
- ④ This is why you should not use the "chaining" of search () and groups () methods in production code. If the search () method does not return a match, then it will return None , this is not a standard regular expression object. Calling None.groups () throws an obvious exception: N one does not have a groups () method . (Of course, this is a little less obvious when you get this exception from the depths of your code. Yes, my experience says so now.)

>>> phonePattern = re . compile (r '^ (\ d {3}) - (\ d {3}) - (\ d {4}) - (\ d +) $') ① >>> phonePattern. search (' 800-555-1212-1234') . groups () ② ('800' , '555' , '1212' , '1234') >>> phonePattern. search ('800 555 1212 1234') ③ >>> >>> phonePattern. search ('800-555-1212') ④ >>>

- ① This regular expression is almost identical to the previous one. As before, it coincides with the beginning of the line, then with a memorized group of three numbers, then a hyphen, then a memorized group of three numbers, then a hyphen, then a memorized group of four numbers. What's new? This is a match with another hyphen and a memorized group of one or more digits.
- ② The groups () method now returns a tuple of four elements, and the regular expression now remembers four groups.

- ③ Unfortunately, this regex is not the final answer, as it assumes the different parts of the number are separated by a hyphen. What happens if they are separated by spaces, commas, or periods?
- ④ You need a more general solution to match different types of separators.

Shit! What this regular expression does is not quite what you want. In fact, this is even a step backwards as you cannot handle phone numbers without extensions. This is not at all what you wanted; if there is an extension, you would like to know what it is, but if it does not exist, you still want to know the different parts of the phone number.

The following example shows how a regular expression handles separators between different parts of a phone number.

>>> phonePattern = re . compile (r '^ (\ d {3}) \ D + (\ d {3}) \ D + (\ d {4}) \ D + (\ d +) $') ① >>> phonePattern. search ('800 555 1212 1234') . groups () ② ('800' , '555' , '1212' , '1234') >>> phonePattern. search ('800-555-1212-1234') . groups () ③ ('800' , '555' , '1212' , '1234') >>> phonePattern. search ('80055512121234') ④ >>> >>> ph onePattern. search ('800-555-1212') ⑤ >>>

- ① Keep your hat on. You match the beginning of a line, then a group of three digits, then \ D +. What the hell is this? Ok, \ D matches any character other than numbers and also "+" means "1 or more". So \ D + means one or more non-digit characters. This is what you use instead of the hyphen "-" to match any delimiters.

- ② Using \ D + instead of "-" means that now the regular expression matches a phone number separated by spaces instead of hyphens.
- ③ Of course, hyphenated phone numbers also work.
- ④ Unfortunately, this is not yet the final answer, as it implies a delimiter. What if the number is entered without any separators?
- ⑤ Optsayo! And the expansion problem has not yet been resolved. Now you have two problems, but you can handle them using the same technique.

The following example shows a regular expression for handling telephone numbers without separators.

>>> phonePattern = re . compile (r '^ (\ d {3}) \ D * (\ d {3}) \ D * (\ d {4}) \ D * (\ d *) $') ① >>> phonePattern. search ('80055512121234') . groups () ② ('800' , '555' , '1212' , '1234') >>> phonePattern. search ('800.555.1212 x1234') . groups () ③ ('800' , '555' , '1212' , '1234') >>> phonePattern. search ('800-555-1212') . groups () ④ ('800' , '555' , '1212' , '') >>> phonePattern. search ('(800) 5551212 x1234') ⑤ >>>

- ① Only one change, replacing "+" with "*". Instead of \ D + between parts of the number, \ D * is now used. Remember that "+" means "1 or more"? Ok, "*" means "zero or more". So now you can process the number even if it doesn't contain delimiters.
- ② Just think, it really works. Why? You have the same beginning of the line, then a group of three digits (800) is

remembered, then zero or more non-numeric characters, then a group of three digits (555) is memorized, then zero or more non-numeric characters, then a group of four digits is memorized (1212), then zero or more non-digital characters, then a group of an arbitrary number of digits (1234) is stored, then the end of the line.

- ③ Variations also work: periods instead of hyphens, and spaces or "x" before expansion.
- ④ Finally you have solved a long-standing problem: the extension is optional again. If no extension is found, the groups () method still returns four elements, but the fourth element is just an empty string.
- ⑤ I hate to be the bad news messenger, but you're not done yet. What is the problem here? There are additional characters before the "area" code, but the regular expression thinks that the area code is the first one at the beginning of the line. No problem, you can use the same "zero or more non-numeric characters" technique to skip the starting characters before the area code.

The following example shows how to work with characters up to a phone number.

```
>>> phonePattern = re . compile ( r '^ \ D * ( \ d {3}) \ D * ( \ d {3}) \ D * ( \ d {4}) \ D * ( \ d *) $' ) ① >>> phonePattern. search ( '(800) 5551212 ext. 1234' ) . groups () ② ( '800' , '555' , '1212' , '1234' ) >>> phonePattern. search ( '800-555-1212' ) . groups () ③ ( '800' , '555' , '1212' , '' ) >>> phonePattern. search ( 'work 1- (800) 555.1212 # 1234' ) ④ >>>
```

- ① This is the same as in the previous example, except for \ D *, zero or more non-numeric characters, up to the first memorized

group (area code). Note that you do not remember those non-numeric characters before the area code (they are not in parentheses). If you find them, you just skip them and remember the area code.

- ② You can successfully process a phone number even with parentheses before the area code. (The right parenthesis is also treated; as a non-digit character and matches \ D * after the first group to remember.)
- ③ A simple check to see if we have broken something that should have worked. Since the leading characters are completely optional, the beginning of the line matches, zero non-digital characters, then a group of three digits (800) is remembered, then one non-digital character (hyphen), then a group of three digits (555), then one non-digital (hyphen), then a group of four digits (1212) is remembered, then zero non-digit characters, then a group of digits from zero characters, then the end of the line.
- ④ This is where the regular expression gouges my eyes out with a blunt object. Why didn't this number match? Because 1 is before the area code, but you assumed that all leading characters before the area code are not numbers (\ D *).

Let's go back a second. So far, the regex has matched the beginning of the line. But now you see that at the beginning there may be some unpredictable characters that we would like to ignore. Better not to try to find a match for them, but just skip them all, let's make another assumption: not try to match the beginning of the line at all. This approach is shown in the following example.

>>> phonePattern = re . compile (r '(\ d {3}) \ D * (\ d {3}) \ D * (\ d {4}) \ D * (\ d *) $') ① >>> phonePattern. search ('work 1 - (800) 555.1212 # 1234') . groups () ② ('800' , '555' , '1212' , '1234') >>> phonePattern. search ('800-555-1212') ③ ('800' , '555' , '1212' , '') >>> phonePattern. search ('80055512121234') ④ ('800' , '555' , '1212' , '1234')

- ① Notice the lack of `^` in the regular expression. You are no longer the same as the beginning of the line. Now nothing tells you how to deal with the entered data for your regular expression. The regular expression handler will do the heavy lifting to figure out where the input string will start to match.
- ② Now you can successfully process a phone number that includes leading characters and numbers, plus any type of separator between parts of the number.
- ③ Simple check. Everything is working.
- ④ And even that works too.

See how quickly the regex gets out of hand? Let's take a look at the previous iterations. Can you explain the difference between one and the other?

As long as you understand the final answer (and this is really it; if you find a situation that it does not handle, I do not want to know about it), we will write a detailed regular expression, until you forget why you made the choice you made.

>>> phonePattern = re . compile (r " ' # don't match beginning of string, number can start anywhere (\ d {3}) # area code is 3 digits (eg' 800 ') \ D * # optional separator is any number of non- digits (\ d {3}) # trunk is 3 digits (eg '555') \ D * # optional separator (\ d {4}) # rest of number is 4 digits (eg '1212') \ D * # optional separator (\ d *) # extension is optional and can be any number of digits $ # end of string " ' , re . VERBOSE) >>> phonePattern. search ('work 1- (800) 555.1212 # 1234') . groups () ① ('800' , '555' , '1212' , '1234') >>> phonePattern. search ('800-555-1212') ② ('800' , '555' , '1212' , ")

- ① Apart from being split over many lines, this is exactly the same regular expression as in the last step, and it shouldn't come as a surprise that it handles the same input.
- ② Final simple check. Yes, it still works. You did it.

Outcome

This is just the tip of the iceberg of what regular expressions can do. In other words, even if you are completely overwhelmed by them now, trust me, you haven't seen anything yet.

You should now be proficient in the following technique:

^ matches the beginning of a line.
$ matches the end of the line.
\ b matches a word boundary.
\ d matches a digit.
\ D matches a non-digit.
x? matches the optional character x (in other words, zero or one x characters).
x * matches zero or more x.
x + matches one or more x.
x {n, m} matches x at least n times, but no more than m times.
(a | b | c) matches a or b or c.
(x) memorization group. You can get the value using the groups () method on the object that re.search returns.

Regular expressions are extremely powerful, but not always the correct way

to solve every problem. You should study more about them to figure out when they are suitable for solving a problem, as sometimes they can add more problems than solve

---

Closures and generators

Immersion

For reasons beyond understanding, I have always admired languages. Not programming languages. Although yes, they, as well as natural languages. Take English for example. English is a schizophrenic language that borrows words from German, French, Spanish, and Latin (not to mention the others). Quite frankly, "borrows" is an inappropriate word; he rather "steals" them. Or, perhaps, "assimilates" - like the Borghi . Yes, that's a good option.

> We are Borghi . Your linguistic and etymological characteristics will become ours. Resistance is futile.

In this chapter, you will learn about plural nouns. You will also learn about functions that return other functions, complex regular expressions and generators. But first, let's talk about how plural nouns are formed. (If you haven't read the section on regular expressions, now is the time. The material in this section assumes that you understand the basics of regular expressions, and you will quickly get to their non-trivial use).

If you grew up in an English speaking country, or studied English in a formal school setting, you are probably familiar with the basic rules:

1. If the word ends with an *S* , *X* or *Z* , add *ES* . *Bass* becomes *basses* , *fax* becomes *faxes* , and *waltz* - *Waltzes* .
2. If the word ends with an *H* , add *ES* ; if the ends deaf *H* , you just need to add *the S* . What is voiced *H* ? It is one that, along with other letters, is combined into a sound that you can hear. Accordingly, *coach* becomes *coaches* and *rash* becomes *rashes* , because you hear *CH* and *SH* sounds when you say these words. But *cheetah* becomes *cheetahs* because *H* is deaf here.

3. If the word ends with *Y* , which reads *I* , then replace *Y* with *IES* ; if the *Y* is combined with the vowel sounds like something different, simply add *the S* . So *vacancy* becomes *vacancies* , but *day* becomes *days* .
4. If none of the rules apply, just add *S* and hope for the best.

(I know there are many exceptions. *Man* becomes *men* , a *woman* - *women* , but *human* becomes *Humans* . *Mouse* - *the mic an e* , and *louse* - *lice* , but the *house* in the plural - *houses* . *Knife* becomes *knives* , and the *wife* becomes *wives* , but *lowlife* becomes *lowlifes* , and don't make me stare at words that don't change in the plural, like *sheep* , *deer* or *haiku*).

The rest of the languages are, of course, completely different.

Let's develop a Python library that automatically plurals an English word. We'll start with these four rules, but keep in mind that you will inevitably need to add more.

I know we use regular expressions!

So you are looking at words, and at least in English, that means you are looking at sequences of characters. You have rules that say that you need to look for different combinations of symbols, then perform various actions with them. It looks like this is a job for regular expressions!

import re def plural (noun) : if re . search ('[sxz] $' , noun) : UNIQd2f4acf57a9a5326-ref- 00000003 -QINU return re . sub ('$' , 'es' , noun) UNIQd2f4acf57a9a5326-ref- 00000006 -QINU elif re . search ('[^ aeioudgkprt] h $' , noun) : return re . sub ('$' , 'es' , noun) elif re . search ('[^ aeiou] y $' , noun) : return re . sub ('y $' , 'ies' , noun) else : return noun + 's'

1. ↑ This is a regular expression, but it uses syntax that you did not see in the chapter Regular Expressions. The square brackets mean "match exactly one of these characters." Therefore [sxz] means " s or x or z ", but only one of them. The $ symbol should be familiar to you. It looks for matches at the end of the string. All regular expression checks whether the ends noun to s , x or the z .
2. ↑ The mentioned function re . sub () performs regular expression based substring substitution.

Let's take a closer look at replacement with a regular expression.

>>> import re >>> re . search ('[abc]' , 'Mark') ① < _sre. SRE_Match object at 0x001C1FA8 > >>> re . sub ('[abc]' , 'o' , 'Mark') ② 'Mork' >>> re . sub ('[abc]' , 'o' , 'rock') ③ 'rook' >>> re . sub ('[abc]' , 'o' , 'caps') ④ 'oops'

1. Does Mark contain characters a , b, or c ? Yes, contains a .
2. Great, now look for a , b or c and replace it with o . Mark becomes Mork .
3. The same function turns rock into rook .
4. You might think this code converts caps to oaps , but it doesn't. re . sub replaces *all* matches, not just the first one found. So, this regex will turn caps into oops , because both c and a are replaced with o .

Let's go back to the plural () function ...

def plural (noun) : if re . search ('[sxz] $' , noun) : return re . sub ('$' , 'es' , noun) ① elif re . search ('[^ aeioudgkprt] h $' , noun) : ② return re . sub ('$' , 'es' , noun) elif re . search ('[^ aeiou] y $' , noun) : ③ return re . sub ('y $' , 'ies' , noun) else : return noun + 's'

1. This is where you replace the end of the line (found with the $ character) with the string es . In other words, add es to the line. You could accomplish the same with string concatenation like noun + 'es' , for example, but I preferred to use regular expressions for each rule, for reasons that will become clear later.

2. Take a look, this regular expression contains something new. The ^ character as the first character in square brackets has a special meaning: negation. [^ abc] means "any single character other than a , b or c ". So [^ aeioudgkprt] means any character other than a , e , i , o , u , d , g , k , p , r, or t . Then this character must be followed by the character h , followed by the end of the line. You are looking for words ending in H that can be heard.

3. Same here: find words that end in Y , where the character before Y is not a , e , i , o, or u . Are you looking for words that end in the Y , which sounds like I of .

Let's take a closer look at regular expressions involving negation.

>>> import re >>> re . search ('[^ aeiou] y $' , 'vacancy') ① < _sre. SRE_Match object at 0x001C1FA8 > >>> re . search ('[^ aeiou] y $' , 'boy') ② >>> >>> re . search ('[^ aeiou] y $' , 'day') >>> >>> re . search ('[^ aeiou] y $' , 'pita') ③ >>>

1. vacancy is appropriate because it ends in cy and c is not a , e , i , o, or u .
2. boy does not match because it ends in oy , and you specifically indicated that the character before y cannot be o . day does not match because it ends in ay .
3. pita doesn't fit because it doesn't end with y .

>>> re . sub ('y $' , 'ies' , 'vacancy') ① 'vacancies' >>> re . sub ('y $' , 'ies' , 'agency') 'agencies' >>> re . sub ('([^ aeiou]) y $' , r ' \ 1 ies' , 'vacancy') ② 'vacancies'

1. This regex will convert vacancy to vacancies and agency to agencies , which is what you want. Note that it would convert boy to boies , but that never happens in the function because you did re first . search to find out whether to do re . sub .
2. Note at the same time that it is possible to combine these two regular expressions (one to find out if the rule is applied, and the other to actually apply it) into one regular expression. This is what the result would look like. Most should be familiar to you: you are using the memorable group that you learned about in Case Study: Parsing a Phone Number. The group is used to remember the character before y . Then in the wildcard string, you use the new syntax, \ 1 , which means “hey, is that the first group you remember? put it here. " So you remember c before y ; when you do a substitution, you put c in place of c , and ies in place of y . (If you have more than one memorable group, you can use \ 2 and \ 3, and so on.)

Regex substitutions are extremely powerful, and the \ 1 syntax makes them even more powerful. But the whole operation, combined into one regular expression, also becomes difficult to read, in addition, this method does not directly correlate with how you originally described the rules for forming the plural. Initially, you designed rules in the form "if a word ends in S , X, or Z ,

add ES ." And if you are looking at a function, then you have two lines of code that say "if the word ends in S , X, or Z , add ES ." It will not work even closer to the original version.

Feature List

Now you will add an abstraction layer. You started by defining a list of rules: if this is true, do that, otherwise refer to the next rule. Let's temporarily complicate part of the program so that you can simplify another part of it.

import re def match_sxz (noun) : return re . search ('[sxz] $' , noun) def apply_sxz (noun) : return re . sub ('$' , 'es' , noun) def match_h (noun) : return re . search ('[^ aeioudgkprt] h $' , noun) def apply_h (noun) : return re . sub ('$' , 'es' , noun) def match_y (noun) : ① return re . search ('[^ aeiou] y $' , noun) def apply_y (noun) : ② return re . sub ('y $' , 'ies' , noun) def match_default (noun) : return True def apply_default (noun) : return noun + 's' rules = ((match_sxz , apply_sxz) , ③ (match_h , apply_h) , (match_y , apply_y) , (match_default , apply_default)) def plural (noun) : for matches_rule , apply_rule in rules: ④ if matches_rule (noun) : return apply_rule (noun)

1. Now each rule-condition of the match is a separate function that returns the results of calling the re function . search () .
2. Each action rule is also a separate function that calls the re function . sub () to apply the appropriate plurality rule.
3. Instead of a single function (plural ()) with multiple rules, you now have a rules data structure , which is a sequence of function pairs.
4. Since the rules are deployed in a separate data structure, the new plural () function can be reduced to a few lines of code. Using a for loop , you can extract condition and substitution rules from the rules structure at the same time. On the first iteration of the for loop, match_rules becomes match_sxz and apply_rule becomes apply_sxz . During the second iteration, if we get to it, matches_rule will be assigned match_h and apply_rule will become apply_h . The function is guaranteed to return something when finished, because the last matching rule (match_default) simply returns True , implying that the corresponding replacement rule (

apply_default) will always be applied.

The reason this example works is because in Python everything is an object, even functions. The rules data structure contains functions - not function names, but actual function objects. When assigned in a for loop, matches_rule and apply_rule are real functions that you can call. On the first iteration of the for loop, this is equivalent to calling matches_sxz (noun) , and if it returns a match, calling apply_sxz (noun) .

-> The variable " rules " is a sequence of function pairs. [wa p- rob i n . with om]

If this extra layer of abstraction confuses you, try expanding the function to see that we get the same thing. The entire for loop is equivalent to the following :

def plural (noun) : if match_sxz (noun) : return apply_sxz (noun) if match_h (noun) : return apply_h (noun) if match_y (noun) : return apply_y (noun) if match_default (noun) : return apply_default (noun)

The advantage here is that the plural () function is simplified. It takes a sequence of rules defined elsewhere and follows them.

1. Get match rule
2. Does the rule work? Then apply the replacement rule and return the result.
3. No matches? Start from point 1.

Rules can be defined anywhere, in any way. There is absolutely no difference for the plural () function .

So adding this layer of abstraction was worth it? Not really yet. Let's try to imagine what it will take to add a new rule to a function. In the first example, this would require adding a new if construct to the plural () function . In the

second example, this would require adding two functions, match_foo () and apply_foo () , and then updating the rules sequence to indicate when the new match and replace rules should be called in relation to the rest of the rules.

But in reality it is only a means to move on to the next chapter. Moving on ...

List of templates

It is not necessary to define separate named functions for each condition and replacement rule. You never call them directly; you add them to the rules sequence and call them through it. Moreover, each function follows one of two patterns. All match functions call re . search () , and all replacement functions call re . sub () . Let's exclude templates to make it easier to declare new rules.

import re def build_match_and_apply_functions (pattern , search , replace) : def matches_rule (word) : ① return re . search (pattern , word) def apply_rule (word) : ② return re . sub (search , replace , word) return (matches_rule , apply_rule) ③

1. build_match_and_apply_functions () is a function that dynamically creates other functions. It takes pattern , search, and replace , then defines a matches_rule () function that calls re . search () with pattern , passed to build_match_and_apply_functions () as an argument, and word , which is passed to the matches_rule () function that you define.
2. We build the apply function in the same way. The apply function is a function that takes one parameter and calls re . sub () with search and replace parameters passed to the build_match_and_apply_functions function and word passed to the apply_rule () function you are creating. The approach of using the values of external parameters within a dynamic function is called closures. Basically, you define constants in a replace function: it

takes one parameter (word), but then acts using that and two other values (search and replace) that were set when the replace function was defined.

3. In the end, the build_match_and_apply_functions () function returns a tuple with two values, the two functions you just created. The constants you defined inside those functions (pattern inside the match_rule () function), search and replace in the apply_rule () function) stay with those functions, even. This is insanely cool.

If this confuses you (and it should be, this is very strange behavior), the picture may become clearer when you see how to use this approach.

patterns = \ ① (('[sxz] $' , '$' , 'es') , (' [^ aeioudgkprt] h $ ' , ' $ ' , ' es') , ('(qu | [^ aeiou]) y $ ' , ' y $ ' , ' ies') , ('$' , '$' , 's') ②) rules = [build_match_and_apply_functions (pattern , search , replace) ③ for (pattern , search , replace) in patterns]

1. Our plural rules are now defined as a tuple of string (not function) tuples. The first line in each group is the regex you would use in re . search () to determine if a given rule matches. The second and third lines in each group are the search and replace expressions you would use in re . sub () to apply the rule and convert the noun to plural.

