Python 3.x

Manual

José Fernando Mendes
da Silva Costa

GitHub profile:
https://github.com/Ze1598

1. Abstract

Python 3.x: Manual is a technical document written with the intention of providing a quick and easy way for developers, or anyone in general, to have information available at all times, online and offline, digitally and on paper, about programming with Python. The document starts with basic definitions so that any reader can still learn something about the topic. Beyond that, the topics tackled range from the most basic ones, such as variable assignment, to more intermediate topics, such as web scraping. All in all, the rationale while creating this document was to have information about Python available at the ready while at the same time making sure that any reader, regarding their programming-related knowledge, could still learn something new.

2. Introduction

Python 3.x: Manual is a technical document I've created based on my own experience as a Python developer, notes taken during practice, various websites and tutorials across the Internet, and last but not least, what is available in the official documents for the programming language[1].

While the official documents are available at the click of a button online, this contains a lot of information, all the information available about Python to be precise, so there is a lot of information you probably won't be interested in or in some cases you may trouble navigating through it. Thus, I decided to create this Manual. This way there would always be at least one option available to look up information (both online and offline), while at the same time having much less information, focusing around the necessary parts needed to understand the language, in a way that would cover, at least, most bases.

Besides this, at a personal level, writing this document was an invaluable experience for a couple of reasons: it was the first time I wrote a document of this magnitude, it made me learn how to interact with technical documentation (specifically programming-related documentation) and, above all, throughout the development of this Manual I found and learned a whole lot of functionalities and intricacies of Python.

Before wrapping up this introduction, I would like to mention that while writing and organizing the content, I tried to keep in mind its accessibility and usability, that is, how the reader

[1] The documentation for Python (from version 3 onward) is available at https://docs.python.org/3/

could have easier access to the topics available in the document, both when using this document digitally as a pdf or when reading it printed. As an answer to that, I used hyperlinks throughout the text so that users reading it digitally could access different chapters with a simple click, while the readers using traditional formats have, for example, the table of contents and indexes available, with the references to chapters or external documents being written explicitly.

3. Table of Contents

4. Figures' Index

5. Table's Index

6. Considerations

Before proceeding to the meat of the document, there are a couple of considerations that need to be addressed for an overall better understanding of this Manual.

Recovering the topic of usability mentioned in the **Introduction** (Chapter 2), at the footer of each page there is a ⌂ symbol. You can click this in any page and it will bring you back to the beginning of the **Table of Contents** (Chapter 3).

The second point that needs to be addressed is the text's formatting. Every word/phrase/expression/etc. written with `grey background` means that portion of the text is written exactly as if it was written in a source code editor (e.g. Microsoft's Visual Studio Code), which means it is simulating code rather than being just "normal" text. Hence, this formatting, in my view, will make it simpler to differentiate code from the rest of the text. Though words can simply be written with an *italic* effect, just to denote that that word or expression is to be interpreted in a programming context.

For the "code part" of the text, some things should be clarified: code written `this way` means it is the actual syntax to be used for a given function, loop, keyword or anything of the sort, basically it means it's something that's always written that way; but if it is in `italic` then it means it's just an argument or a placeholder for something else. In the case of `str.format()`, only the `.format()` part is the syntax for this *method*: `str` is only written as such to signify that a string (`str`) will be in that place when using `.format()`. Take a look at a couple of practical examples for `.format()` to see how this applies:

```
1   a = 1
2   b = 2
3   c = "Hello"
4   d = "World"
5   #1 plus 2 equals 3.
6   print("{} plus {} equals 3.".format(a,b))
7   #Hello World.
8   print("{} {}.".format(c,d))
9   #2 plus 1 equals 3.
10  print("{1} plus {0} equals 3.".format(a,b))
```

Figure 1: Considerations example

If an argument for, a function for example, appears inside brackets, as in `input([prompt])`, then it means that argument is