2. There is a slight change in the alternate rule. In the previous example, the match_default () function simply returns True , implying that if no particular rule applies, the code should simply add s to the end of the given word. Functionally, this example does the same. The final regex will know if the word ends ($ searches for the end of the line). Of course, every line has an

end, even an empty one, so the expression always works. Thus, it serves the same purpose as the match_default () function , which always returned True : it ensures that if no other specific rules are executed, the code appends s to the end of the given word.

3. This is a magic line. It takes a sequence of strings in patterns and turns them into a sequence of functions. How? By "mapping" lines to the build_and_apply_functions () function . That is, it takes every three lines and calls the build_match_and_apply_functions () function with those three lines as arguments. The build_match_and_apply_functions () function returns a tuple of two functions. This means that rules eventually becomes functionally equivalent to the previous example: a list of tuples, where each tuple is a pair of functions. The first function is the match function that calls re . search () , and the second function is to apply the rule (replace), which calls re . sub () .

Let's end this version of the script with the main entry point, the plural () function .

def plural (noun) : for matches_rule , apply_rule in rules: ① if matches_rule (noun) : return apply_rule (noun)

1. Since the rules list is the same as in the previous example (yes, it is), it is not surprising that the plural () function has not changed at all. It is completely generalized; it takes a list of rule functions and calls them in order. She doesn't care how the rules are defined. In the previous example, they were defined as separate named functions. Now they are created dynamically by matching the result of the build_match_and_apply_functions () function to a list of regular strings. It doesn't matter. the plural () function continues to work as before.

Template file

You have taken out all the duplicate code and added enough abstraction to be able to store your plural rules in a list of strings. The next logical step is to take these lines and put them in a separate file, where they can be maintained separately from the code that uses them.

First, let's create a text file that contains the rules we need. No complicated data structures, just three-column data. Let's call it plural4-rules.txt

[sxz] $ $ es [^ aeioudgkprt] h $ $ es [^ aeiou] y $ y $ ies $ $ s

Now let's see how you can use this rules file.

import re def build_match_and_apply_functions (pattern , search , replace) : ① def matches_rule (word) : return re . search (pattern , word) def apply_rule (word) : return re . sub (search , replace , word) return (matches_rule , apply_rule) rules = [] with open ('plural4-rules.txt' , encoding = 'utf-8') as pattern_file: ② for line in pattern_file: ③ pattern , search , replace = line. split (None , 3) ④ rules. append (build_match_and_apply_functions (⑤ pattern , search , replace))

1. The build_match_and_apply_functions () function has not changed . You are still using closures to dynamically create two functions that will use the variables from the outer function.
2. The open () global function opens a file and returns a file object. In this case, the file we are opening contains template strings for the rules for forming the plural. The with statement creates what's called a context: when the with block ends, Python

will automatically close the file, even if an exception was thrown inside the with block . You will learn more about with blocks and file objects in the Files chapter.

3. The " for line in < fileobject > " form reads data from an open file line by line and assigns the text to the variable line . You will learn more about reading files in the Files chapter.

4. Each line in the file does contain three values, but they are separated by white space (tabs or spaces, no difference). To split them, use the split () string method . The first argument to split () is None , which means "split with any white space character (tab or space, no difference)." The second argument is 3, which means "split with free space 3 times, then leave the rest of the string". A string like " [sxz] $ $ es " will be split and converted to a list ['[sxz] $' , '$' , 'es '] , which means that the pattern will be ' [sxz] $ ' , search - ' $ ' , and replace will be set to ' es' . It's pretty powerful for one small line of code

5. Finally, you pass pattern , search and replace to build_match_and_apply_function () , which returns a tuple of functions. You add this tuple to the rules list , and at the end of the rules, it contains the list of matching and substitution functions that the plural () function expects .

The improvement made is that you have completely moved the rules out to an external file so that it can be maintained separately from the code that uses it. Code is code, data is data, and life is good.

Generators

But wouldn't it be cool if the generalized function plural () parses the file with rules? Extract the rules, find matches, apply the appropriate changes, move on to the next rule. This is all the plural () function has to do, and nothing else is required of it.

def rules (rules_filename) : with open (rules_filename , encoding = 'utf-8') as pattern_file: for line in pattern_file: pattern , search , replace = line. split (None , 3) yield build_match_and_apply_functions (pattern , search , replace) def plural (noun , rules_filename = 'plural5-rules.txt') : for matches_rule , apply_rule in rules (rules_filename) : if matches_rule (noun) : return apply_rule (noun) raise ValueError ('no matching rule for {0}' . format (

noun))

How the hell does it work? Let's first look at an example with explanations.

>>> def make_counter (x) : ... print ('entering make_counter') ... while True : ... yield x ① ... print ('incrementing x') ... x = x + 1 . .. >>> counter = make_counter (2) ② >>> counter ③ < generator object at 0x001C9C10 > >>> next (counter) ④ entering make_counter 2 >>> next (counter) ⑤ incrementing x 3 >>> next (counter) ⑥ incrementing x 4

1. The presence of the yield keyword in make_counter means that this is not a regular function. This is a special kind of function that generates values one at a time. You can think of it as an ongoing function. Calling it will return a generator that can be used to generate subsequent x values .
2. To instantiate the make_counter generator , simply call it like any other function. Note that this does not actually execute the function code. You can say that because the first line of make_counter () calls print () , but nothing has been printed so far.
3. Make_counter () function returns a generator object.
4. The next () function takes a generator and returns its next value. The first time you call next () with the counter generator , it executes the code in make_counter () until the first yield statement , then returns the value that was returned by yield . In this case, it will be 2, since you originally created the generator by calling make_counter (2) .
5. Calling next () again with the same generator continues the computation exactly where it left off and continues until it encounters the next yield . All variables, local states , etc. are saved during yield , and restored when next () is called. The next line of code awaiting execution calls print () , which prints incrementing x . This is followed by the statement x = x + 1 . Then the while loop is executed again , and the first thing that occurs in it is the yield x statement , which saves all state and returns the current value of x (now it is 3).
6. After the second call to next (counter) , everything is the same, only now x becomes equal to 4.

Since make_counter goes into an infinite loop, you could theoretically do it endlessly, and it would keep incrementing x and returning its values. But instead, let's look at more productive uses of generators.

Fibonacci sequence generator

def fib (max) : a , b = 0 , 1 ① while a < max : yield a ② a , b = b , a + b ③

1. A Fibonacci sequence is a sequence of numbers in which each number is the sum of the previous two. It starts from 0 and 1, at first it gradually increases, then it grows faster and faster. To start a sequence, you need two variables: a starts at 0 and b starts at 1.
2. a is the initial value of the sequence, so it should be returned.
3. b is the next number in the sequence, so assign it to a , but also count the next value (a + b) and assign it to b for later use. Note that this happens at the same time. If a is 3 and b is 5, then a , b = b , a + b will set a to 5 (previous value of b) and b to 8 (sum of previous values of a and b).

So now you have a function that outputs consecutive Fibonacci numbers. Of course, you could do it with recursion, but this implementation is easier to read. It also works better with for loops .

-> yield pauses the function, next () runs it again in the same state

>>> from fibonacci import fib >>> for n in fib (1000) : ① ... print (n , end = ") ② 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 > >> list (fib (1000)) ③ [0 , 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 , 34 , 55 , 89 , 144 , 233 , 377 , 610 , 987]

1. You can use a generator like fib () directly in a for loop. The for loop automatically calls next () to get the values from the fib () generator and assign them to the for loop variable n .
2. Each time the for loop goes through , n takes on a new value from yield in fib () , and all you have to do is print it. Once fib () goes beyond the numbers (a becomes bigger than max , which in this case is 1000), so once the cycle for finishing work.

3. This is a useful technique: give the generator to list () and it will loop through the entire generator (just like the for loop in the previous example) and return a list of all values.

A Plural Rule Generator

Let's go back to plural5.py and see how this version of the plural () function works .

def rules (rules_filename) : with open (rules_filename , encoding = 'utf-8') as pattern_file: for line in pattern_file: pattern , search , replace = line. split (None , 3) ① yield build_match_and_apply_functions (pattern , search , replace) ② def plural (noun , rules_filename = 'plural5-rules.txt') : for matches_rule , apply_rule in rules (rules_filename) : ③ if matches_rule (noun) : return apply_rule (noun) raise ValueError ('no matching rule for {0}' . format (noun))

1. No magic. Remember that the lines in the rules file each contain three values separated by white space, so you are using line. split (None , 3) to get three "columns" and assign them to three local variables.
2. And then you call yield . What are you returning? Two functions created dynamically by your old helper, build_match_and_apply_functions () , which is the same as in the previous examples. In other words, rules () is a generator that returns match and change rules on demand.
3. Since rules () is a generator, you can use it directly in a for loop. The first time you go through the for loop, you call the rules () function , which will open the templates file, read the first line,

dynamically build the condition and modify function from the template on that line, and return the dynamically generated functions. When you go through the loop a second time, you will continue exactly where you left rules () (this was inside the for line in pattern_file loop). The first thing it will do is read the next line of the file (which is still open), dynamically create other condition and modification functions based on the templates of that line of the file, and serve these two functions.

What have you purchased compared to option 4? Less startup time. In option 4, when you imported the plural4 module , it read the entire template file and built a list of all possible rules before you even thought about calling plural () . With generators, you can do everything lazily: you read the first rule and create functions and try them, and if that works, you never read the rest of the file and create other functions.

Where are you missing? In productivity! Each time you call the plural () function , the rules () generator starts over from the beginning - which means reopening the template file and reading from the beginning line by line.

What if you could get the best of both worlds: minimal startup overhead (don't execute any code during import) and maximum performance (don't create the same functions over and over). And yes, you still want to keep the rules in a separate file (because code is code and data is data), until you suddenly need to read the same line twice.

To do this, you will need to build your own iterator. But before you do that, you need to learn about Python classes.

Further reading

- PEP 255: Simple Generators
- Understanding Python's "with" statement
- Closures in Python
- Fibonacci numbers
- English Irregular Plural Nouns

Classes and iterators

Immersion

Generators are really just a special case of iterators. A function that returns (yields) values is a simple and compact way to get the functionality of an iterator without *creating the* iterator directly . Remember the Fibonacci number generator? Here's a sketch of what a similar iterator might look like:

class Fib: " 'iterator that yields numbers in the Fibonacci sequence' " def __init__ (self , max) : self . max = max def __iter__ (self) : self . a = 0 self . b = 1 return self def __next__ (self) : fib = self . a if fib > self . max : raise StopIteration self . a , self . b = self . b , self . a + self . b return fib

Let's take a closer look at this example:

class Fib:

class ? What is a class?

Defining classes

Python is completely object-oriented, meaning you can define your own classes, inherit new classes from your own or built-in classes, and instantiate the classes you have already defined.

Defining a class in Python is easy. As with functions, separate interface

declarations are not required. You just define the class and start programming. A class definition in Python begins with the reserved word class , followed by the name (identifier) of the class. Formally, this is all that is needed in the case when the class should not be inherited from another class.

```
class PapayaWhip:[K 1]
    pass[K 2]
```

1. ↑ The above class is named PapayaWhip and does not inherit from any other class. Class names are usually capitalized, for example like this , but this is just a convention, not a requirement.
2. ↑ You probably already guessed that every line in the class definition is indented, just like with functions, the if statement , the for loop, or any other block of code. The first non-indented line is outside the class block .

The PapayaWhip class does not contain method or attribute definitions, but from a syntax point of view, the class body cannot be empty. In such cases, the pass statement is used . In Python, pass is a reserved word that tells the interpreter, "go ahead, there is nothing here." This is a statement that does absolutely nothing, but nevertheless is a convenient solution when you need to stub a function or class.

> The pass expression in Python is analogous to the empty set or curly braces in Java or C ++ .

Many classes inherit from other classes, but not this one. Many classes define their own methods, but not this one. A Python class doesn't have to have anything but a name. In particular, it may seem strange to people familiar with C ++ that a class in Python does not have a constructor and a destructor explicitly. Although not required, a Python class can have something similar to a constructor: the __init__ () method .

The __init __ () method

The following example demonstrates the initialization of the Fib class using the __init__ () method .

```
class Fib: " 'iterator that yields numbers in the Fibonacci sequence' "[K 1]
```

[K 2]

```
def __init__ ( self , max ) :
```

1. ↑ Classes, by analogy with modules and functions, can (and should) have docstrings.
2. ↑ The __init__ () method is called immediately after the class is instantiated. It would be tempting, but formally incorrect, to think of it as the "constructor" of the class. It's tempting because it resembles a C ++ class constructor : *externally* (it is generally accepted that the __init__ () method should be the first method defined on the class), and *in action* (this is the first block of code executed in the context of the newly created class instance). Incorrect, because at the time of calling __init__ () the object is actually already created, and you can operate with a correct reference to it (self)

The first argument to any class method, including the __init__ () method , is always a reference to the current instance of the class. It is customary to call this argument self . This argument acts as a reserved word this in C ++ or Java , but nonetheless self is not a reserved word in Python . Although this is just a convention, please do not call this argument anything else.

In the case of the __init__ () method , self refers to the newly created object; in other methods - on the instance, the method of which was called. And, although you need to explicitly specify self when defining a method, you do not need to do so when calling; Python will add it for you automatically.

Instantiation

To create a new instance of a class in Python, you call the class as if it were a function, passing the required arguments to the __init __ () method . We will receive the newly created object as a return value.

```
>>> import fibonacci2 >>> fib = fibonacci2. Fib ( 100 )
                                                                    1
>>> fib                                                             2
< fibonacci2. Fib object at 0x00DB8810 > >>> fib .__ class__
                                                                    3
< class 'fibonacci2.Fib' > >>> fib .__ doc__
                                                                    4
```

'iterator that yields numbers in the Fibonacci sequence'

1. You create a new instance of the Fib class (defined in the fibonacci2 module) and assign the newly created object to the fib variable . The only passed argument, 100, corresponds to the named argument max , in the __init __ () method of the Fib class .
2. fib is now an instance of the Fib
3. Each instance of the class has a built-in __class__ attribute that points to the class of the object. Java programmers may be familiar with the Class class , which contains the getName () and getSuperclass () methods used to get information about an object. In Python, this kind of metadata is available through the appropriate attributes, but the idea behind it is the same.
4. You can get the docstring of a class, similar to a function and a module. All instances of the class share the same docstring.

> To create a new instance of a class in Python, simply call the class as if it were a function; there are no explicit operators like new in C ++ or Java in Python.

Instance variables

Let's move on to the next line:

class Fib: def __init__ (self , max) : self . max = max

1 .

1. What is self . max ? This is an instance variable. It has nothing to do with the max variable that we passed to the __init__ () method as an argument. self . max is "global" for the entire instance. This means that you can refer to it from other methods.

class Fib: def __init__ (self , max) : self . max = max

1 .

...
...

... def __next__ (self) : fib = self . a if fib > self . max :

2

.

1. self . max is defined in the __init__ () method ...
2. ... and used in the __next__ () method .

Instance variables are associated with only one instance of the class. For example, if you create two instances of the Fib class with different maximum values, each will only remember its own value.

>>> import fibonacci2 >>> fib1 = fibonacci2. Fib (100) >>> fib2 = fibonacci2. Fib (200) >>> fib1. max 100 >>> fib2. max 200

Fibonacci number iterator

You are now ready to learn how to create an iterator. An iterator is a regular class that defines an __iter__ () method .

class Fib: ① def __init__ (self , max) : ② self . max = max def __iter__ (self) : ③ self . a = 0 self . b = 1 return self def __next__ (self) : ④ fib = self . a if fib > self . max : raise StopIteration ⑤ self . a , self . b = self . b , self . a + self . b return fib ⑥

① To build an iterator from scratch, Fib must be a class, not a function.

More on iterators

Immersion

HAWAII + IDAHO + IOWA + OHIO == STATES . Or, to put it another way, 510199 + 98153 + 9301 + 3593 == 621246. Do you think I'm delusional? No, it's just a puzzle.

Let me give you a clue.

HAWAII + IDAHO + IOWA + OHIO == STATES
510199 + 98153 + 9301 + 3593 == 621246

H = 5
A = 1
W = 0
I = 9
D = 8
O = 3
S = 6
T = 2
E = 4

These puzzles are called cryptarithms . Letters make up existing words, and if you replace each letter with a number from 0 to 9, you also get the correct mathematical equality. The trick is to figure out which letter corresponds to each number. All occurrences of each letter must be replaced by the same digit, several letters cannot correspond to one digit and “words” cannot begin with the digit 0.

In this chapter, we'll take a look at an amazing Python program written by

Raymond Heittinger. This program solves crypto-rithmic puzzles and consists of only 14 lines of code.

The most famous cryptarithmic puzzle SEND + MORE = MONEY.

```
import re import itertools def solve ( puzzle ) : words = re . findall ( '[AZ] +' , puzzle. upper ()) unique_characters = set ( '' . join ( words )) assert len ( unique_characters ) <= 10 , 'Too many letters' first_letters = { word [ 0 ] for word in words } n = len ( first_letters ) sorted_characters = '' . join ( first_letters ) + \ '' . join ( unique_characters - first_letters ) characters = tuple ( ord ( c ) for c in sorted_characters ) digits = tuple ( ord ( c ) for c in '0123456789' ) zero = digits [ 0 ] for guess in itertools . permutations ( digits , len ( characters )) : if zero not in guess [ : n ] : equation = puzzle. translate ( dict ( zip ( characters , guess ))) if eval ( equation ) : return equation if __name__ == '__main__' : import sys for puzzle in sys . argv [ 1 : ] : print ( puzzle ) solution = solve ( puzzle ) if solution: print ( solution )
```

You can run the program from the command line. On Linux it will look like this. (Execution of the program may take some time depending on the speed of your computer, and the progress bar is not in the program. So just please be patient .)

you @ localhost: ~ / diveintopython3 / examples $ python3 alphametics.py "HAWAII + IDAHO + IOWA + OHIO == STATES"
HAWAII + IDAHO + IOWA + OHIO = STATES
510199 + 98153 + 9301 + 3593 == 621246
you @ localhost: ~ / diveintopython3 / examples $ python3 alphametics.py "I + LOVE + YOU == DORA"
I + LOVE + YOU == DORA
1 + 2784 + 975 == 3760
you @ localhost: ~ / diveintopython3 / examples $ python3 alphametics.py "SEND + MORE == MONEY "
SEND + MORE == MONEY
9567 + 1085 == 10652

Finding All Occurrences of a Pattern

The first thing this program does is find all the words (in the original - letters, since it is looking for letters, approx. Transl.) In the puzzle.

>>> import re >>> re . findall ('[0-9] +' , '16 2-by-4s in rows of 8 ') ① [' 16 ' , ' 2 ' , ' 4 ' , ' 8 '] >>> re . findall ('[AZ] +' , 'SEND + MORE == MONEY') ② ['SEND' , 'MORE' , 'MONEY']

① The re module implements regular expressions in Python. This module has a handy findall () function that takes a regular expression pattern and a string as parameters, and finds all substrings that match the pattern. In this case, the pattern matches sequences of numbers. Findall () returns a list of all substrings that represent a sequence of numbers.

② Here the regular expression matches sequences of letters. Again, the return value is a list, each element of which is a string that matches a regular expression pattern.

This is the most difficult tongue twister in English.

Here's another example that you might have to puzzle over.

>>> re . findall ('s. *? s' , "The sixth sick sheikh's sixth sheep's sick.") ['sixth s' , "sheikh's s" , "sheep's s"]

Are you surprised? The regexp looks for a space, followed by the letter s, the shortest possible sequence of any characters (. *?), A space, and another letter s.

1. The **sixth s** ick sheikh's sixth sheep's sick.
2. The sixth **sick s** heikh’s sixth sheep's sick.
3. The sixth sick **sheikh’s s** ixth sheep's sick.
4. The sixth sick sheikh’s **sixth s** heep’s sick.
5. The sixth sick sheikh's sixth **sheep’s s** ick.

But the re.findall () function only finds three occurrences: the first, third, and fifth. Why is that? Because it doesn't return overlapping substrings. The first substring overlaps with the second, so only the first string is returned, and the second is skipped. Then the third substring overlaps with the fourth, so only the third substring is returned and the fourth is skipped. Finally the fifth substring is returned. Three matches, not five.

This has nothing to do with solving cryptarithms, I just thought it was interesting.

Finding Unique Items in Sequences

Sets make the task of finding unique elements in a sequence trivial.

>>> a_list = ['The' , 'sixth' , 'sick' , "sheik's" , 'sixth' , "sheep's" , 'sick'] >>> set (a_list) ① { 'sixth' , 'The' , "sheep's" , 'sick' , "sheik's" } >>> a_string = 'EAST IS EAST' >>> set (a_string) ② { 'A' , ' ' , 'E' , 'I' , 'S' , 'T' } >>> words = ['SEND' , 'MORE' , 'MONEY'] >>> '' . join (words) ③ 'SENDMOREMONEY' >>> set ('' . join (words)) ④ { 'E' , 'D' , 'M' , 'O' , 'N' , 'S' , 'R' , 'Y' }

① Given a list of multiple lines, the set () function will return many unique lines from this list. How this function works is clearer if you provide a for loop. Take the first item from the list and add it to the set. Second. Third. Fourth. Fifth - wait a minute, it's already in the set, so you don't need to add it, because Python sets don't allow you to have duplicate elements. Sixth. Seventh - duplicate again, skip it. What's the result? All unique elements of the original list without duplicates. You don't even need to sort the original list beforehand.

② The same approach works if the set () function is passed a string rather than a list, since a string is just a sequence of characters.

③ *Given a* list of strings, the *.join (a_list)* function *concatenates all these strings into one.*

④ This code snippet, after receiving a list of strings, returns all unique characters found in all strings.

Our cryptarithm solver uses this technique to get the set of all the unique letters used in the puzzle.

unique_characters = set (" . join (words))

Checking the fulfillment of conditions

Like many programming languages, Python has an assert statement. Here's how it works.

Like many programming languages, Python has a no-error confirmation statement. This is how it works .

>>> assert 1 + 1 == 2 ① >>> assert 1 + 1 == 3 ② Traceback (most recent call last) : File "<stdin>" , line 1 , in < module > AssertionError >>> assert 2 + 2 == 5 , "Only for very large values of 2" ③ Traceback (most recent call last) : File "<stdin>" , line 1 , in < module > AssertionError : Only for very large values of 2

① The word assert is followed by any valid Python expression. In this case, the expression 1 + 1 == 2 returns True, so assert does nothing.

② However, if the expression returns False, assert throws an exception.

③ You can also add an informative message to be displayed when an AssertionError is raised.

Therefore , this line of code

assert len (unique_characters) <= 10 , 'Too many letters'

equivalent to

if len (unique_characters) > 10 : raise AssertionError ('Too many letters')

The cryptarithm solver uses this expression to stop execution if the puzzle

contains more than ten unique letters. Since each letter corresponds to a number, and there are only ten numbers, a puzzle with more than ten unique letters cannot have a solution.

Generator expressions

Generator expressions, like a function generator, but no function.

>>> unique_characters = { 'E' , 'D' , 'M' , 'O' , 'N' , 'S' , 'R' , 'Y' } >>> gen = (ord (c) for c in unique_characters) ① >>> gen ② < generator object < genexpr > at 0x00BADC10 > >>> next (gen) ③ 69 >>> next (gen) 68 >>> tuple (ord (c) for c in unique_characters) ④ (69 , 68 , 77 , 79 , 78 , 83 , 82 , 89)

① A generator expression is like an anonymous function that outputs values. The expression itself looks like a list, but it is not wrapped in square braces, but in curly braces.

② The generator expression returns an iterator.

③ Calling next (gen) returns the next value of the iterator.

④ If you like, you can iterate through all possible values and return a tuple, list, or set by passing a generator expression to tuple () , list () , or set () . In these cases, you don't need extra parentheses - just pass the bare expression ord (c) for c in unique_characters to the tuple () function, and Python understands that it is a generator expression.

> Using generator expressions instead of a list helps save cpu and ram . If you use a list to throw it away later (for example, pass it to a tuple () or set ()), use a generator instead!

Here's another way to do the same using a function generator:

def ord_map (a_string) : for c in a_string: yield ord (c) gen = ord_map (

unique_characters)

Generator expressions are more compact, but functionally equal.

Testing

(not) Immersion

(This page is under translation)

Modern youth. Spoiled for fast computers and trendy "dynamic" languages. Write, then provide your code, then debug (at best). There was discipline these days. I said discipline! We had to write programs by hand, on paper, and enter them into a computer on punched cards. And we loved it!

In this section, you will write and debug a set of helper functions for converting to and from the Roman system. You saw how Roman numerals are constructed and validated in Case Study: Roman Numerals. Let's go back a little and imagine what the implementation would look like as a function that converts in both directions.

The rules for the formation of Roman numbers lead us to several interesting observations:

1. There is only one correct way to write a number in Roman numerals.

2. The converse is also true: if a character string is a sequence of Roman characters, it represents only one number, that is, it can be interpreted in a single way.

3. The range of numbers that can be written in Roman numerals is from 1 to 3999. The Romans had several ways to write larger numbers, in particular, using the bar above the number, which meant 6y, that the value must be multiplied by 1000. For purposes for this chapter, it is enough for us to restrict ourselves to the range 1 - 3999.

4. There is no way to represent 0 in the Roman system.

5. There is no way to represent negative numbers in the Roman system.

6. There is no way to represent fractional or non-integer numbers in the Roman system.

Let's try to reflect what the roman.py module should do. It will contain two main functions, to_roman () and from_roman (). The to_roman () function must accept an integer in the range 1 to 3999 and return a string containing the Roman representation of that number ...

Let's stop here. Let's now do something unexpected: let's describe a small test case that checks if the to_roman () function works as we expect it to. You read that right: we're going to write code that tests code that hasn't been written yet.

This is called test-driven-development (TDD). A set of two conversion functions - to_roman (), from_roman () - can be written and tested as a unit, separate from any large program that imports them. Python has a framework for unit testing, a module named unittest.

Unit testing is an important part of the entire test -driven development strategy. If you write unit tests, you need to write them early and update them as your code and requirements change. Many people advocate writing tests before writing testable code, and this is the approach I'm going to demonstrate in this section. However, unit tests are beneficial no matter when you write them.

- Before writing the code, writing unit tests forces you to detail the requirements in a form convenient for their implementation.
- As you write your code, unit tests protect you from unnecessary coding. When all tests pass, the unit under test is ready.
- During refactoring, they help prove that the new version behaves the same as the old one.
- During code maintenance, the existence of unit tests will get your butt off when someone yells that your last change broke their code. ("But sir, all tests passed when I made a commit.")
- When code is written as a team, having a comprehensive test suite greatly reduces the risk of your code breaking another developer's code, since you can run their unit tests. (I've seen how this works in

practice in code sprints (coding for speed? :) ???). The team breaks down the task, the participants parse the specifications of their tasks, write unit tests for them, then exchange unit tests with the entire team. This way, no one goes too far in developing code that is poorly suited for the team.)

The only question.

One test case answers one question about the code under test. The test case should be able to ...

- ... run on its own, without human input. Unit testing should be automated
- ... to determine independently whether the tested function passed the test or not, without human intervention in order to interpret the results
- ... run in isolation, separate from the rest of the test cases (even if they test the same functionality)

Each test case is an island.

With this in mind, let's create a test case (test) for the first requirement: 1. The to_roman () function must return the Roman numeral representation for all numbers from 1 to 3999

It's not immediately clear how this script does ... well, anything. It defines a class that does not contain an __init __ () method. The class contains another method that is never called. The script contains a __main__ block, but it does not refer to the class or its methods. But he does something, trust me.

import roman1
import unittest class KnownValues (unittest . TestCase) : ① known_values = ((1 , 'I') , (2 , 'II') , (3 , 'III') , (4 , 'IV') , (5 , 'V') , (6 , 'VI') , (7 , 'VII') , (8 , 'VIII') , (9 , 'IX') , (10 , 'X') , (50 , ' L ') , (100 , ' C ') , (500 , ' D ') , (1000 , ' M ') , (31 , ' XXXI ') , (148 , ' CXLVIII ') , (294 , ' CCXCIV ') , (312 , 'CCCXII') , (421 , 'CDXXI') , (528 , 'DXXVIII') , (621 , 'DCXXI') , (782 , 'DCCLXXXII') , (870 , 'DCCCLXX') , (941 , 'CMXLI') , (1043 , 'MXLIII') , (1110 , 'MCX') , (1226 , 'MCCXXVI') , (1301 , 'MCCCI') , (1485 , 'MCDLXXXV') , (1509 , 'MDIX') , (1607 , 'MDCVII') , (1754 , 'MDCCLIV') , (1832 , 'MDCCCXXXII') , (1993 , 'MCMXCIII') , (2074 , 'MMLXXIV') , (2152 , ' MMCLII ') , (2212 , ' MMCCXII ') , (

2343 , ' MMCCCXLIII ') , (2499 , ' MMCDXCIX ') , (2574 , ' MMDLXXIV ') , (2646 , ' MMDCXLVI ') , (2723 , ' MMDCCXXIII ') , (2892 , 'MMDCCCXCII') , (2975 , 'MMCMLXXV') , (3051 , 'MMMLI ') , (3185 , ' MMMCLXXXV ') , (3250 , ' MMMCCL ') , (3313 , ' MMMCCCXIII ') , (3408 , ' MMMCDVIII ') , (3501 , ' MMMDI ') , (3610 , ' MMMDCX ') , (3743 , 'MMMDCCXLIII') , (3844 , 'MMMDCCCXLIV') , (3888 , 'MMMDCCCLXXXVIII') , (3940 , 'MMMCMXL') , (3999 , 'MMMCMXCIX')) ② def test_to_roman_known_values (self) : ③ ' " to_roman should give known result with known input " ' for integer , numeral in self . known_values : result = roman1. to_roman (integer) ④ self . assertEqual (numeral , result) ⑤ if __name__ == '__main__' : unittest . main ()

① To describe the test case, the first step is to define the TestCase class as a subclass of the unittest module. This class contains many useful methods that you can use in your tests for specific conditions.

② These are many manually defined number / value pairs. It includes the minimum 10 numbers, the largest (3999), all numbers that are converted to one character, and a set of random numbers. You don't need to test every possible variation, but you should test every unique variation.

③ Each test is defined by a separate method that is called without parameters and does not return a value. If the method completes normally, without throwing an exception, the test is considered passed, if an exception is thrown, the test is failed.

④ This is where the to_roman () function being tested is called. (Well, the function hasn't been written yet, but when it does, it will be the string that will call it.) Notice that you just defined the API for the to_roman () function: it must take a number to convert and return a string (represented as Roman number). If the API is different from the above, the test will return an error. Also note that you are not catching any exceptions when you call to_roman (). This is done on purpose. to_roman () should not return an exception when called with the correct input parameters and the correct values for those parameters. If to_roman () throws an exception, the Test fails.

⑤ Assuming the to_roman () function is defined correctly, called correctly, succeeded, and returned a value, the last step is to verify that the returned value is correct. This is a common question, so we use the AssertEqual method of the TestCase class to check the equality (equivalence) of two values. If the result returned by to_roman () is not equal to the known value you expect (numeral), assertEqual will throw an exception and the test will fail. If the values are equivalent, assertEqual will do nothing. If every value returned by to_roman () matches the expected known value, assertEqual will never throw an exception, which means test_to_roman_known_values will eventually execute fine, which means that the to_roman () function passed the test.

Once you have a test, you can write the to_roman () function itself. First, you need to write a stub, an empty function and make sure the test fails. If the test succeeds, when the function is not doing anything yet, then the test is not working at all! Unit testing is like a dance: the test leads, the code follows. You write the test that fails, then the code until the test passes.

roman1.py
def to_roman (n) :
" 'convert integer to Roman numeral' "
pass ①

① At this point, you are defining the API for the to_roman () function, but you don’t want to code it yet. (For the first test of the test.) To stub a function, use the Python reserved word pass, which ... does nothing. Run romantest1.py on in the interpreter to test the test. If you invoke the script with the -v parameter, details about the script's operation (verbose) will be displayed, and you can see in detail what happens in each test. If you 're lucky , you'll see something like this :

you @ localhost: ~ / diveintopython3 / examples $ python3 romantest1. py -v test_to_roman_known_values (__main__ . KnownValues) ① to_roman should give known result with known input ... FAIL ② ========== ========== FAIL: to_roman should give known result with known input ---------- ------- ---------- Traceback (most recent call last) : File "romantest1.py" , line 73 , in test_to_roman_known_values self . assertEqual (numeral , result) AssertionError : 'I' ! = None ③ ---------- ---- ---------- Ran 1 test in 0.016s ④ FAILED (failures = 1) ⑤

① Launched script performs a method of unittest . main (), which runs each test case . Each test case is a class method in romantest.py. There are no special requirements for the organization of these classes; they can be as a class with a method for a single test case, mfr and one class + multiple methods for all test cases. You just need each class to inherit from unittest.TestCase.

② For each test case, unittest will print the docstring of the method and the result, success or failure. As you can see, the test has failed.

③ For each failed test, the system displays detailed information about what exactly happened. In this case, the call to assertEqual () raised an AssertionError because to_roman (1) was expected to return 'I', but it did not. (If the function doesn't have an explicit return, then it will return None, which is null in Python.)

④ After detailing each test case, unittest displays a summary of how many tests were run and how long it took.

⑤ In general, the test is considered failed if at least one case fails. Unittest distinguishes between errors and failures. Failure calls an assertXYZ method such as assertEqual or assertRaises, which will fail if the declared condition is invalid or the expected exception is not thrown. An error is another type of exception that is thrown by the code under test or a test unit and is not expected.

Finally, we can write the to_roman () function.

roman_numeral_map = (('M' , 1000) , ('CM' , 900) , ('D' , 500) , ('CD' , 400) , ('C' , 100) , ('XC' , 90) , ('L' , 50) , ('XL' , 40) , ('X' , 10) , ('IX' , 9) , ('V' , 5) , ('IV' , 4) , ('I' , 1)) ① def to_roman (n) : " 'convert integer to Roman numeral' " result = " for numeral , integer in roman_numeral_map: while n > = integer: ② result + = numeral n - = integer return result

① roman_numeral_map is a tuple of tuples that defines three things: a representation of the basic characters of Roman numerals and popular combinations of them; the order of Roman characters (backwards, from M to I); the meaning of Roman numerals. Each inner tuple is a pair of values (representation, number). And it's not just single-character Roman numerals; they are also pairs of characters like CM (“one thousand without a hundred”). This greatly simplifies the to_roman () function code.

② Here you can see what the advantage of such a roman_numeral_map structure is, since no tricky logic is required in subtraction processing. To convert to a Roman number, you just need to loop through the roman_numeral_map loop, finding the smallest number that fits the rest of the input. If one is found, the corresponding Roman representation is added to the return value of the function, the input is decreased by this number, and then the operation is repeated for the next tuple.

If you still don't understand how the to_roman () function works, add print () to the end of the loop:

while n > = integer: result + = numeral n - = integer print ('subtracting {0} from input, adding {1} to output' . format (integer , numeral))

This debug output shows the following:

>>> import roman1 >>> roman1. to_roman (1424) subtracting 1000 from input , adding M to output subtracting 400 from input , adding CD to output subtracting 10 from input , adding X to output subtracting 10 from input , adding X to output subtracting 4 from input , adding IV to output ' MCDXXIV '

Well, the to_roman () function seems to work as suggested at the beginning of the chapter. But it will pass if it is written before the test ?

you @ localhost: ~ / diveintopython3 / examples $ python3 romantest1. py -v
test_to_roman_known_values (__main__ . KnownValues)
to_roman should give known result with known input ... ok
---------- ----------
Ran 1 test in 0.016s OK

1. Hurray ! The to_roman () function passed the “known values” test . It may not be a comprehensive check, but it did validate a variety of inputs, including numbers written in a single Roman character, the largest original value (3999), and the value that yields the largest Roman number (3888). At this point, we can say that the function will correctly process any valid input values.

"Correct" initial values? Hmm. What about the wrong ones?

"Stop and light up"

It is not enough to test a function with just the correct input; you also need to make sure that the function will generate an error if you enter it incorrectly. And not just a mistake - but such as expected .

>>> import roman1 >>> roman1. to_roman (4000) 'MMMM' >>> roman1. to_roman (5000) 'MMMMM' >>> roman1. to_roman (9000) ① 'MMMMMMMMM'

1. This is definitely not what was expected - these are not correct Roman numbers! In fact, all these numbers are out of range, but the function still returns a result, only a dummy one. Quiet return incorrect values - ooooooooooochen wrong; if an error occurs, it is best that the program exits quickly and noisily. "Stop and light up," as they say. The "Python" way to stop and catch fire is to throw an exception.

The question is, how can this be taken into account in the testing requirements? For beginners, like this: the to_roman () function should throw an OutOfRangeError if passed a number greater than 3999. What will the test look like ?

class ToRomanBadInput (unittest . TestCase) : ① def test_too_large (self) : ② " 'to_roman should fail with large input' " self . assertRaises (roman2. OutOfRangeError , roman2. to_roman , 4000) ③

1. As in the previous case, create a subclass from unittest.TestCase. You may have more than one test per class (as you will see later in this chapter), but I decided to create a separate class for this because this case is different from the previous ones. We've put all tests for "positive" in one class and for errors in another.

2. As in the previous case, a test is a method whose name is the name of the test.

3. The unittest.TestCase class provides an assertRaises method that takes the following arguments: the type of exception expected, the name of the function being tested, and the arguments for that function. (If the function

under test takes more than one argument, they are all passed to the assertRaises method in order, as if you were passing them to the function under test.)

Pay close attention to the last line of code. Instead of calling to_roman () and manually checking that it is throwing an exception (by wrapping it in a try-catch block), the assertRaises method does it all for us. All you do is say what type of exception you are expecting (roman2.OutOfRangeError), the name of the function (to_roman ()), and its arguments (4000). The assertRaises method will take care of the to_roman () function call and check that it returns a roman2.OutOfRangeError exception.

Also notice that you are passing to_roman () as an argument; You don't call it and pass its name as a string. I think I mentioned that everything in Python is an object?

What happens when you run a script with a new test?

you @ localhost: ~ / diveintopython3 / examples $ python3 romantest2. py -v test_to_roman_known_values (__main__ . KnownValues) to_roman should give known result with known input ... ok test_too_large (__main__ . ToRomanBadInput) to_roman should fail with large input ... ERROR ① ========== ========== ======== ERROR: to_roman should fail with large input ---------- ---------- ---------- Traceback (most recent call last) : File "romantest2.py" , line 78 , in test_too_large self . assertRaises (roman2. OutOfRangeError , roman2. to_roman , 4000) AttributeError : 'module' object has no attribute 'OutOfRangeError' ② ---------- ---------- ---------- Ran 2 tests in 0.000 s FAILED (errors = 1)

1. This failure should be expected (unless of course you have written additional code), but ... it's not really a "failure", but rather a mistake. This is a subtle but very important distinction. The test can return three states: success, failure, and error. Success, of course, means that the test is passed - the code does what it should. "Failure" is what the test returned above - the code is executed, but it does not return the expected value. "Error" means that your code is not working correctly.

2. Why is the code not executing correctly? Unrolling the stack explains everything. The unit under test does not throw an OutOfRangeError exception. The same one that we fed to the assertRaises () method, because we expect it when entering a large number. But no exception is thrown, so the call to the assertRaises () method failed. No chance - the to_roman () function will never throw an OutOfRangeError.

Let's solve this problem by defining the OutOfRangeError exception class in roman2.py.

class OutOfRangeError (ValueError) : ① pass ②

1. Exceptions are classes. An "out of range" error is a form of error — the argument is out of range . Therefore, this exception inherits from the ValueError exception. This is not strictly necessary (in theory, inheriting from the Exception class is sufficient), but it is correct.

2. Exception doesn't do anything, but you need at least one line in the class. The built-in pass function does nothing, but is required for minimal code definition in Python.

Now let's run the test again.

you @ localhost: ~ / diveintopython3 / examples $ python3 romantest2. py -v
test_to_roman_known_values (__main__ . KnownValues)
to_roman should give known result with known input ... ok
test_too_large (__main__ . ToRomanBadInput)
to_roman should fail with large input ... FAIL ① ==========
==
======= FAIL: to_roman should fail with large input ----------
--------- ---------- Traceback (most recent call last) :

File "romantest2.py" , line 78 , in test_too_large self . assertRaises (roman2. OutOfRangeError , roman2. to_roman , 4000) AssertionError : OutOfRangeError not raised by to_roman ② ---------- ---------- Ran 2 tests in 0.016s FAILED (failures = 1)

1. The test still fails, although it no longer throws an error. This is progress! This means that the assertRaises () method was executed and the to_roman () function test was performed.

2. Of course, the to_roman () function does not throw the newly defined OutOfRangeError, as you haven't "forced" it yet. And that's good news! This means that the test works, but it will fail until you write the condition for its successful passage.

This and Let us .

def to_roman (n) : " 'convert integer to Roman numeral' " if n > 3999 : raise OutOfRangeError ('number out of range (must be less than 4000)') ① result = " for numeral , integer in roman_numeral_map : while n > = integer: result + = numeral n - = integer return result

1. Everything is simple: if the passed parameter is greater than 3999, throw an OutOfRangeError exception. The test does not look for a text string explaining the reason for the exception, although you can write a test to test this (but be aware of the difficulties associated with different languages - the length of the lines or the environment may differ).

Will this allow the test to pass? Find out :

you @ localhost: ~ / diveintopython3 / examples $ python3 romantest2. py -v test_to_roman_known_values (__main__ . KnownValues) to_roman should give known result with known input ... ok test_too_large (__main__ . ToRomanBadInput) to_roman should fail with large input ... ok ① ---------- ---------- ---------- -------- Ran 2 tests in 0.000s OK

1. Hurray! Passed both tests. Since you worked, switching between coding and testing, you can confidently say that it was the last 2 lines of code that allowed the test to return "success" and not "failure". Such confidence did not come cheap, but it will pay off with interest in the future.

More STOPs, more Fire

Along with testing for too large "input", it is necessary to test too small. As noted in the functionality requirements, Roman numerals cannot be less than or equal to 0.

>>> import roman2 >>> roman2. to_roman (0) " >>> roman2. to_roman (-1) "

Not good. Let's add tests for each case.

class ToRomanBadInput (unittest . TestCase) : def test_too_large (self) : " 'to_roman should fail with large input' " self . assertRaises (roman3. OutOfRangeError , roman3. to_roman , 4000) ① def test_zero (self) : " 'to_roman should fail with 0 input' " self . assertRaises (roman3.

OutOfRangeError , roman3. to_roman , 0) ② def test_negative (self) : " 'to_roman should fail with negative input' " self . assertRaises (roman3. OutOfRangeError , roman3. to_roman , - 1) ③

1. The test_too_large () method has not changed. I included it here to show the similarity of the code.

2. This is a new test: test_zero (). Like test_too_large (), we are forcing the assertRaises () method defined in unittest.TestCase to call our to_roman () function with parameter "0", and check that it throws the appropriate exception, OutOfRangeError.

3. The test_negative () method is almost the same, but passes -1 to the to_roman () function. And none of these methods will return an OutOfRangeError (because our function returns a value), and the test is considered to have failed.

Now let's check that the test fails:

you @ localhost: ~ / diveintopython3 / examples $ python3 romantest3. py -v
test_to_roman_known_values (__main__ . KnownValues)
to_roman Should give known of result with known input the ... ok The
test_negative (__main__ . ToRomanBadInput)
to_roman Should the fail with negative input the ... the FAIL
test_too_large (__main__ . ToRomanBadInput)
to_roman Should the fail with large input the . .. ok
test_zero (__main__ . ToRomanBadInput)
to_roman should fail with 0 input ... FAIL
==========

====================================== FAIL: to_roman should fail with negative input --- -- ---------- Traceback (most recent call last) : File "romantest3.py" , line 86 , in test_negative self . assertRaises (roman3. OutOfRangeError , roman3. to_roman , - 1) AssertionError : OutOfRangeError not raised by to_roman ==========
====================================== FAIL: to_roman should fail with 0 input - -- ---------- Traceback (most recent call last) : File "romantest3.py" , line 82 , in test_zero self . assertRaises (roman3. OutOfRangeError , roman3. to_roman , 0) AssertionError : OutOfRangeError not raised by to_roman ---------- ---------- ---------- Ran 4 tests in 0.000s FAILED (failures = 2)

Sumptuously. Both tests failed as expected. Now let's turn to the code and see what we can do to successfully pass the test.

def to_roman (n) : " 'convert integer to Roman numeral' " if not (0 < n < 4000) : ① raise OutOfRangeError ('number out of range (must be 1..3999)') ② result = ' ' for numeral , integer in roman_numeral_map: while n > =

integer: result + = numeral n - = integer return result

1. An excellent example of Python shorthand: multiple comparisons in one line. This is equivalent to "If not ((0 <n) and (n <4000))", but easier to read. This one-line code covers the "bad" input range.

2. Changing the condition requires changing the exception message. The test framework doesn't care, but when manually debugging it, it can be difficult if the message doesn't describe the situation correctly.

I could provide a number of examples to show that the one-line code works, but I'll just run the test instead.

you @ localhost: ~ / diveintopython3 / examples $ python3 romantest3. py -v
test_to_roman_known_values (__main__ . KnownValues)
to_roman Should give known of result with known input the ... ok The
test_negative (__main__ . ToRomanBadInput)
to_roman Should the fail with negative input the ... ok The
test_too_large (__main__ . ToRomanBadInput)
to_roman Should the fail with large input the . .. ok
test_zero (__main__ . ToRomanBadInput)
to_roman should fail with 0 input ... ok ---------- ----------
---------- Ran 4 tests in 0.016s OK

And one more thing ...

Another functionality requirement is handling non-integers.

>>> import roman3 >>> roman3. to_roman (0.5) ① " >>> roman3. to_roman (1.0) ② 'I'

1. Oh, that's bad.

2. Oh, that's even worse.

Both cases should throw an exception. Instead, the function returns a false value.

Testing non-numbers is hard. First, let's define a NotIntegerError exception.

```
# roman4.py
class OutOfRangeError ( ValueError ) : pass
class NotIntegerError ( ValueError ) : pass
```

Next, let's write a test case to check if a NotIntegerError exception was thrown.

class ToRomanBadInput (unittest . a TestCase) : . ... def test_non_integer (self) : " 'to_roman should fail with non-integer input' " self . assertRaises (roman4. NotIntegerError , roman4. to_roman , 0.5)

We see , that the test is failed .

you @ localhost: ~ / diveintopython3 / examples $ python3 romantest4. py -v
test_to_roman_known_values (__main__ . KnownValues)
to_roman Should give known of result with known input the ... ok The
test_negative (__main__ . ToRomanBadInput)
to_roman Should the fail with negative input the ... ok The
test_non_integer (__main__ . ToRomanBadInput)
to_roman Should the fail with the non-integer The input ... FAIL

test_too_large (__main__ . ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__ . ToRomanBadInput)
to_roman should fail with 0 input ... ok ==========
==========
======= FAIL: to_roman should fail with non-integer input ----------
---------- ---------- Traceback (most recent call last)
: File "romantest4.py " , line 90 , in test_non_integer self . assertRaises (
roman4. NotIntegerError , roman4. to_roman , 0.5) AssertionError :
NotIntegerError not raised by to_roman ---------- ----------
---------- Ran 5 tests in 0.000s FAILED (failures = 1)

We write the code to pass the test.

def to_roman (n) : " 'convert integer to Roman numeral' " if not (0 < n <
4000) : raise OutOfRangeError ('number out of range (must be 1..3999)') if
not isinstance (n , int) : ① raise NotIntegerError ('non-integers can not be
converted') ② result = " for numeral , integer in roman_numeral_map:
while n > = integer: result + = numeral n - = integer return result

1. The built-in function isinstance () checks whether a variable belongs to a certain type (more precisely, technically, to a type inheritor).

2. If the argument n is not a number, throw our new NotIntegerError exception.

Finally, let's test the code against a test.

you @ localhost: ~ / diveintopython3 / examples $ python3 romantest4. py -v
test_to_roman_known_values (__main__ . KnownValues)
to_roman Should give known of result with known input the ... ok The
test_negative (__main__ . ToRomanBadInput)
to_roman Should the fail with negative input the ... ok The
test_non_integer (__main__ . ToRomanBadInput)
to_roman Should the fail with the non-integer The input ... ok
test_too_large (__main__ . ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__ . ToRomanBadInput)
to_roman should fail with 0 input ... ok ---------- ----------
---------- ------- Ran 5 tests in 0.000s OK

The to_roman () function has successfully passed all the tests, and no more tests occur to me, so it's time to move on to the from_roman () function.

Nice symmetry

Converting a number from roman to decimal is more complex than converting decimal to roman. The main challenge is validation. It is easy enough to check if the integer is positive; however, it is a little more difficult to check if the string is a valid Roman number. Fortunately, we have already written a regular expression that checks Roman numbers.

The task of converting the string itself remains. As we'll see in a minute, the from_roman () function is a trivial task, thanks to the data structure we have defined that maps integers to roman numbers.

But tests first. We'll need the known values to randomly check the correctness of the conversion. We will use the set of known_values described earlier for these values:

def test_from_roman_known_values (self) : " 'from_roman should give known result with known input' " for integer , numeral in self . known_values : result = roman5. from_roman (numeral) self . assertEqual (integer , result)

Here we see an interesting symmetry. The to_roman () and from_roman () functions are reciprocal. The first one converts the decimal representation of a number to Roman, the second one does the opposite. In theory, we should be able to "close the circle" by passing a number to the to_roman () function, then passing the result of the execution to the from_roman () function, the return value of which must match the original number:

n = from_roman (to_roman (n)) for all values of n

In this case, “all values” means any number in the range [1, 3999]. Let's write a test that passes all numbers from this interval to the to_roman () function, then calls from_roman () and checks that the result matches the original number:

class RoundtripCheck (unittest . TestCase) : def test_roundtrip (self) : " 'from_roman (to_roman (n)) == n for all n' " for integer in range (1 , 4000) : numeral = roman5. to_roman (integer) result = roman5. from_roman (numeral) self . assertEqual (integer , result)

Our new tests aren't even a failure yet - they failed because we haven't yet implemented the from_roman () function:

```
you @ localhost: ~ / diveintopython3 / examples $ python3 romantest5. py
E. E ....
======================================
==========================
ERROR: test_from_roman_known_values ( __main__ . KnownValues )
from_roman should give known result with known input
------ ---------------------------------------------- ----------
Traceback ( most recent call last ) :
File "romantest5.py" , line 78 , in test_from_roman_known_values result =
roman5. from_roman ( numeral ) AttributeError : 'module' object has no
attribute 'from_roman' ==============================
================================== ERROR: test_roundtrip (
__main__ . RoundtripCheck ) from_roman ( to_roman ( n ) ) == n for all n ---
------------------------------------ ------------------------- Traceback ( most
recent call last ) : File "romantest5.py" , line 103 , in test_roundtrip result =
roman5 ... from_roman ( numeral ) AttributeError : 'module' object has no
attribute 'from_roman' ------------------------------- ------------------------------
----- Ran 7 tests in 0.019s FAILED ( errors = 2 )
```

Stubbing a function will solve this problem:

```
# roman5.py
def from_roman ( s ) : " 'convert Roman numeral to integer' "
```

(Did you notice? I wrote a function that has nothing but a docstring. That's okay. It's Python. In fact, many developers have this style. “Don't stub; document!”)

Now the tests really fail:

you @ localhost: ~ / diveintopython3 / examples $ python3 romantest5. py F. F .... ========== ========== FAIL: test_from_roman_known_values (__main__ . KnownValues) from_roman should give known result with known input ------ ---------- ---------- Traceback (most recent call last) : File "romantest5.py" , line 79 , in test_from_roman_known_values self . assertEqual (integer , result) AssertionError : 1 ! = None ========== ========== FAIL: test_roundtrip (__main__ . RoundtripCheck) from_roman (to_roman (n)) == n for all n ---------- ---------- ---------- Traceback (most recent call last) : File "romantest5.py" , line 104 , in test_roundtrip self . assertEqual (integer , result) AssertionError : 1 ! = None ---------- ---------- Ran 7 tests in 0.002s FAILED (failures = 2)

Now let's write a function from _ roman ():

def from_roman (s) : "" "convert Roman numeral to integer" "" result = 0 index = 0 for numeral , integer in roman_numeral_map: while s [index: index + len (numeral)] == numeral: ① result + = integer index + = len (numeral) return result

The writing style here is exactly the same as in the to_roman () function. We go through all roman_numeral_map values, but instead of taking the largest integer as long as possible, we take the maximum roman representation of the number and search for it in the string as long as possible.

If you are still not quite clear on how from_roman () works, add output at the end of the loop:

def from_roman (s) : "" "convert Roman numeral to integer" "" result = 0 index = 0 for numeral , integer in roman_numeral_map: while s [index: index + len (numeral)] == numeral: result + = integer index + = len (numeral) print ('found' , numeral , 'of length' , len (numeral) , ', adding' , integer) >>> import roman5 >>> roman5. from_roman ('MCMLXXII') found M , of length 1 , adding 1000 found CM of length 2 , adding 900 found L of length 1 , adding 50 found X of length 1 , adding 10 found X of length 1 , adding 10 found I of length 1 , adding 1 found I of length 1 , adding 1 1972

Let's restart the tests :

you @ localhost: ~ / diveintopython3 / examples $ python3 romantest5. py
.....
---------- ----------
Ran 7 tests in 0.060s
OK

I have two news for you. Both are good. First, the from_roman () function works for correct input (at least for known values); secondly, our "circle" has closed. These two facts allow you to be sure that the to_roman () and from_roman () functions work correctly for all valid values. (Actually, this is not guaranteed to work correctly. In theory, the to_roman () function may have an implementation bug that results in an incorrect Roman representation of a number for some inputs, and the from_roman () function may have a "reverse" bug. which results in a number that coincidentally coincides with the original. If you are concerned, write more complex tests.)

More bad inputs

Now that the from_roman () function works properly with good input, it's time to fit in the last piece of the puzzle: making it work properly with bad input. That means finding a way to look at a string and determine if it's a valid Roman numeral. This is inherently more difficult than validating numeric input in the to_roman () function, but you have a powerful tool at your disposal: regular expressions. (If you're not familiar with regular expressions, now would be a good time to read the regular expressions chapter.) As you saw in Case Study: Roman Numerals, there are several simple rules for constructing a Roman numeral, using the letters M , D, C, L,

X, V, and I. Let's review the rules: 1. Sometimes characters are additive. I is 1, II is 2, and III is 3. VI is 6 (literally, “5 and 1”), VII is 7, and VIII is 8. 2. The tens characters (I, X, C, and M) can be repeated up to three times. At 4, you need to subtract from the next highest fives character. You can't represent 4 as IIII; instead, it is represented as IV (“1 less than 5”). 40 is written as XL (“10 less than 50”), 41 as XLI, 42 as XLII, 43 as XLIII, and then 44 as XLIV (“10 less than 50, then 1 less than 5”). 3. Sometimes characters are… the opposite of additive. By certain putting characters before others, you subtract from the final value. For example, at 9, you need to subtract from the next highest tens character: 8 is VIII, but 9 is IX (“1 less than 10”), not VIIII (since the I character can not be repeated four times). 90 is XC, 900 is CM. 4. The fives characters can not be repeated. 10 is always represented as X, never as VV. 100 is always C, never LL. 5. Roman numerals are read left to right, so the order of characters matters very much. DC is 600; CD is a completely different number (400, “100 less than 500”). CI is 101; IC is not even a valid Roman numeral (because you can't subtract 1 directly from 100; you would need to write it as XCIX, “10 less than 100, then 1 less than 10”). Thus, one useful test would be to ensure that the from_roman () function should fail when you pass it a string with too many repeated numerals. How many is “too many” depends on the numeral.

class FromRomanBadInput (unittest . TestCase) : def test_too_many_repeated_numerals (self) : " 'from_roman should fail with too many repeated numerals" ' for s in (' MMMM ' , ' DD ' , ' CCCC ' , ' LL ' , ' XXXX ' , ' VV ' , ' IIII ') : self . assertRaises (roman6. InvalidRomanNumeralError , roman6. from_roman , s)

Another useful test would be to check that certain patterns aren't repeated. For example, IX is 9, but IXIX is never valid.

def test_repeated_pairs (self) : " 'from_roman should fail with repeated pairs of numerals' " for s in ('CMCM' , 'CDCD' , 'XCXC' , 'XLXL' , 'IXIX' , 'IVIV') : self ... assertRaises (roman6. InvalidRomanNumeralError , roman6. from_roman , s)

A third test could check that numerals appear in the correct order, from highest to lowest value. For example, CL is 150, but LC is never valid, because the numeral for 50 can never come before the numeral for 100. This test includes a randomly chosen set of invalid antecedents: I before M, V before X, and so on.

def test_malformed_antecedents (self) : " 'from_roman should fail with malformed antecedents' " for s in ('IIMXCC' , 'VX' , 'DCM' , 'CMM' , 'IXIV' , 'MCMC' , 'XCX' , 'IVI' , 'LM' , 'LD' , 'LC') : self . assertRaises (roman6. InvalidRomanNumeralError , roman6. from_roman , s)

Each of these tests relies the from_roman () function raising a new exception, InvalidRomanNumeralError, which we haven't defined yet.

def test_malformed_antecedents (self) :
roman6.py
class InvalidRomanNumeralError (ValueError) : pass

All three of these tests should fail, since the from_roman () function doesn't currently have any validity checking. (If they don't fail now, then what the heck are they testing?)

you @ localhost: ~ / diveintopython3 / examples $ python3 romantest6. py
FFF .....
==========
==========
FAIL: test_malformed_antecedents (__main__ . FromRomanBadInput) from_roman should fail with malformed antecedents
------- ---------- -------
Traceback (most recent call last) :
File "romantest6.py" , line 113 , in test_malformed_antecedents self . assertRaises (roman6. InvalidRomanNumeralError , roman6. from_roman , s) AssertionError : InvalidRomanNumeralError not raised by from_roman
==========

====================================== FAIL: test_repeated_pairs (__main__ . FromRomanBadInput) from_roman should fail with repeated pairs of numerals -- ---------- Traceback (most recent call last) : File "romantest6.py" , line 107 , in test_repeated_pairs self . assertRaises (roman6. InvalidRomanNumeralError , roman6. from_roman , s) AssertionError : InvalidRomanNumeralError not raised by from_roman ==========
====================================== FAIL: test_too_many_repeated_numerals (__main__ . FromRomanBadInput) from_roman should fail with too many repeated numerals ---------- ---------- ---------- Traceback (most recent call last) : File "romantest6.py" , line 102 , in test_too_many_repeated_numerals self . assertRaises (roman6. InvalidRomanNumeralError , roman6. from_roman , s) AssertionError : InvalidRomanNumeralError not raised by from_roman ---------- ---------- ---------- Ran 10 tests in 0.058s FAILED (failures = 3)

Good deal. Now, all we need to do is add the regular expression to test for valid Roman numerals into the from_roman () function.

CD

And testing again ...

you @ localhost: ~ / diveintopython3 / examples $ python3 romantest7. py
.....
---------- ----------
Ran 10 tests in 0.066s
OK

And the award for major disappointment this year goes to ... I'm so worried ... the word "OK", derived from a successful test.

Refactoring

Immersion

Whether you like it or not, bugs do happen. Despite our best efforts to create complete unit tests, bugs still exist. What do I mean by the word "bug"? A bug is a test case that hasn't been written yet.

```
>>> import roman7
>>> roman7. from_roman ( " ) ①
0
```

① Actually, a bug. A call from_roman with an empty string (or any other sequence of characters that is not a valid Roman number) must fail with an InvalidRomanNumeralError exception.

After reproducing the bug and before fixing it, write a test case that ends with an error, thus illustrating the bug.

```
class FromRomanBadInput ( unittest . TestCase ) :
    ...
    ...
```

```
    ...
    def testBlank ( self ) :
        " 'from_roman should fail with blank string' "
        self . assertRaises ( roman6. InvalidRomanNumeralError , roman6.
from_roman , " ) ①
```

① Everything is extremely simple: call from_roman () with an empty string and check that an InvalidRomanNumeralError exception is thrown. The hardest part was finding the bug; now that such a bug is known to exist, coding the validation won't take long.

Since the code contains a bug and we have a test for checking this bug, this test case fails:

```
you @ localhost: ~ / diveintopython3 / examples $ python3 romantest8.py -v
from_roman should fail with blank string ... FAIL
from_roman should fail with malformed antecedents ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ok
to_roman should give known result with known input ... ok
from_roman ( to_roman ( n )) == n for all n ... ok
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

======================================================================
FAIL: from_roman should fail with blank string
---------------------------------------------- ----------------------
Traceback ( most recent call last ) :
  File "romantest8.py" , line 117 , in test_blank
    self.assertRaises ( roman8.InvalidRomanNumeralError,
roman8.from_roman, " )
AssertionError: InvalidRomanNumeralError not raised by from_roman

---------------------------------------------- ----------------------
Ran 11 tests in 0.171s
```

```
FAILED ( failures = 1 )
```

Only now you can fix the bug.

```
def from_roman ( s ) :
   " 'convert Roman numeral to integer' "
   if not s: ①
      raise InvalidRomanNumeralError ( 'Input can not be blank' )
   if not re . search ( romanNumeralPattern , s ) :
      raise InvalidRomanNumeralError ( 'Invalid Roman numeral: {}' . format
( s ))  ②

   result = 0
   index = 0
   for numeral , integer in romanNumeralMap:
      while s [ index: index + len ( numeral )] == numeral:
         result + = integer
         index + = len ( numeral )
   return result
```

① Only 2 lines of code are required: an explicit check with an empty string and an exception throw.

② Not sure if it was mentioned earlier in the book, so let this be the last trick when formatting strings. Since Python 3.1, it is allowed to omit numbers when using position indices in the format string. That is, instead of using {0} to refer to the first parameter of the format () method, you can write {} and Python will fill in the corresponding position index for you. This can be done for any number of arguments: the first {} are equal to {0}, the second {} are equal to {1}, and so on.

```
you @ localhost: ~ / diveintopython3 / examples $ python3 romantest8.py -v
from_roman should fail with blank string ... ok ①
from_roman should fail with malformed antecedents ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ok
to_roman should give known result with known input ... ok
from_roman ( to_roman ( n )) == n for all n ... ok
to_roman should fail with negative input ... ok
```

```
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

---------------------------------------------- --------------------
Ran 11 tests in 0.156s

OK ②
```

① The test for handling an empty string now passes, which means the bug has been fixed.

② All other test cases still run without errors, which means that when we fixed the error, we did not add new ones. It's time to stop editing the code!

Coding through writing tests does not make the bug fixing process easier. To fix simple bugs (like the example above), simple tests are needed; complex bugs, of course, require complex tests. If a project is being developed through testing, it *may seem* that fixing the bug will take longer, since you have to find the lines of code with the bug (in fact, write a test case to check these lines) and then fix the bug. If the test fails again, then you have to figure out whether the bug was fixed correctly or the test itself contains errors. However, with long-term development, these fixes in code-in-test-in code pay off, as most likely the bug will be fixed the first time. Also, since you can easily rerun *all* tests, including the new one, you are unlikely to "mess up" the old code when you fix a bug. Today's unit tests will become regression tests tomorrow.

Files

11. Files

"Nine miles of walking is no joke, especially in the rain." —Harry Kemelman, The Nine Mile Walk

There were 38493 files on my Windows laptop before I installed one application. Installing Python 3 added almost 3000 files to the total. Files represent the primary storage paradigm in major operating systems; this concept is so ingrained that most people will not accept anything else as an alternative. Figuratively speaking, your computer is drowning in a sea of files.

11.2 Reading from text files

Before reading from a text file, you need to open it. Opening a file in Python is easy:

a_file = open ('examples / chinese.txt', encoding = 'utf-8')

Python has a built-in open () function that is passed a filename as an argument. In the example, the file name is 'examples / chinese.txt' and it has 5 interesting things:

1. It is not just a filename, it is a combination of a directory path and filename. Hypothetically, two parameters could be passed to the file open function: the path to the file and the file name, but only one can be passed to the open () function. In Python, you can include some or all of the directory paths when needed.
2. When specifying a directory path, / (forward slash, right slash) is used, without discussing which operating system is used. Windows uses \ (backslash, backslash, left slash) to indicate directory paths, while Linux and MacOS operating systems use / (forward backslash, slash, right slash). In Python, forward slash just always works, even on Windows.
3. The directory path does not start with a forward slash or letter, it is called a relative path. About what? Have patience, grasshopper!
4. These are lines. All modern operating systems (including Windows) use Unicode for storing file and directory names. Python 3 fully supports non-ascii paths.
5. The file does not need to be on local drives. You can use network drives. This file can be a virtual filesystem object (/ proc on linux). If your computer thinks it is a file and lets you access it

as a file, then Python can open that file.

The open () function call is not limited to passing a file path parameter and file name. There is one more parameter called encoding. Oh yes, dear reader, this sounds truly awful!

11.2.1 Encoding features show their scary face

Bytes bytes; abstraction symbols. A string is a sequence of Unicode characters. But the files on disk are not a sequence of Unicode characters, but a sequence of bytes. If you are reading a text file from disk, then how does Python convert this sequence of bytes to a sequence of characters? It decodes a byte using a specific encoding algorithm and returns a sequence of Unicode characters (that is , as a string).

```
>> file = open ( 'examples / chinese.txt' )
... >>> a_string = file . read ()
... Traceback ( most recent call last ) :
... File " < stdin > " , line 1 , in < module >
... File "C: \ Python31 \ lib \ encodings \ cp1252. py " , line 23 , in decode
... return codecs . charmap_decode ( input , self . errors , decoding_table ) [ 0
]
... UnicodeDecodeError : 'charmap' codec can't decode byte 0x8f in position
28 : character maps to < undefined >
```

[[Category : Diving into Python 3]]

XML

Immersion

Most of the chapters in this book are built on snippets, code examples. But xml is more data than code. One way to use xml is to "syndicate content" such as recent articles from a blog, forum, or other frequently updated site. Most popular blogging software can create feeds and update them when new articles, topics are published. You can follow a blog by subscribing to its feed, or you can follow multiple blogs using "aggregator programs" such as Google Reader [1]

So, below is the XML data that we will be working with in this chapter. This feed format Atom syndication feed

```
<? xml version = '1.0' encoding = 'utf-8' ?>
<feed xmlns = 'http://www.w3.org/2005/Atom' xml: lang = 'en' >
  <title > dive into mark </ title >
  <subtitle > currently between addictions </ subtitle >
  <id > tag: diveintomark.org, 2001-07-29: / </ id >
  <updated > 2009-03-27T21: 56: 07Z </ updated >
  <link rel = 'alternate' type = 'text / html' href = 'http://diveintomark.org/' />
  <link rel = 'self' type = 'application / atom + xml' href =
'http://diveintomark.org/feed/' />
  <entry >
    <author >
      <name > Mark </ name >
      <uri > http://diveintomark.org/ </ uri >
    </ author >
    <title > Dive into history, 2009 edition </ title >
    <link rel = 'alternate' type = 'text / html'
      href = 'http://diveintomark.org/archives/2009/03/27/dive-into-history-
2009-edition' />
    <id > tag: diveintomark.org, 2009-03-27: / archives / 20090327172042 </
id >
    <updated > 2009-03-27T21: 56: 07Z </ updated >
    <published > 2009-03-27T17: 20: 42Z </ published >
    <category scheme = 'http://diveintomark.org' term = 'diveintopython' />
    <category scheme = 'http://diveintomark.org' term = 'docbook' />
    <category scheme = 'http://diveintomark.org' term = 'html' />
  <summary type = 'html' > Putting an entire chapter on one page sounds
    bloated, but consider this & amp; mdash; my longest chapter so far
    would be 75 printed pages, and it loads in under 5 seconds & amp; hellip;
    On dialup. </ summary >
  </ entry >
  <entry >
    <author >
      <name > Mark </ name >
      <uri > http://diveintomark.org/ </ uri >
    </ author >
```

```
   <title > Accessibility is a harsh mistress </ title >
   <link rel = 'alternate' type = 'text / html'
     href = 'http://diveintomark.org/archives/2009/03/21/accessibility-is-a-
harsh-mistress' />
   <id > tag: diveintomark.org, 2009-03-21: / archives / 20090321200928 </
id >
   <updated > 2009-03-22T01: 05: 37Z </ updated >
   <published > 2009-03-21T20: 09: 28Z </ published >
   <category scheme = 'http://diveintomark.org' term = 'accessibility' />
   <summary type = 'html' > The accessibility orthodoxy does not permit
people to
     question the value of features that are rarely useful and rarely used. </
summary >
  </ entry >
  <entry >
   <author >
     <name > Mark </ name >
   </ author >
   <title > A gentle introduction to video encoding, part 1: container formats
</ title >
   <link rel = 'alternate' type = 'text / html'
     href = 'http://diveintomark.org/archives/2008/12/18/give-part-1-container-
formats' />
   <id > tag: diveintomark.org, 2008-12-18: / archives / 20081218155422 </
id >
   <updated > 2009-01-11T19: 39: 22Z </ updated >
   <published > 2008-12-18T15: 54: 22Z </ published >
   <category scheme = 'http://diveintomark.org' term = 'asf' />
   <category scheme = 'http://diveintomark.org' term = 'avi' />
   <category scheme = 'http://diveintomark.org' term = 'encoding' />
   <category scheme = 'http://diveintomark.org' term = 'flv' />
   <category scheme = 'http://diveintomark.org' term = 'GIVE' />
   <category scheme = 'http://diveintomark.org' term = 'mp4' />
   <category scheme = 'http://diveintomark.org' term = 'ogg' />
   <category scheme = 'http://diveintomark.org' term = 'video' />
   <summary type = 'html' > These notes will eventually become part of a
     tech talk on video encoding. </ summary >
```

```
  </ entry >
</ feed >
```

5 minute introduction to XML

If you are already familiar with XML, you can skip this chapter.

XML is a markup language for describing a hierarchy of structured data. An XML *document* contains one or more *elements* separated by *opening* and *closing tags* . This is a valid, albeit uninteresting, XML document:

```
<foo >    ①
</ foo >   ②
```

① This is the *opening (start)* tag of the foo element.

② This is the corresponding *closing (end)* tag of the foo element. As in mathematics and programming languages, each opening parenthesis must have a corresponding closing parenthesis; in XML, every opening tag must be *closed* with a corresponding closing tag .

Elements can be *nested* within each other indefinitely . Since the bar element is nested within the foo element, it is called a *sub-element* or *child of the* foo element.

```
<foo >
  <bar > </ bar >
</ foo >
```

The first element of every XML document is called the root element. An XML document can contain only one root element. The example below is **not an XML document** , as it has two root elements:

```
<foo > </ foo >
<bar > </ bar >
```

Elements can have *attributes* consisting of a name-value pair. Attributes are listed inside the opening tag of the element and separated by spaces. *Attribute names* cannot be repeated within the same element. *Attribute values* must be surrounded by single or double quotes.

```
<foo lang = 'en' >                          ①
  <bar id = 'papayawhip' lang = "fr" > </ bar >   ②
</ foo >
```

① The foo element has one attribute named lang. The lang attribute value is

assigned the string en.

② The bar element has two attributes: id and lang. The lang value is fr. This does not conflict with the lang attribute of the foo element, since each element has its own set of attributes.

If an element has more than one attribute, the order of the attributes is irrelevant. Element attributes are an unordered collection of keys and values, similar to dictionaries in Python. An unlimited number of attributes can be specified for each element.

Elements can have *text (text content)* .

```
<foo lang = 'en' >
  <bar lang = 'fr' > PapayaWhip </ bar >
</ foo >
```

Elements that do not contain text or children are called *empty* .

```
<foo > </ foo >
```

There is a shorthand for an empty element. By placing a fraction / at the end of the start tag, you can omit the end tag. The XML document of the previous example with empty elements can be written as follows:

```
<foo />
```

Just as Python functions can be declared in different *modules* , XML elements can be declared in different *namespaces* . Namespaces usually look like URL paths. The xmlns directive is used to declare the *default namespace* . A namespace declaration is very similar to an attribute, but has a special meaning.

```
<feed xmlns = 'http://www.w3.org/2005/Atom' >   ①
  <title > dive into mark </ title >             ②
</ feed >
```

① The feed element is in the http://www.w3.org/2005/Atom namespace .

② The title element is also in the http://www.w3.org/2005/Atom namespace . The namespace applies both to the element in which it was defined and to all child elements.

You can declare the xmlns: prefix namespace and match *the* prefix to it. Then

each element in the given namespace must be explicitly declared with the prefix prefix.

```
<atom: feed xmlns: atom = 'http://www.w3.org/2005/Atom' >   ①
  <atom: title > dive into mark </ atom: title >            ②
</ atom: feed >
```

① The feed element is in the http://www.w3.org/2005/Atom namespace .

② The title element is also in the http://www.w3.org/2005/Atom namespace .

From the point of view of the XML parser, the previous two XML documents are identical. The pair "namespace" + "element name" specifies the XML identity. Prefixes are used only to refer to the namespace, and do not change the attribute name. If the namespaces match, the element names match, the attributes (or lack thereof) match, and the element texts match, then the XML documents are the same.

Finally, XML documents can contain character encoding information on the first line up to the root element. (If you are interested in how a document can contain information that must be known to the XML parser prior to parsing the XML document, see Catch-22 section F of the XML specification)

```
<? xml version = '1.0' encoding = 'utf-8' ?>
```

Now you know enough about XML to get the next sections of the chapter out of your way!

Atom feed syndication format structure

Consider a blog (weblog) or any site with frequently updated content such as CNN.com . The site contains a headline ("CNN.com"), a subtitle ("Breaking News, US, World, Weather, Entertainment & Video News"), the date it was last modified ("updated 12:43 pm EDT, Sat May 16, 2009") and list of articles published at different times. Each article, in turn, also has a title, the date of the first publication (and, possibly, the date of the last update, in case the article was corrected) and a unique URL.

The Atom syndication format is designed to store this kind of information in a standard way. My blog and CNN.com are completely different in design, content and site visitors, but both share a similar structure. Both sites have headlines and publish articles.

At the top level, an Atom feed must have a root element named feed in the http://www.w3.org/2005/Atom namespace .

```
<feed xmlns = 'http://www.w3.org/2005/Atom'   ①
    xml: lang = 'en' >                        ②
```

① the http://www.w3.org/2005/Atom - space names Atom

② Each element can contain an xml: lang attribute that defines the language of the element and its children. In this case, the xml: lang attribute declared in the root element sets the English language for the entire feed.

The Atom feed contains additional information about itself in the children of the root element:

```
<feed xmlns = 'http://www.w3.org/2005/Atom' xml: lang = 'en' >
  <title > dive into mark </ title >                                    ①
  <subtitle > currently between addictions </ subtitle >                ②
  <id > tag: diveintomark.org, 2001-07-29: / </ id >                    ③
  <updated > 2009-03-27T21: 56: 07Z </ updated >                        ④
  <link rel = 'alternate' type = 'text / html' href = 'http://diveintomark.org/' />
⑤
```

① The title contains the text 'dive into mark'.

② The subtitle of the feed subtitle is the line 'currently between addictions'.

③ Each feed must have a globally unique identifier. RFC 4151 contains information on how to create such identifiers.

④ This feed was last updated on March 27, 2009 at 21:56 GMT. The updated element is usually equivalent to the last modified date of any article on the site.

⑤ And this is where the fun begins. The link element has no text content, but has three attributes: rel, type, and href. The value of the rel attribute tells you what type of link is. rel = 'alternate' means this is an alternate link for this feed. The type = 'text / html' attribute says that this is a link to an HTML page. And, in fact, the link path is contained in the href attribute.

We now know that the above feed is from the dive into mark site. The site is available at http://diveintomark.org/ and was last updated on March 27, 2009.

> While the order of elements may be important in some XML documents, in Atom feeds, the order of elements is arbitrary.

Let's continue to look at the structure of the feed: after the meta information about the feed, there is a list of recent articles. The article looks like this:

```
<entry >
  <author >                                                    ①
    <name > Mark </ name >
    <uri > http://diveintomark.org/ </ uri >
  </ author >
  <title > Dive into history, 2009 edition </ title >                    ②
  <link rel = 'alternate' type = 'text / html'                          ③
    href = 'http://diveintomark.org/archives/2009/03/27/dive-into-history-
2009-edition' />
  <id > tag: diveintomark.org, 2009-03-27: / archives / 20090327172042 </ id
>         ④
  <updated > 2009-03-27T21: 56: 07Z </ updated >                        ⑤
  <published > 2009-03-27T17: 20: 42Z </ published >
  <category scheme = 'http://diveintomark.org' term = 'diveintopython' />
⑥
  <category scheme = 'http://diveintomark.org' term = 'docbook' />
  <category scheme = 'http://diveintomark.org' term = 'html' />
  <summary type = 'html' > Putting an entire chapter on one page sounds ⑦
    bloated, but consider this & amp; mdash; my longest chapter so far
    would be 75 printed pages, and it loads in under 5 seconds & amp; hellip;
    On dialup. </ summary >
</ entry >                                                      ⑧
```

① The author element tells who wrote the article: some guy named Mark who is fooling around at http://diveintomark.org/ (In this case, the link to the author's site is the same as the alternative link in the feed meta information, but this is not always true, as many blogs have multiple authors, each with their own site.)

② The title element contains the title of the article "Dive into history, 2009 edition".

③ As with alternate feed links, the link element contains the URL of the HTML version of this article.

④ The entry element, like feeds, has a unique identifier.

⑤ The entry element has two dates: the date it was first posted and the date

it was last modified.

⑥ Entry elements can have any number of categories. The article in question will fall into the diveintopython, docbook, and html categories.

⑦ The summary element provides an overview of the article. (There is also a content element not shown here that is intended to be included in a feed with the full text of an article.) This summary element contains an Atom feed-specific type = 'html' attribute indicating that the content of the element is HTML text. This is important because the & mdash; and & hellip; those present in the element should be displayed as "-" and "..." and not printed "as is".

⑧ Finally, the end tag of the entry element indicates the end of the metadata for this article.

Parsing XML

In Python, XML documents can be processed using different libraries. The language has the usual DOM and SAX parsers , but I will be using a different library, ElementTree.

```
>>> import xml . etree . ElementTree as etree ①
>>> tree = etree. parse ( 'examples / feed.xml' )  ②
>>> root = tree. getroot ()                ③
>>> root ④
< Element { http: // www. w3 . org / 2005 / Atom } feed at cd1eb0 >
```

① Module ElementTree included in the standard library of Python, a way to import xml.etree.ElementTree.

② The parse () function is the core function of the ElementTree module. The function takes a file name or file-like object. This function parses the document at a time. If the developed program needs to save memory, then you can parse the XML document in parts.

③ The parse () function returns an object that is a representation of the entire document. However, the tree object is not the root element. To get a reference to the root element, you must call the getroot () method.

④ As you might expect, the root element is a feed element in the http://www.w3.org/2005/Atom namespace. The string representation of the root object once again underlines an important point: an XML element is a combination of a namespace and its tag-name (also called a local name).

Every item in this document is in Atom space, so the root item is represented as {http://www.w3.org/2005/Atom }feed.

> The ElementTree module always represents XML elements as '{namespace} local name'. You will have to use this format repeatedly when using the ElementTree API.

XML elements are Python lists

In the ElementTree API, elements are represented by Python's built-in list type. Each of the list items are XML child items.

```
# continuation of the previous example
>>> root. tag                    ①
'{http://www.w3.org/2005/Atom>feed'
>>> len ( root )                 ②
8
>>> for child in root: ③
...   print ( child )             ④
...
< Element { http: // www. w3 . org / 2005 / Atom } title at e2b5d0 >
< Element { http: // www. w3 . org / 2005 / Atom } subtitle at e2b4e0 >
< Element { http: // www. w3 . org / 2005 / Atom } id at e2b6c0 >
< Element { http: // www. w3 . org / 2005 / Atom } updated at e2b6f0 >
< Element { http: // www. w3 . org / 2005 / Atom } link at e2b4b0 >
< Element { http: // www. w3 . org / 2005 / Atom } entry at e2b720 >
< Element { http: // www. w3 . org / 2005 / Atom } entry at e2b510 >
< Element { http: // www. w3 . org / 2005 / Atom } entry at e2b750 >
```

① Continuing the previous example: the root element is root - {http://www.w3.org/2005/Atom]

② The "length" of the root element is the number of children of the root.

③ You can use an element as an iterator over all children.

④ From the posts you can see that the root element has 8 child elements: 5 elements with feed meta information (title, subtitle, id, updated and link) and 3 elements with entry articles.

You must have guessed by now, but I want to point out this explicitly: the child list only contains direct children. In turn, each child entry element can

contain its own children, but they will not be included in the list. They will be included in the list of the entry element, not the list of sub-elements of the feed element. There are several ways to find specific elements of any nesting level; below we will consider 2 of them.

XML attributes are Python dictionaries

Recall that an XML document is not just a collection of elements; each element also has a set of attributes. Given a specific XML element, you can easily retrieve its attributes like a Python dictionary.

```
# continuation of the previous example
>>> root. attrib                    ①
{ '{http://www.w3.org/XML/1998/namespace#Lang' : 'en' }
>>> root [ 4 ]                      ②
< Element { http: // www. w3 . org / 2005 / Atom } link at e181b0 >
>>> root [ 4 ] . attrib             ③
{ 'href' : 'http://diveintomark.org/' ,
'type' : 'text / html' ,
'rel' : 'alternate' }
>>> root [ 3 ]                      ④
< Element { http: // www. w3 . org / 2005 / Atom } updated at e2b4e0 >
>>> root [ 3 ] . attrib             ⑤
{}
```

① The attrib property returns a dictionary of the element's attributes. The original XML markup was <feed xmlns = 'http: //www.w3.org/2005/Atom' xml: lang = 'en'>. The xml prefix: refers to a standard namespace that any XML document can use without being declared.

② The fifth subelement is the link element (index [4] is used, since Python lists are indexed starting at 0).

③ The link subelement has three attributes href, type, and rel.

④ The fourth subelement (with index [3] in the list starting with 0) is the updated element.

⑤ The updated subelement has no attributes, hence the .attrib property returns an empty dictionary.

Finding nodes in an XML document

So far, we've looked at an XML document from top to bottom, starting at the

root element, down to its children, and so on down to the bottom of the document. However, in many cases when working with XML, you need to search for specific elements. Etree handle and with this task .

```
>>> import xml . etree . ElementTree as etree
>>> tree = etree. parse ( 'examples / feed.xml' )
>>> root = tree. getroot ()
>>> root. findall ( '{http://www.w3.org/2005/Atom#Entry' )     ①
[ < Element { http: // www. w3 . org / 2005 / Atom } entry at e2b4e0 >,
< Element { http: // www. w3 . org / 2005 / Atom } entry at e2b510 >,
< Element { http: // www. w3 . org / 2005 / Atom } entry at e2b540 > ]
>>> root. tag
'{http://www.w3.org/2005/Atom>feed'
>>> root. findall ( '{http://www.w3.org/2005/Atom#Feed' )      ②
[]
>>> root. findall ( '{http://www.w3.org/2005/Atom>author' )    ③
[]
```

① The findall () method searches for children that match the query. (The format of the request is discussed below.)

② All elements (including root and children) have a findall () method. The method finds all elements among the children that match the request. Why did the method return an empty list? While this may seem counterintuitive, this query only searches child elements. Since the root element feed has no children named feed, the query returns an empty list.

③ This result may also surprise you. The XML document does indeed have an author element; in fact, there are even three of them (one in each entry). But these author elements are not *direct children of the* root element; they are "sub-sub-elements" (sub-elements of a sub-element). If you need to find author elements of any nesting level, you have to change the query string.

```
>>> tree. findall ( '{http://www.w3.org/2005/Atom#Entry' )     ①
[ < Element { http: // www. w3 . org / 2005 / Atom } entry at e2b4e0 >,
< Element { http: // www. w3 . org / 2005 / Atom } entry at e2b510 >,
< Element { http: // www. w3 . org / 2005 / Atom } entry at e2b540 > ]
>>> tree. findall ( '{http://www.w3.org/2005/Atom>author' )    ②
[]
```

① For convenience, the tree object (which the etree.parse () function returns)

has several methods that are identical to those of the root element. The function results are the same as when calling the tree.getroot (). Findall () method.

② Probably surprised, but this query does not find an author element in this document. Why then? Because this call is identical to the call to tree.getroot () .findall ('{http://www.w3.org/2005/Atom} author'), which means "find all author elements that are subelements of the root element." Author elements are not children of the root element; they are sub-elements of entry elements. Therefore, no matches were found when executing the query.

Besides the findall () method, there is a find () method that only returns the first element found. The method can be useful in cases when as a result of a search you expect only one element or you only care about the first element from the list of found ones.

```
>>> entries = tree. findall ( '{http://www.w3.org/2005/Atom>entry' )
①
>>> len ( entries )
3
>>> title_element = entries [ 0 ] . find (
'{http://www.w3.org/2005/Atom#Title' )   ②
>>> title_element. text
'Dive into history, 2009 edition'
>>> foo_element = entries [ 0 ] . find ( '{http://www.w3.org/2005/Atom>foo'
)      ③
>>> foo_element
>>> type ( foo_element )
< class 'NoneType' >
```

① As you saw in the previous example, findall () returns a list of atom: entry elements.

② The find () method takes an ElementTree request and returns the first element that satisfies the request.

③ There are no children in foo, so find () returns a None object.

> There is a catch here when using the find () method. In a boolean context, ElementTree objects with no children are False (i.e., if len (element) evaluates to 0). Code the if element. find ('...') does not check whether the find () method has found a matching item; the

code checks if the found element contains child elements! In order to check if the find () method has found an element, use if element. find ('...') is not None .

Consider searching within child elements, i.e. subelements, sub-subelements, and so on at any nesting level.

```
>>> all_links = tree. findall ( '// {http://www.w3.org/2005/Atom>link' )   ①
>>> all_links
[ < Element { http: // www. w3 . org / 2005 / Atom } link at e181b0 >,
< Element { http: // www. w3 . org / 2005 / Atom } link at e2b570 >,
< Element { http: // www. w3 . org / 2005 / Atom } link at e2b480 >,
< Element { http: // www. w3 . org / 2005 / Atom } link at e2b5a0 > ]
>>> all_links [ 0 ] . attrib                                    ②
{ 'href' : 'http://diveintomark.org/' ,
'type' : 'text / html' ,
'rel' : 'alternate' }
>>> all_links [ 1 ] . attrib                                    ③
{ 'href' : 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition' ,
'type' : 'text / html' ,
'rel' : 'alternate' }
>>> all_links [ 2 ] . attrib
{ 'href' : 'http://diveintomark.org/archives/2009/03/21/accessibility-is-a-harsh-mistress' ,
'type' : 'text / html' ,
'rel' : 'alternate' }
>>> all_links [ 3 ] . attrib
{ 'href' : 'http://diveintomark.org/archives/2008/12/18/give-part-1-container-formats' ,
'type' : 'text / html' ,
'rel' : 'alternate' }
```

① This request - //▪http://www.w3.org/2005/Atom†link - is very similar to the requests from the previous examples. The difference is that there are two forward slashes // at the beginning of the query string. The // symbols mean "I want to find all elements regardless of nesting level, not just immediate children." Therefore, the method returns a list of four elements, not one.

② The first element of the result is a direct sub-element of the root element.

As we can see from its attributes, this is an alternate feed-level link that points to the html version of the website on which the feed is located.

③ The other three result elements are entry-level alternative references. Each of the entry elements has one link subelement. Since the findall () query contained double slashes at the beginning of the query, the search result contains all link subelements.

Overall, the findall () method of the ElementTree library is quite a powerful search tool, but the query format can be a little unpredictable. The ElementTree query format is officially described as "limited support for XPath expressions . " XPath is the W3C standard for building search queries within an XML document. On the one hand, the ElementTree query format is similar enough to the XPath format for performing basic searches. On the other hand, it is so different that it can get annoying if you already know XPath. Next, we'll look at third-party XML libraries that extend the ElementTree API to fully support the XPath standard.

Working with LXML

lxml is a third-party open source library based on the well-known libxml2 parser . The library provides 100% compatibility with the ElementTree API, fully supports XPath 1.0, and has several other nice features. For Windows, you can download the installer ; Linux users should check for compiled packages in the distribution repositories (for example using yum or apt-get tools). Otherwise, you have to install lxml manually .

```
>>> from lxml import etree ①
>>> tree = etree. parse ( 'examples / feed.xml' )  ②
>>> root = tree. getroot ()                ③
>>> root. findall ( '{http://www.w3.org/2005/Atom>entry' )  ④
[ < Element { http: // www. w3 . org / 2005 / Atom } entry at e2b4e0 >,
< Element { http: // www. w3 . org / 2005 / Atom } entry at e2b510 >,
< Element { http: // www. w3 . org / 2005 / Atom } entry at e2b540 > ]
```

① When importing, lxml provides exactly the same API as the built-in ElementTree library.

② parse () function: same as in ElementTree.

③ The getroot () method: the same.

④ The findall () method: exactly the same.

When processing large XML documents, lxml is significantly faster than the built-in ElementTree library. If you use functions only from the ElementTree API and want the processing to be done as quickly as possible, then you can try importing the lxml library and, if it is not there, use ElementTree.

```
try :
   from lxml import etree
except ImportError :
   import xml . etree . ElementTree as etree
```

However, lxml is not only faster than ElementTree: the findall () method supports more complex queries.

```
>>> import lxml. etree                                                    ①
>>> tree = lxml. etree . parse ( 'examples / feed.xml' )
>>> tree. findall ( '// {http://www.w3.org/2005/Atom }* [@href]' )
                    ②
[ < Element { http: // www. w3 . org / 2005 / Atom } link at eeb8a0 >,
< Element { http: // www. w3 . org / 2005 / Atom } link at eeb990 >,
< Element { http: // www. w3 . org / 2005 / Atom } link at eeb960 >,
< Element { http: // www. w3 . org / 2005 / Atom } link at eeb9c0 > ]
>>> tree. findall ( "// {http://www.w3.org/2005/Atom>*
[@href='http://diveintomark.org/ ']" )   ③
[ < Element { http: // www. w3 . org / 2005 / Atom } link at eeb930 > ]
>>> NS = '{http://www.w3.org/2005/Atom}'
>>> tree. findall ( '// {NS} author [{NS} uri]' . format ( NS = NS ))
                        ④
[ < Element { http: // www. w3 . org / 2005 / Atom } author at eeba80 >,
< Element { http: // www. w3 . org / 2005 / Atom } author at eebba0 > ]
```

① In this example, I import the lxml.etree object (instead of the etree object: from lxml import etree) to emphasize that the described functionality is only possible with lxml.

② This query will find all elements in the Atom namespace (any nesting) that have an href attribute. The // characters at the beginning of a query denote "any nesting elements, not just descendants of the root element." {http://www.w3.org/2005/Atom} stands for “elements of the Atom namespace only”. The * means “elements with any local name”. And [@href] stands for "the element has an href attribute".

③ The query found all Atom elements with the href attribute equal to http://diveintomark.org/.

④ After transforming the string (otherwise these queries become incredibly long), this query looks for Atom author elements that have Atom uri subelements. The request only returns 2 author elements: in the first and second entry elements. In the last entry, the author element contains only name, not uri.

Not enough for you? lxml has built-in support for XPath 1.0 expressions. We will not go into detail about XPath syntax, as this is a topic for a separate book. However, we will look at an example of using XPath in lxml.

```
>>> import lxml. etree
>>> tree = lxml. etree . parse ( 'examples / feed.xml' )
>>> NSMAP = { 'atom' : 'http://www.w3.org/2005/Atom' }                ①
>>> entries = tree. xpath ( "// atom: category [@ term = 'accessibility'] / .." ,
②
...     namespaces = NSMAP )
>>> entries ③
[ < Element { http: // www. w3 . org / 2005 / Atom } entry at e2b630 > ]
>>> entry = entries [ 0 ]
>>> entry. xpath ( './atom:title/text ()' , namespaces = NSMAP )              ④
[ 'Accessibility is a harsh mistress' ]
```

① To perform an XPath query for elements from a namespace, you must define a mapping for the prefix of that namespace. This is actually a regular Python dictionary.

② And here is the XPath query. This expression searches for category elements (Atom namespaces) containing an attribute with a name-value pair term = 'accessibility'. But that's not exactly what the query returns. Did you notice the / .. characters at the end of the query string? This means "return the element that was not found, but its parent." And so, with one query, we will find all entry elements with <category term = 'accessibility'> children.

③ The xpath () function returns a list of ElementTree objects. The analyzed document contains only one entry element with the term = 'accessibility' attribute.

④ The XPath expression does not always return a list of items. Formally, the DOM of a parsed XML document contains no elements, it contains *nodes* .

Depending on their type, nodes can be elements, attributes, or even text. The result of an XPath query is always a list of nodes. This query returns a list of text nodes: the text () of the title (atom: title) element is a sub-element of the current element (./).

XML creation

ElementTree can not only parse existing XML documents, but also create them from scratch.

```
>>> import xml . etree . ElementTree as etree
>>> new_feed = etree. Element ( '{http://www.w3.org/2005/Atom#Ffeed' ,
    ①
...     attrib = { '{http://www.w3.org/XML/1998/namespace#Lang' : 'en' })
②
>>> print ( etree. tostring ( new_feed ))                    ③
< ns0: feed xmlns: ns0 = 'http://www.w3.org/2005/Atom' xml : lang = 'en' / >
```

① To create a new element, you need to create an object of the Element class. As the first parameter to the constructor, we pass the element name (namespace and local name). This expression creates a feed element in the Atom space. This will be the root element of our new XML document.

② In order to add attributes to the element being created, we pass a dictionary of attribute names and their values in the second argument attrib. Note that attribute names must be in the format ElementTree {namespace} local_name.

③ At any time you can serialize the element and its subelements using the tostring () function of the ElementTree library.

Are you surprised at the result of serializing new_feed? Formally, ElementTree serializes XML elements correctly, but not optimally. The sample XML document at the beginning of the chapter is defined in the *default space* xmlns = 'http: //www.w3.org/2005/Atom'. Defining a default space is useful for documents (e.g., Atom feeds) where all elements belong to the same space, that is, you can declare a space once and reference the elements using a local name (<feed>, <link>, <entry>) ... Unless you intend to declare elements from a different namespace, then there is no need to use the default space prefix.

The XML parser will not "notice" the difference between a default spaced XML document and a namespaced document in front of each element. The

resulting DOM of this serialization looks like

```
<ns0: feed xmlns: ns0 = 'http://www.w3.org/2005/Atom' xml: lang = 'en'
/>
```

which is equivalent

```
<feed xmlns = 'http://www.w3.org/2005/Atom' xml: lang = 'en' />
```

The only difference is that the second option is a few characters shorter. If we rewrite our example using the ns0: prefix in each start and end tag, that would add 4 characters per start tag × 79 tags + 4 characters per namespace declaration, 320 characters in total. In UTF-8 encoding, this would be 320 bytes. (After gzip archiving, the difference is reduced to 21 bytes; however, 21 bytes is 21 bytes). You might not have paid attention to those tens of bytes, but for Atom feeds that are downloaded a thousand times when modified, the gain of a few bytes per request quickly turns into kilobytes.

Another advantage of lxml: unlike the standard ElementTree library, lxml provides finer control over the serialization of elements.

```
>>> import lxml. etree
>>> NSMAP = { None : 'http://www.w3.org/2005/Atom' }                ①
>>> new_feed = lxml. etree . Element ( 'feed' , nsmap = NSMAP )
②
>>> print ( lxml. etree . tounicode ( new_feed ))                   ③
< feed xmlns = 'http://www.w3.org/2005/Atom' / >
>>> new_feed. set ( '{http://www.w3.org/XML/1998/namespace#Lang' , 'en'
)  ④
>>> print ( lxml. etree . tounicode ( new_feed ))
< feed xmlns = 'http://www.w3.org/2005/Atom' xml : lang = 'en' / >
```

① First, let's define a namespace using a dictionary. Dictionary values are namespaces; dictionary keys are the prefix you specify. Using the None object as a prefix, we set the default namespace.

② When creating an element, we pass the lxml-specific nsmap argument used to pass namespace prefixes.

③ As expected, the serialization defines the default Atom namespace and declares one feed element without a namespace prefix.

④ Oops, we forgot to add the xml: lang attribute. Using the set () method, you can always add an attribute to any element. The method takes two arguments: the name of the attribute in the standard ElementTree format and the value of the attribute. (This method is also available in the ElementTree library. The only difference between lxml and ElementTree in this example is passing an nsmap argument to specify namespace prefixes.)

Are our documents limited to only one element? Of course not. We can easily create children.

```
>>> title = lxml. etree . SubElement ( new_feed , 'title' ,          ①
...     attrib = { 'type' : 'html' })                              ②
>>> print ( lxml. etree . tounicode ( new_feed ))                  ③
< feed xmlns = 'http://www.w3.org/2005/Atom' xml : lang = 'en' > < title type
= 'html' / > < / feed >
>>> title. text = 'dive into & hellip;'                            ④
>>> print ( lxml. etree . tounicode ( new_feed ))                  ⑤
< feed xmlns = 'http://www.w3.org/2005/Atom' xml : lang = 'en' > < title type
= 'html' > dive into & amp ; hellip ; < / title > < / feed >
>>> print ( lxml. etree . tounicode ( new_feed , pretty_print = True ))   ⑥
< feed xmlns = 'http://www.w3.org/2005/Atom' xml : lang = 'en' >
< title type = 'html' > dive into & amp ; hellip ; < / title >
< / feed >
```

① To create a subelement of an existing element, you need to create an object of the SubElement class. The parent element (in this case new_feed) and the name of the new element are passed to the class constructor. We do not re-declare the namespace for the child we are creating, since it inherits the namespace from the parent.

② We also pass a dictionary with attributes for the element. Dictionary keys are used as attribute names, and dictionary values are used as attribute values.

③ Unsurprisingly, a new title element was created in the Atom space and is a sub-element of the feed element. Since the title element has no text content or subelements, lxml serializes it as an empty element and closes it with />.

④ In order to add text content, we set the .text property.

⑤ The title element is now serialized with the text content just given. If the text contains characters "less than" <or "ampersand" ', then they must be escaped during serialization. Lxml handles such situations automatically.

⑥ When serializing, you can use "pretty printing", which inserts a line break after the end tag or start tag of elements with subelements but no text content. Technically, lxml adds "insignificant whitespace" to make the XML more readable.

> You might be interested in trying another third-party xmlwitch library that makes use of the Python with statement all over the place to make your XML creation code more readable.

Parsing Non-Integer XML

The XML specification dictates that all XML parsers must perform " draconian (strict) error handling". That is, if found in an XML document formal errors or *"wellformedness"* (*wellformedness*) analyzers must immediately interrupt the analysis and "break out." Well-formedness errors include inconsistent opening and closing tags, undefined elements, incorrect Unicode characters, and other esoteric situations. This error handling contrasts strongly with other well-known formats, for example, HTML - the browser does not stop rendering the web page if the HTML closing tag is forgotten in the page or the tag attribute value contains an unescaped ampersand. (There is a common misconception that HTML does not specify error handling. In fact, HTML error handling is well documented, but it is much more complex than just “stopping and catching fire on the first error.”)

Some people think (myself included) that it was a mistake on the part of the XML developers to force error handling so strictly. Don't get me wrong, I'm certainly in favor of simplifying the error handling rules. In practice, however, the notion of "well-formedness" turns out to be more insidious than it sounds, especially for XML documents that are published on the Internet and transmitted over HTTP (for example, Atom feeds). Despite the maturity of XML, which standardized draconian error handling in 1997, research consistently shows that a significant portion of Atom feeds on the Internet contain well-formedness errors.

So, I have both theoretical and practical reasons to process XML documents "at any cost", that is, not to stop and explode at the first error. If you find yourself in a similar situation, lxml can help.

Below is a fragment of a "broken" XML document.

```
<? xml version = '1.0' encoding = 'utf-8' ?>
<feed xmlns = 'http://www.w3.org/2005/Atom' xml: lang = 'en' >
```

```
  <title > dive into & hellip; </ title >
...
</ feed >
```

There is an error in the feed because the sequence & hellip; not defined in XML (it is defined in HTML). If you try to parse a broken feed with default settings, then lxml will stumble on an undefined occurrence of hellip.

```
>>> import lxml. etree
>>> tree = lxml. etree . parse ( 'examples / feed-broken.xml' )
Traceback ( most recent call last ) :
  File "<stdin>" , line 1 , in < module >
  File "lxml.etree.pyx" , line 2693 , in lxml. etree . parse ( src / lxml / lxml.
etree . c : 52591 )
  File "parser.pxi" , line 1478 , in lxml. etree ._parseDocument ( src / lxml /
lxml. etree . c : 75665 )
  File "parser.pxi" , line 1507 , in lxml. etree ._parseDocumentFromURL ( src
/ lxml / lxml. etree . c : 75993 )
  File "parser.pxi" , line 1407 , in lxml. etree ._parseDocFromFile ( src / lxml
/ lxml. etree . c : 75002 )
  File "parser.pxi" , line 965 , in lxml. etree ._BaseParser._parseDocFromFile
( src / lxml / lxml. etree . c : 72023 )
  File "parser.pxi" , line 539 , in lxml. etree
._ParserContext._handleParseResultDoc ( src / lxml / lxml. etree . c : 67830 )
  File "parser.pxi" , line 625 , in lxml. etree ._handleParseResult ( src / lxml /
lxml. etree . c : 68877 )
  File "parser.pxi" , line 565 , in lxml. etree ._raiseParseError ( src / lxml /
lxml. etree . c : 68125 )
lxml. etree . XMLSyntaxError : Entity 'hellip' not defined , line 3 , column 28
```

In order to process an XML document with errors, you need to create a new XML parser.

```
>> parser = lxml. etree . XMLParser ( recover = True )                ①
>>> tree = lxml. etree . parse ( 'examples / feed-broken.xml' , parser )   ②
>>> parser . error_log                                                ③
examples / feed-broken. xml : 3 : 28 : FATAL: PARSER:
ERR_UNDECLARED_ENTITY: Entity 'hellip' not defined
>>> tree. findall ( '{http://www.w3.org/2005/Atom#Title' )
[ < Element { http: // www. w3 . org / 2005 / Atom } title at ead510 > ]
```

```
>>> title = tree. findall ( '{http://www.w3.org/2005/AtomtTitle' ) [ 0 ]
>>> title. text                                                    ④
'dive into'
>>> print ( lxml. etree . tounicode ( tree. getroot ()))                ⑤
< feed xmlns = 'http://www.w3.org/2005/Atom' xml : lang = 'en' >
  < title > dive into < / title >
...
... [the rest of the serialization output has been omitted for brevity ]
...
```

① In order to create a new parser, we create a new class lxml.etree.XMLParser. Although it can take many different parameters , the only one of interest to us is the recovery argument recover. When set to True, lxml will go out of its way to recover from well-formedness errors.

② In order to parse the XML document with the new parser, we pass the parser object as the second argument to the parse () function. This time lxml does not throw an exception if the & hellip; sequence is undefined.

③ The analyzer contains messages about all found errors. (In fact, these messages persist regardless of the recover option.)

④ Since the analyzer does not know what to do with undefined & hellip;, it just throws out the word. The text content of the title element turns into 'dive into'.

⑤ Once again: after serialization, the sequence & hellip; disappeared, lxml threw it away.

It is important to note that **there is no guarantee of portability of error recovery** from XML parsers. Another analyzer might be smarter and recognize that & hellip; is a valid HTML sequence and restore it as an ampersand. Is it "better"? Maybe. Is this "more correct"? No, since both solutions are incorrect from the XML point of view. The correct behavior (according to the XML specification) is to stop processing and fire. If it is necessary not to follow the specification, then you do it at your own peril and risk.

Further reading

XML on Wikipedia

The ElementTree XML API (eng .)

Elements and Element Trees - Elements and trees elements (Eng .)

XPath Support in ElementTree - XPath Support in ElementTree (eng .)

The ElementTree iterparse Function - Function iterparse in the ElementTree (Eng .)

lxml (eng .)

Parsing XML and HTML with lxml - Processing XML and HTML in lxml (Eng .)

XPath and XSLT with lxml - XPath and XSLT in lxml (Eng .)

xmlwitch (eng .)

Serializing Python Objects

Immersion

At first glance, the idea behind serialization is simple. You have a data structure in memory that you want to store, reuse, or send to someone else. How do you do it? It depends on how you save it, how you want to use it, and who you want to send it to. Many games allow you to save your progress before exiting and resume the game after launch. (In general, many non-gaming applications also allow you to do this). In this case, the structure that stores your progress in the game must be saved to disk when you close the game and loaded from disk when you launch it. The data is intended only for use by the same program that created it, is never sent over the network, and is never read by anything other than the program that created it. Therefore, compatibility issues are limited so that later versions of the program can read data created by earlier versions.

For such cases, the pickle module is ideal. It is part of the Python standard library, so it is always available. It's fast, most of it is written in C, just like the Python interpreter itself. It can store completely arbitrary complex Python data structures.

What can pickle store?

- All of Python's built-in data types: boolean, Integer, floating point, complex numbers, strings, bytes objects, byte arrays, and

None.

- Lists, tuples, dictionaries, and sets containing any combination of built-in data types
- Lists, tuples, dictionaries, and sets containing any combination of lists, tuples, dictionaries, and sets containing any combination of built-in data types (and so on, up to the maximum nesting level Python supports).
- Functions, classes and class instances (with caveats).

If this is not enough for you, then the pickle module is also extensible. If you are interested in this feature, see the links in the "Further Reading" section at the end of this chapter.

A small note on the examples in this chapter.

This part is about two Python consoles. All of the examples in this chapter are part of one larger story. You will need to switch back and forth between the two consoles to demonstrate the pickle and json modules.

In order not to get confused, open the Python console and define the following variable:

>>> shell = 1

Leave this window open. And open another Python console and define the following variable:

>>> shell = 2

In this chapter, I will use the shell variable to indicate which Python console I am using in each example.

Saving data to a Pickle file.

The Pickle module works with data structures. Let's create one .

>>> shell 1 ① >>> entry = {} ② >>> entry ['title'] = 'Dive into history, 2009 edition' >>> entry ['article_link'] = 'http://diveintomark.org / archives / 2009/03/27 / dive-into-history-2009-edition ' >>> entry [' comments_link '] = None >>> entry [' internal_id '] = b ' \ x DE \ x D5 \ x B4 \ x F8 ' >>> entry [' tags '] = (' diveintopython ' , ' docbook ' , ' html ') >>> entry [' published '] = True >>> import time >>> entry [' published_date '] = time . strptime ('Fri Mar 27 22:20:42 2009') ③ >>> entry ['published_date'] time . struct_time (tm_year = 2009 , tm_mon = 3 , tm_mday = 27 , tm_hour

= 22 , tm_min = 20 , tm_sec = 42 , tm_wday = 4 , tm_yday = 86 , tm_isdst = - 1)

① Everything further happens in the Python # 1 console.

② The idea is to create a dictionary that will represent something useful, such as an Atom mailing list item. I also want to make sure it contains several different data types to expose the pickle module. Don't read too much into these variables.

③ The time module contains a data structure (struct_time) to represent a point in time (down to milliseconds) and functions for working with these structures. The strptime () function takes a formatted string as input and converts it to struct_time. This string is in standard format, but you can control it with format codes. For a more detailed description, see the time module.

We now have a wonderful dictionary. Let's save it to a file.

>>> shell ① 1 >>> import pickle >>> with open ('entry.pickle' , 'wb') as f: ② ... pickle . dump (entry , f) ③ ...

① We're still in the first console

② Use the open () function to open the file. Set the file mode to 'wb' in order to open the file for writing in binary mode. Let's wrap it in a with clause to

make sure that the file is closed automatically when you finish working with it.

③ The dump () function of the pickle module takes a Python serializable data structure, serializes it to a binary, Python-dependent format uses the latest pickle protocol and saves it to an open file.

The last sentence was very important.

- The pickle protocol depends on Python; there is no guarantee of compatibility with other languages. You may not be able to take the entry.pickle file you just made and use it in any way with Perl, PHP, Java or any other programming language.
- Not every Python data structure can be serialized by the Pickle module. The pickle protocol has changed several times with the addition of new data types to the Python language, and it still has limitations.
- As a result, there is no guarantee of compatibility between different Python versions. Newer versions of Python support the old serialization formats, but older versions of Python do not support the new formats (because they do not support the new data formats)
- Unless you indicate otherwise, the pickle module functions will use the latest pickle protocol. This is to make sure you have the most flexibility in the types of data you can serialize, but it also means that the resulting file will not be readable with older versions of Python that do not support the latest pickle protocol.
- The latest version of the pickle protocol is binary. Make sure to open pickle files in binary mode, or the data will get corrupted when written.

Loading data from file pickle.

Now switch to the second Python console - that is , not the one where you created the entry dictionary.

>>> shell ① 2 >>> entry ② Traceback (most recent call last) : File " <stdin>" , line 1 , in < module > NameError : name 'entry' is not defined >>> import pickle >>> with open ('entry.pickle' , 'rb') as f: ③ ... entry = pickle . load (f) ④ ... >>> entry ⑤ { 'comments_link' : None , 'internal_id' : b ' \ x DE \ x D5 \ x B4 \ x F8' , 'title' : 'Dive into history, 2009 edition ' , ' tags ' : (' diveintopython ' , ' docbook ' , ' html ') , ' article_link ' : '

http://diveintomark.org/archives/2009/03/27/dive-into-history-2009- edition ' , ' published_date ' : time . struct_time (tm_year = 2009 , tm_mon = 3 , tm_mday = 27 , tm_hour = 22 , tm_min = 20 , tm_sec = 42 , tm_wday = 4 , tm_yday = 86 , tm_isdst = - 1) , 'published' : True }

① This is the second Python console

② The entry variable is not defined here. You defined the entry variable in the first Python console, but this is a completely different environment with its own state.

③ Open the entry.pickle file you created in the first Python console. The pickle module uses binary data format, so you always need to open the file in binary mode.

④ The pickle.load () function takes a stream as input, reads serialized data from the stream, creates a new Python object, restores the serialized data to a new Python object, and returns a new Python object.

⑤ The entry variable is now a dictionary with familiar keys and values.

The result of the pickle.dump () / pickle.load () loop is a new data structure

equivalent to the original data structure.

>>> shell ① 1 >>> with open ('entry.pickle' , 'rb') as f: ② ... entry2 = pickle . load (f) ③ ... >>> entry2 == entry ④ True >>> entry2 is entry ⑤ False >>> entry2 ['tags'] ⑥ ('diveintopython' , 'docbook' , 'html') >> > entry2 ['internal_id'] b ' \ x DE \ x D5 \ x B4 \ x F8'

① Switch back to the first Python console.

② Open the entry.pickle file

③ Load serialized data into new variable entry2

④ Python confirms that the two dictionaries (entry and entry2) are equivalent. In this console, you created an entry from scratch by manually assigning values to the keys from an empty dictionary. You have serialized this dictionary and saved it in your entry.pickle file. Now you have read the serialized data from this file and created a perfect copy of the original structure.

⑤ Equivalence does not mean identity. I said that you created a _ideal copy_ of the original data structure, and that's true. But it's still a copy.

⑥ For reasons that will become clear later, I want to indicate that the values for the 'tags' key are a tuple and the 'internal_id' value is a bytes object.

> Many articles on the Pickle module link to cPickle. There are two implementations of the pickle module in Python 2, one written in pure Python and the other in C (but still callable from Python). In Python 3, these two modules were merged, so you should always

use import pickle. You may find these articles helpful, but you should ignore the outdated cPickle information.

Using Pickle without files

The example from the previous section showed how to serialize an object directly to a file on disk. But what if you don't need it or you didn't want to use the file? You can serialize to a bytes object in memory.

```
>>> shell
1
>>> b = pickle . dumps ( entry )  ①  >>> type ( b )  ②  < class 'bytes' > >>>
entry3 = pickle . loads ( b )  ③  >>> entry3 == entry  ④  True
```

① The pickle.dumps () function (note the 's' at the end of the function name) does the same serialization as the pickle.dump () function. Instead of taking a stream as input and writing serialized data to disk, it simply returns the serialized data.

② Since the pickle protocol uses a binary data format, pickle.dumps () returns a bytes object.

③ The pickle.loads () function (again note the 's' at the end of the function name) does the same deserialization as the pickle.load () function. But instead of accepting a stream as input and reading serialized data from a file, it accepts a bytes object containing serialized data, such as returned by the pickle.dumps () function.

④ The end result is the same: a perfect copy of the original dictionary.

Bytes and lines rearing their ugly heads again

The pickle protocol has been around for many years, and it has evolved with Python itself. There are currently four different versions of the pickle protocol.

- Python 1.x spawned two versions of the protocol, text based format (version 0) and binary format (version 1)
- Python 2.3 introduced the new pickle protocol (version 2) in order

to support new functionality in Python classes. It is binary.

- Python 3.0 introduced another pickle protocol (version 3) with full support for bytes and byte arrays. It is also binary.

Wow look, the difference between strings and bytes rears its ugly head again. (If you're surprised, you haven't paid enough attention.) In practice, this means that while Python 3 can read data stored using protocol version 2, Python 2 cannot read data saved using protocol version 3.

Debugging pickle files

What does the pickle protocol look like? Let's put the python console aside for a moment and take a look at the entry.pickle file we created. To an unarmed eye, it looks like gibberish.

you @ localhost: ~ / diveintopython3 / examples $ ls -l entry.pickle -rw-r - r-- 1 you you 358 Aug 3 13 : 34 entry.pickle you @ localhost: ~ / diveintopython3 / examples $ cat entry. pickle comments_linkqNXtagsqXdiveintopythonqXdocbookqXhtmlq? qX publishedq? XlinkXJhttp: // diveintomark.org / archives / 2009 / 03 / 27 / dive-Into-history- 2009 -edition q Xpublished_dateq ctime struct_time qRqXtitleqXDive Into? History , 2009 editionqu.

Not very helpful. You can see the strings, but the rest of the data types look like non-printable (or at least non-readable) characters. The fields are not even separated by even tabs or spaces. This is not the format you would like to manually debug.

>>> shell
1
>>> import pickletools >>> with open ('entry.pickle' , 'rb') as f: ... pickletools . dis (f) 0 : \ x80 PROTO 3 2 : } EMPTY_DICT 3 : q BINPUT 0

5 : (MARK 6 : X BINUNICODE 'published_date' 25 : q BINPUT 1 27 : c GLOBAL 'time struct_time' 45 : q BINPUT 2 47 : (MARK 48 : M BININT2 2009 51 : K BININT1 3 53 : K BININT1 27 55 : K BININT1 22 57 : K BININT1 20 59 : K BININT1 42 61 : K BININT1 4 63 : K BININT1 86 65 : J BININT - 1 70 : t TUPLE (MARK at 47) 71 : q BINPUT 3 73 : } EMPTY_DICT 74 : q BINPUT 4 76 : \ x86 TUPLE2 77 : q BINPUT 5 79 : R REDUCE 80 : q BINPUT 6 82 : X BINUNICODE 'comments_link' 100 : q BINPUT 7 102 : N NONE 103 : X BINUNICODE 'internal_id' 119 : q BINPUT 8 121 : C SHORT_BINBYTES 'ÞÕ´ø' 127 : q BINPUT 9 129 : X BINUNICODE 'tags' 138 : q BINPUT 10 140 : X BINUNICODE ' diveintopython ' 159 : q BINPUT 11 161 : X BINUNICODE ' docbook ' 173 : q BINPUT 12 175 : X BINUNICODE ' html ' 184 : q BINPUT 13 186 : \ x87 TUPLE3 187 : q BINPUT 14 189 : X BINUNICODE ' title ' 199 : q BINPUT 15 201 : X BINUNICODE 'Dive into history, 2009 edition' 237 : q BINPUT 16 239 : X BINUNI CODE 'article_link' 256 : q BINPUT 17 258 : X BINUNICODE 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition' 337 : q BINPUT 18 339 : X BINUNICODE ' published ' 353 : q BINPUT 19 355 : \ x88 NEWTRUE 356 : u SETITEMS (MARK at 5) 357 :. STOP highest protocol among opcodes = 3

The most interesting piece of information in the disassembler is on the last line, because it includes the version of the protocol with which the file was saved. There is no explicit pickle protocol token. To determine which version of the protocol was used to save the Pickle file, you need to look into the markers ("opcodes") inside the saved data and use the hardcoded information about which tokens were entered in which version of the Pickle protocol. The pickletools.dis () function does exactly that, and it prints the result on the last line of disassembled output. Here is a function that only returns the version number, no output:

import pickletools def protocol_version (file_object) : maxproto = - 1 for opcode , arg , pos in pickletools . genops (file_object) : maxproto = max (maxproto , opcode. proto) return maxproto And here it is in action : >>> import pickleversion >>> with open ('entry.pickle' , 'rb') as f: .. . v = pickleversion. protocol_version (f) >>> v 3

Serializing Python Objects for Reading with Other Languages

The data format used by the pickle module is Python dependent. It doesn't try

to be compatible with other programming languages. If cross-language compatibility is among your needs, you should look at serialization formats. One such format is JSON. JSON is an acronym for JavaScript Object Notation, but don't let the name fool you - JSON was certainly designed to be used by many programming languages.

Python 3 includes the json module in the standard library. Like the pickle module, the json module has functions to serialize data structures, save serialized data to disk, load serialized data from disk, and deserialize data back into a new Python object. There are also several important differences. First, the json data format is textual, not binary. RFC 4627 defines the json format and how different data types should be converted to text. For example, a boolean value is stored as a five character string 'false' or a four character string 'true'. All values in json are case sensitive.

In - Second, as with any text format, there is the problem of gaps. JSON allows you to insert an arbitrary number of spaces (tabs, newlines, and blank lines) between values. The spaces in it are "insignificant", which means JSON encoders can add as much or as little white space as they like, and JSON decoders will ignore spaces between values. This allows you to use pretty-print output to display your data in JSON format, conveniently displaying nested values with different indentation levels so that you can read everything in a standard viewer or text editor. The json module in Python has beautiful output options while encoding data.

In - Third, there is a long-standing problem sets. JSON stores values as plain text, but as you know, there is no such thing as "plain text". JSON must be stored in Unicode encoding (UTF-32, UTF-16, or standard UTF-8), and section 3 of RFC 4627 defines how to specify the encoding to use.

Saving data to a JSON file

JSON looks remarkably similar to a data structure that you could define manually in JavaScript. This is no coincidence, you can actually use the JavaScript eval () function to "decode" data serialized to json. (The usual protests against untrusted input are accepted, but the point is that json is valid JavaScript.) On the merits , JSON may be already well familiar to you .

>>> shell
1
>>> basic_entry = {} ① >>> basic_entry ['id'] = 256 >>> basic_entry [

'title'] = 'Dive into history, 2009 edition' >>> basic_entry ['tags'] = ('diveintopython' , 'docbook' , 'html') >>> basic_entry ['published'] = True >>> basic_entry ['comments_link'] = None >>> import json >>> with open ('basic .json ' , mode = ' w ' , encoding = ' utf-8 ') as f: ② ... json . dump (basic_entry , f) ③

① We are going to create a new data structure instead of using the existing entry data structure. Later in this chapter, we will see what happens when we try to encode a more general data structure into JSON.

② JSON is a text format, which means you must open the file in text mode and specify the encoding. You can never go wrong with UTF-8.

③ Like the pickle module, the json module defines a dump () function that takes a Python data structure and a stream to write as input. The dump () function serializes a Python data structure and writes it to a stream object. Since we do this in the with clause, we can be sure that the file will be closed correctly when we finish working with it.

Well, what does the result of serialization in json format look like?

you @ localhost: ~ / diveintopython3 / examples $ cat basic.json
{ "published" : true , "tags" : ["diveintopython" , "docbook" , "html"] ,
"comments_link" : null,
"id" : 256 , "title" : "Dive into history, 2009 edition" }

This is undoubtedly much more readable than the pickle file. But json can contain any number of spaces between values, and the json module provides an easy way to create an even more readable json file.

>>> shell
1
>>> with open ('basic-pretty.json' , mode = 'w' , encoding = 'utf-8') as f: ...

json . dump (basic_entry , f , indent = 2) ①

① If you pass the ident parameter to the Json function . The dump () it will make the resulting file json more readable in damage to the size of the file . The ident parameter is an integer. 0 means "put each value on a separate line". A number greater than 0 means "put each value on a separate line, and use number of spaces to indent nested data structures."

And here's the result :

you @ localhost: ~ / diveintopython3 / examples $ cat basic-pretty.json
{ "published" : true , "tags" : ["diveintopython" , "docbook" , "html"] , "comments_link" : null, "id" : 256 , "title" : "Dive into history, 2009 edition" }

Python to JSON data type mapping

Since JSON was not designed for Python only, there are some shortcomings in the coverage of Python data types. Some of them are just differences in type names, but there are two important Python data types that are completely overlooked. Let's see if you can spot them:

Remarks	Json	PYTHON 3
	object	dictionary
	array	list
	string	string
	integer	integer
	real number	float
*	true	True

*	false	False
*	null	None
*	Cell text	Cell text

- - All variables in JavaScript are case sensitive

Have you noticed what's lost? Tuples and bytes! JSON has a type, an array, which the JSON module maps to a list in Python, but there is no separate type for "static arrays" (tuples). And also JSON has good support for strings, but no support for bytes or byte arrays. ...

Serializing data types not supported by JSON

Just because JSON doesn't have built-in support for bytes doesn't mean you can't serialize bytes. The json module provides extensible hooks for encoding and decoding unknown data types. (By "unknowns" I mean "not defined in json." Obviously, the json module knows about byte arrays, but it was built with the limitations of the json specification in mind). If you want to encode bytes or other data types that json does not support, you need to provide specific encoders and decoders for those data types.

>>> shell
1
>>> entry ① { 'comments_link' : None , 'internal_id' : b ' \ x DE \ x D5 \ x B4 \ x F8' , 'title' : 'Dive into history, 2009 edition' , 'tags' : ('diveintopython' , 'docbook' , 'html') , 'article_link' : 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition' , 'published_date' : time . struct_time (tm_year = 2009 , tm_mon = 3 , tm_mday = 27 , tm_hour = 22 , tm_min = 20 , tm_sec = 42 , tm_wday = 4 , tm_yday = 86 , tm_isdst = - 1) , 'published' : True } >>> import json >>> with open ('entry.json' , 'w' , encoding = 'utf-8') as f: ② ... json . dump (entry , f) ③ ... Traceback (most recent call last) : File "<stdin>" , line 5 , in < module > File "C: \ P ython31 \ l ib \ j son \ _ _init__.py " , line 178 , in dump for chunk in iterable: File " C: \ P ython31 \ l ib \ j son \ e ncoder.py " , line 408 , in _iterencode for chunk in _iterencode_dict (o , _current_indent_level) : File " C : \ P ython31 \ l ib \ j son \ e ncoder.py " , line 382 , in _iterencode_dict for chunk in chunks: File " C: \ P ython31 \ l ib \ j son \ e ncoder.py " , line 416 , in _iterencode o = _default (o) File "C: \ P ython31 \ l ib \ j son \ e ncoder.py" , line 170 , in default raise TypeError (repr (o) + "is not JSON serializable") TypeError : b ' \ x DE \ x D5 \ x B4 \

x F8' is not JSON serializable

① Okay, now is the time to revisit the entry data structure. Everything is there: boolean values, an empty value, a string, a tuple of strings, a bytes object, and a structure that stores time.

② I know I said this earlier, but I will repeat again: json is a text format. Always open json files in text mode with utf-8 encoding.

③ Well ... _THIS_ is not good. What happened?

Here's the thing: the json.dump () function tried to serialize the bytes object b '\ xDE \ xD5 \ xB4 \ xF8', but it failed because there is no support for bytes objects in json. However, if storing such objects is important to you, you can define your own "mini serialization format".

```
def to_json ( python_object ) :  ①  if isinstance ( python_object , bytes ) :  ②
return { '__class__' : 'bytes' , '__value__' : list ( python_object )}  ③  raise
TypeError ( repr ( python_object ) + 'is not JSONizable' ) ④
```

① To define your own "mini serialization format" for data types that json does not support out of the box, simply define a function that takes a Python object as a parameter. This Python object will be exactly the object that the json.dump () function cannot serialize itself - in this case, it is a bytes object b '\ xDE \ xD5 \ xB4 \ xF8'

② Your specific serialization function should check the type of the Python objects passed to it by the json.dump () function. This is not necessary if your function only serializes one datatype, but it makes it crystal clear which case the function is covering and makes it easier to improve the function if you need to serialize more datatypes later.

③ In this case, I decided to convert the bytes object to a dictionary. The __class__ key will contain the name of the original data type, and the __value__ key will hold the value itself. Of course, it cannot be objects of type bytes, so you need to convert it to something serializable using json. Bytes objects are just a sequence of numbers, each number will be anywhere from 0 to 255. We can use the list () function to convert a bytes object to a list of numbers. So b '\ xDE \ xD5 \ xB4 \ xF8' becomes [222, 213, 180, 248]. (Count it! It works! The \ xDE byte in hex is 222 in decimal, \ xD5 is 213, and so on.)

④ This line is important. The data structure that you are serializing can contain data types that are not in json and that your function does not handle. In such a case, your handler must raise a TypeError so the json.dump () function knows that your handler was unable to recognize the data type.

This is it, you don't need anything else. Indeed, the handler function you defined returns a Python dictionary, not a string. You don't write the serialization to json entirely yourself, you just do the conversion-to-a-supported data type. Json.dump () will do the rest for you.

>>> shell
1
>>> import customserializer ① >>> with open ('entry.json' , 'w' , encoding = 'utf-8') as f: ② ... json . dump (entry , f , default = customserializer. to_json) ③ ... Traceback (most recent call last) : File "<stdin>" , line 9 , in < module > json. dump (entry , f , default = customserializer. to_json) File "C: \ P ython31 \ l ib \ j son \ _ _init__.py" , line 178 , in dump for chunk in iterable: File "C: \ P ython31 \ l ib \ j son \ e ncoder.py " , line 408 , in _iterencode for chunk in _iterencode_dict (o , _current_indent_level) : File " C: \ P ython31 \ l ib \ j son \ e ncoder.py " , line 382 , in _iterencode_dict for chunk in chunks: File "C: \ P ython31 \ l ib \ j son \ e ncoder.py" , line 416 , in _iterencode o = _default (o) File "/ Users / pilgrim / diveintopython3 / examples / customserializer .py " , line 12 , in to_json raise TypeError (repr (python_object) + 'is not JSON serializable') ④ TypeError : time . struct_time (tm_year = 2009 , tm_mon = 3 , tm_mday = 27 , tm_hour = 22 , tm_min = 20 , tm_sec = 42 , tm_wday = 4 , tm_yday = 86 , tm_isdst = - 1) is not JSON serializable

① The customserializer module is where you just defined the to_json ()

function in the previous example

② Text mode, utf-8, tra-la-la. (You will forget! I sometimes forget! And everything works great, until one moment it breaks, and then it starts breaking even more theatrically)

③ This is the important piece: to embed your conversion handler function in json.dump (), pass your function to json.dump () in the default parameter. (Hooray, everything in Python is an object!)

④ Great, it really works. But look at the exception. Now the json.dump () function no longer complains about not being able to serialize the bytes object. Now she's complaining about a completely different object: time.struct_time.

While getting another exception doesn't look like progress, it actually is. You just need to add a couple of lines of code for this to work.

import time def to_json (python_object) : if isinstance (python_object , time . struct_time) : ① return { '__class__' : 'time.asctime' , '__value__' : time . asctime (python_object)} ② if isinstance (python_object , bytes) : return { '__class__' : 'bytes' , '__value__' : list (python_object)} raise TypeError (repr (python_object) + 'is not JSON serializable')

① By adding customserializer.to_json () to the existing function, we have to check that the Python object (with which the json.dump () function has problems) is actually time.struct_time.

② If so, we'll do something similar to the conversion we did with the bytes object: convert the time.struct_time object into a dictionary that only contains data types that can be serialized to json. In this case, the simplest way to convert a date to a value that can be serialized in json is to convert it to a

string using the time.asctime () function. The time.asctime () function converts the disgusting looking time.struct_time to the string 'Fri Mar 27 22:20:42 2009'.

With these two special conversions, the entry data structure should serialize in its entirety without any problem.

>>> shell
1
>>> with open ('entry.json' , 'w' , encoding = 'utf-8') as f: ... json . dump (entry , f , default = customserializer. to_json) ...

you @ localhost: ~ / diveintopython3 / examples $ ls -l example.json -rw-r -r-- 1 you you 391 Aug 3 13 : 34 entry.json you @ localhost: ~ / diveintopython3 / examples $ cat example. json { "published_date" : { "__class__" : "time.asctime" , "__value__" : "Fri Mar 27 22:20:42 2009" } , "comments_link" : null, "internal_id" : { "__class__" : " bytes " , " __value__ " : [222 , 213 , 180 , 248]} , " tags " : [" diveintopython " , " docbook " , " html "] , " title " : " Dive into history, 2009 edition " , " article_link " : " http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition " , " published " : true }

Loading data from json file

As in the pickle module, the json module has a load () function that takes a stream as input, reads json data from it and creates a new Python object that will be a copy of the data structure written in the json file.

>>> shell
2
>>> del entry ① >>> entry Traceback (most recent call last) : File "

<stdin>" , line 1 , in < module > NameError : name 'entry' is not defined >>> import json >>> with open ('entry.json' , 'r' , encoding = 'utf-8') as f: ... entry = json. load (f) ② ... >>> entry ③ { 'comments_link' : None , 'internal_id' : { '__class__' : 'bytes' , ' __value__ ' : [222 , 213 , 180 , 248]} , ' title ' : ' Dive into history, 2009 edition ' , ' tags ' : [' diveintopython ' , ' docbook ' , ' html '] , ' article_link ' : ' http://diveintomark.org/archives/2009/03/27/ dive-into-history-2009-edition ' , ' published_date ' : { ' __class__ ' : ' time.asctime ' , ' __value__ ' : ' Fri Mar 27 22:20:42 2009 ' } , ' published ' : True }

① For demonstration, switch to the second Python console and remove the entry data structure you created earlier in this chapter using the Pickle module.

② In the simplest case, the json. load () works the same as pickle function . load () . You pass it a stream object, and the result is a new Python object.

③ I have good news and bad news. The good news is that the json. load () successfully read the entry.json file you created in the first Python console and created a new Python object that contains the data. Now for the bad news: it didn't recreate the original entry data structure. The two values 'internal_id' and 'published_date' were created as dictionaries - namely, as

dictionaries with json-compatible values (which is what you created in the to_json () conversion function)

Function json. load () doesn't know anything about the transform function you might have passed to json. dump () . Now you need to create a function that is the opposite of to_json () - a function that will take a selectively converted json object and convert it back to the original Python object.

add this to customserializer.py
def from_json (json_object) : ① if '__class__' in json_object: ② if json_object ['__class__'] == 'time.asctime' : return time . strptime (json_object ['__value__']) ③ if json_object ['__class__'] == 'bytes' : return bytes (json_object ['__value__']) ④ return json_object

① This transform function also takes one parameter and returns one value. But the parameter it takes is not a string, it's a Python object - the result of deserializing the JSON string into which the Python object was converted.

② All you need to do is check if the given object contains the key '__class__' that was created by the to_json () function . If so, then the value found by this key will tell you how to decode this object back into the original Python object.

③ To decode the time string returned by the time function . asctime () you need to use the time function . strptime () . This function takes a formatted time string (to define the format, but by default this format matches the format of time . Asctime ()) and returns a time . struct_time

④ To convert a list of numbers back to bytes objects you can use the bytes () function

That's all, there are only two data types that were processed by the to_json () function and now the same data types were processed by the from_json () function . Here's the result :

>>> shell
2
>>> import customserializer >>> with open ('entry.json' , 'r' , encoding = 'utf-8') as f: ... entry = json. load (f , object_hook = customserializer. from_json) ① ... >>> entry ② { 'comments_link' : None , 'internal_id' : b ' \ x DE \ x D5 \ x B4 \ x F8' , 'title' : 'Dive into history, 2009 edition' , 'tags' : [' diveintopython ' , ' docbook ' , ' html '] , ' article_link ' : ' http://diveintomark.org/archives/2009/03/27/dive- into-history-2009-edition ' , ' published_date ' : time . struct_time (tm_year = 2009 , tm_mon = 3 , tm_mday = 27 , tm_hour = 22 , tm_min = 20 , tm_sec = 42 , tm_wday = 4 , tm_yday = 86 , tm_isdst = - 1) , 'published' : True }

① To inline the from_json () function in the deserialization process, pass it in the object_hook parameter in the json function call . load () . Functions that take functions as convenient!

② The entry data structure now contains the 'internal_id' key with a bytes value. It also contains the key 'published_date' with the value time . struct_time .

There is one more glitch, though.

>>> shell
1
>>> import customserializer >>> with open ('entry.json' , 'r' , encoding = 'utf-8') as f: ... entry2 = json. load (f , object_hook = customserializer. from_json) ... >>> entry2 == entry ① False >>> entry ['tags'] ② ('diveintopython' , 'docbook' , 'html') >>> entry2 ['tags'] ③ [

'diveintopython' , 'docbook' , 'html']

① Even after embedding to_json () into serialization and from_json () into deserialization, we still haven't received a complete copy of the original data structure. Why not?

② In the original entry data structure, the 'tags' value was a tuple of strings.

③ But in the recreated data structure entry2, the value for the 'tags' key is a list of strings. JSON doesn't differentiate between tuples and lists, it only has one list-like data type, an array, and the json module quietly converts both lists and tuples to json arrays during serialization. For most cases, you can ignore the distinction between lists and tuples, but this is something to remember when working with the json module.

Further reading

- About the pickle module:
  - PEP 238: Modifying the Division Operator
- About JSON format and json module:
  - json - JavaScript Object Notation Serializer
  - JSON encoding and ecoding with custom objects in Python

HTTP and web services

Immersion

HTTP web services are software-based methods of transmitting and receiving

data from remote servers using nothing but HTTP operations. If you want to get data from the server use HTTP GET; if you want to send data to server use HTTP POST. The more advanced functions of the web service HTTP API allow you to create, modify, and delete data using HTTP PUT and HTTP DELETE. In other words, the "verbs" embedded in the HTTP protocol (GET, POST, PUT and DELETE) can be mapped to application-level operations to retrieve, create, modify, and delete data.

HTTP web services are programmatic ways of sending and receiving data from remote servers using nothing but the operations of HTTP. If you want to get data from the server, use HTTP GET; if you want to send new data to the server, use HTTP POST. Some more advanced HTTP web service APIs also allow creating, modifying, and deleting data, using HTTP PUT and HTTP DELETE. In other words, the "verbs" built into the HTTP protocol (GET, POST, PUT, and DELETE) can map directly to application-level operations for retrieving, creating, modifying, and deleting data.

The main advantage of this approach is simplicity, and this simplicity has proven to be popular. Data - usually XML or JSON - can be built or stored statically, or dynamically generated by a server-side script, and all modern languages (including Python, of course!) Include an HTTP library to load it. Debugging is easy too; because every resource in an HTTP webserver has a unique address (in the form of a URL), you can load it into your web browser and see the raw data immediately.

The main advantage of this approach is simplicity, and its simplicity has proven popular. Data - usually XML or JSON - can be built and stored statically, or generated dynamically by a server-side script, and all major programming languages (including Python, of course!) Include an HTTP library for downloading it. Debugging is also easier; because each resource in an HTTP web service has a unique address (in the form of a URL), you can load it in your web browser and immediately see the raw data.

Examples of HTTP web services:

* Google Data APIs allow you to interact with a wide range of Google services, including Blogger and YouTube.

* Flickr Services allows you to upload and download photos to Flickr.
* Twitter API allows you to post status updates to Twitter.
* and many others

Examples of HTTP web services:

* Google Data APIs allow you to interact with a wide variety of Google services, including Blogger and YouTube.
* Flickr Services allow you to upload and download photos from Flickr.
* Twitter API allows you to publish status updates on Twitter.
*… And many more

Python3 has two libraries for interacting with HTTP web services: http.client is a low-level library that implements RFC 2616 - the HTTP protocol. urllib.request is an abstraction layer built on top of http.client. It provides a standard API for accessing HTTP and FTP servers, automatically follows HTTP redirects (redirects), and can work with some common forms of HTTP authentication.

Python 3 comes with two different libraries for interacting with HTTP web services: http.client is a low-level library that implements RFC 2616 , the HTTP protocol. urllib.request is an abstraction layer built on top of http.client. It provides a standard API for accessing both HTTP and FTP servers, automatically follows HTTP redirects, and handles some common forms of HTTP authentication.

So which one should you use? None. Better to use httplib2 instead , an open source third party library. It implements HTTP more completely than http.client and at the same time provides a better abstraction than urllib.request.

So which one should you use? Neither of them. Instead, you should use httplib2, an open source third-party library that implements HTTP more fully than http.client but provides a better abstraction than urllib.request.

To understand why httplib2 should be your choice, you first need to

understand the HTTP protocol.

To understand why httplib2 is the right choice, you first need to understand HTTP.

14.2 HTTP Features

14.2 Features of HTTP

There are five important features that all HTTP clients must support.

There are five important features which all HTTP clients should support.

14.2.1 Caching

14.2.1 Caching

The most important thing to understand about any web service is that accessing the web is incredibly expensive. I don't mean "dollars and cents" (although bandwidth isn't free). What I'm talking about is that it takes a very long time to open a connection, send a request, and receive a response from a remote server. Even on a fast broadband connection, the latency (the time it takes to send a request and start receiving data back) may still be higher than you expected. Router errors, lost packets, hacker attacks on intermediate proxies - there is not a single quiet minute on the Internet , and nothing can be done about it.

The most important thing to understand about any type of web service is that network access is incredibly expensive. I don't mean "dollars and cents" expensive (although bandwidth ain't free). I mean that it takes an extraordinary long time to open a connection, send a request, and retrieve a response from a remote server. Even on the fastest broadband connection, latency (the time it takes to send a request and start retrieving data in a response) can still be higher than you anticipated. A router misbehaves, a packet is dropped, an intermediate proxy is under attack - there's never a dull moment on the public internet, and there may be nothing you can do about it.

HTTP was designed with caching in mind. There is a whole class of devices (called "caching proxies") whose only job is to stay between you and the rest of the world and minimize network traffic. Your company or your ISP almost certainly has caching proxies, even if you don't know about it. Their work is based on caching built into the HTTP protocol.

HTTP is designed with caching in mind . There is an entire class of devices (called "caching proxies") whose only job is to sit between you and the rest of the world and minimize network access. Your company or ISP almost certainly maintains caching proxies, even if you're unaware of them. They work because caching built into the HTTP protocol.

Here's a concrete example of how caching works . You have visited diveintomark.org using your browser. This site has a picture wearehugh.com/m.jpg. When your browser loads this image, the server includes the following HTTP headers in its response:

Here's a concrete example of how caching works. You visit diveintomark.org in your browser. That page includes a background image, wearehugh.com/m.jpg. When your browser downloads that image, the server includes the following HTTP headers:

```
HTTP / 1.1 200 OK
Date: Sun, 31 May 2009 17:14:04 GMT
Server: Apache
Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT
ETag: "3075-ddc8d800"
Accept-Ranges: bytes
Content-Length: 12405
Cache-Control: max-age = 31536000, public
Expires: Mon, 31 May 2010 17:14:04 GMT
Connection: close
Content-Type: image / jpeg
```

Example: porting chardet to Python 3

Content

Immersion

Question: what is the first reason for gibberish on the pages of the internet, in your mailbox, and in any computer system ever written? This is character encoding. In the chapter on strings, I talked about the history of encodings and the creation of Unicode, "one encoding that rules all." I would love it if I never saw gibberish on the web again, because all systems keep the correct encoding information, all communication protocols support Unicode, and every text system stuck to perfect fidelity when converting from one encoding to another.

I also love ponies.

Unicode pony.

Unicodoponi, so to speak.

I will focus on automatic character encoding detection.

What is automatic character encoding detection?

This means taking a sequence of bytes in an unknown encoding, and trying to figure out the encoding so that you can read the text. It's like hacking when you don't have the decryption key.

Is it possible?

In fact yes. However, some encodings are optimized for specific languages,

and the languages are not random. Some character sequences occur all the time, while others are very rare. A person who speaks English calmly, opening a newspaper and finding there “txzqJv 2! Dasd0a QqdKjvz” will immediately determine that it is not English (even if it consists entirely of English letters). By studying a large amount of "plain" text, a computer algorithm can simulate knowledge of the language and learn to make assumptions about the language of the text.

In other words, an encoding definition is actually a language definition combined with the knowledge of which language to use with which encoding.

Does such an algorithm exist?

Damn it yes! All major browsers have built-in autodetection of encoding, since the Internet is full of pages with no encoding specified at all. Mozilla Firefox includes an open source character encoding autodetection library. I ported it to Python2 and duplicated it in the chardet module. This chapter will walk you through the entire process of porting a chardet module from Python 2 to Python 3.

Introduction to the chardet module

Before I start porting the code, I'll help you understand how this code works! This is a quick tutorial on how to navigate the source code. The chardet library is too big to include here in its entirety, but you can download it from chardet.feedparser.org.

The entry point for the definition algorithm is universaldetector.py, which has one class, UniversalDetector. (You might think that the entry point is the detect function in chardet / __ init__.py, but it's actually just a convenience function that creates a UniversalDetector object, calls it, and returns its result)

There are five categories of encodings that UniversalDetector supports:

- UTF-N with a Byte Order Mark (BCH). This includes UTF-8, both large-Indian and small-Indian variants of UTF-16, and all 4 byte order-dependent UTF-32 variants.
- Escaped sequences that are fully compatible with 7-bit ASCII, where non-7-bit ASCII characters begin with an escape character. For example: ISO-2022-JP (Japanese) and HZ-GB-2312 (Chinese)
- Multibyte encodings, where each character is represented by a different number of bytes. For example: BIG5 (Chinese),

SHIFT_JIS (Japanese), and TIS-620 (Thai)

- One-byte encodings, where each character is represented by one byte. Narpimer: KOI8-R (Russian), WINDOWS-1266 (Hebrew), and TIS-620 (Thai)
- WINDOWS-1252, which is mainly used in Microsoft Windows by middle managers who do not want to think about character encoding while sitting in their burrow.

UTF-N with BCH

If the text begins with BCH, we can reasonably conclude that the text is encoded using UTF-8, UTF-16, or UTF-32 encoding. (MBP will tell us which one; that's what it serves.) This is supported by UniversalDetector, which will return the result immediately without any further research.

Escaped encodings

If the text contains a recognizable escaped sequence, then this can be an indicator of an escaped encoding, UniversalDetector will create an EscCharSetProber (defined in escprober.py) and submit the text for processing.

EscCharSetProber will create a series of state machines based on models HZ-GB-2312, ISO-2022-CN, ISO-2022-JP, and ISO-2022-KR (defined in escsm.py). EscCharSetProber will run the text through each state machine, byte-wise. If only one of the automata gives a positive check result, EscCharSetProber will immediately return it to UniversalDetector, which, in turn, will give it to the process that called it. If any of the state machines encounters an invalid sequence, it stops and processing continues with another state machine.

Multibyte encodings

Based on the BCH, UniversalDetector checks if the text contains high byte characters. If so, it creates a set of "explorers" to identify multibyte encodings, single byte encodings, and as a last resort windows-1252.

A researcher for multibyte encodings, MBCSGroupProber (defined in mbcsgroupprober.py) is actually just a console that controls a group of other researchers, one for each multibyte encoding: Big5, GB2312, EUC-TW, EUC-KR, EUC-JP, SHIFT_JIS, and UTF-8. MBCSGroupProber gives the text to each of these coding dependent researchers and checks the result. If an investigator reports that he found an invalid sequence, he is excluded from

further processing (so any subsequent calls to UniversalDetector.feed () will skip this investigator). If the investigator reports that he is confident enough that he has determined the encoding, MBCSGroupProber reports a positive result to UniversalDetector, which passes the result to the calling process.

Most researchers on multibyte encodings inherit from MultiByteCharSetProber (defined in mbcharsetprober.py) and simply include a suitable state machine and allocation parser and MultiByteCharSetProber does the rest. MultiByteCharSetProber passes the text through the encoding-dependent state machine byte-by-byte to find a sequence of bytes that would indicate a positive or negative result. At the same time, MultiByteCharSetProber passes the text through a charset-dependent allocation parser.

The distribution analyzer (each defined in chardistrubution.py) uses a model that specifies which characters are more common in which language. Once the MultiByteCharSetProber has returned enough text to parse, a similarity score is calculated based on the number of frequently used characters, the total number of characters, and a language-specific distribution factor. If the confidence is high enough, MultiByteCharSetProber returns the result to MBCSGroupProber, which returns the result to UniversalDetector, which in turn returns the result to the process that called it.

Japanese is the hardest thing to do. A single-character allocation parser is not always sufficient to distinguish between EUC-JP and SHIFT_JIS, so SJISProber (defined in sjisprober.py) also uses a 2-character allocation parser. SJISContextAnalysis and EUCJPContextAnalysis (both defined in jpcntx.py and both inherited from the JapaneseContextAnalysis class) check the repetition rate of Hirogana characters in the text. Once enough text has been processed, it returns the confidence level in SJISProber, which checks both parsers and returns the result one level higher in MBCSGroupProber.

Single-byte encodings

Seriously, where is my unicode pony?

The explorer for single-byte encodings, SBCSGroupProber (defined in sbcsgroupprober.py), is also just a console that controls a group of researchers, one for each single-byte encoding and language combination: windows-1251, KOI8-R, ISO-8859-5, MacCyrillic, IBM855 , and IBM866 (Russian); ISO-8859-7 and windows-1253 (Greek) ISO-8859-5 and

windows-1251 (Bulgarian) ISO-8859-2 and windows-1250 (Hungarian) TIS-620 (Thai); windows-1255 and ISO-8859-8 (Hebrew).

SBCSGroupProber gives the text to each such researcher specific to the language and encoding and checks the result. These explorers are all implemented as a single class, SingleByteCharSetProber (defined in sbcharsetprober.py), which takes a language model as an argument. The language model determines how often various two-character sequences occur in plain text. SingleByteCharSetProber processes text and marks the most commonly used two-character sequences. Once enough text has been processed, it calculates a level of similarity based on the number of frequent sequences, the total number of characters, and a language-specific distribution factor.

Hebrew is processed in a special way. If, by analyzing the two-character distribution, it turns out that the text can be in Hebrew, HebrewProber (defined in hebrewprober.py) tries to distinguish Visual Hebrew (where each line of source text is stored "in reverse", and then it is displayed in the same way so that it can be read from right to left) and Boolean Hebrew (where the text is stored in reading order and is then displayed right to left in the client). Since some characters are encoded differently depending on the position in the word, we can make an educated guess about the direction of the source text, and determine the desired encoding (windows-1255 for Logical Hebrew or ISO-8859-8 for Visual Hebrew)

windows-1252

If UniversalDetector detects high-byte characters in the text, but none of the other multibyte or single-byte explorers return a positive result, a Latin1Prober (defined in latin1prober.py) is created to try to detect English text in windows-1252 encoding. This would be an inherently unreliable analysis because English characters are encoded in the same way as in many different encodings. The only way to identify windows-1252 is to look at commonly used characters like smart quotes, curly apostrophes, copyright characters, etc. Latin1Prober automatically lowers its confidence level so that other, more credible researchers can benefit if possible.

Creating library packages "

Porting code to Python 3 with 2to3 "

Special method names

We've already found a few special method names throughout the book - magic methods that python calls when you use a certain syntax. Using special methods, your classes can act as sequences, as dictionaries, as functions, as iterators, or even as numbers. The appendix serves as a guide to the special techniques we have already seen and a short introduction to some of the more esoteric ones.

The basics

If you've read the introduction to classes, you've already seen the most common special methods: the __init__ () method . A lot of the classes I've written require some initialization. There are also some other special methods that are especially useful for debugging your custom classes.

1. The __init__ () method is called after the instance has been created. If you want to control the creation process, use the __new__ () method .
2. By convention, the __repr__ () method must return a string that is a valid Python expression.
3. The __str__ () method is also called when print (x) is used .
4. New in Python3, a new bytes type has been introduced.
5. By convention, format_spec must satisfy the Format Specification Mini-Language decimal.py in the Python standard library, which has its own __format__ () method .

Classes that behave like iterators.

In the chapter on iterators, you saw how to build an iterator from scratch using the __iter__ () and __next__ () methods .

1. The __iter__ () method is called when you create a new iterator. This is a good place to initialize an iterator with initial values.
2. The __next__ () method is called when you get the next value from the iterator.
3. The __reversed__ () method is. It takes an existing sequence and returns an iterator that produces the elements in the sequence in reverse order, from last to first.

As you saw in the chapter on Iterators, a for loop can be applied to an iterator. In this loop:

for x in seq : print (x)

Python 3 will call seq. __iter__ () to create an iterator, then calls the __next__ () method on that iterator to get each x value.

When the __next__ () method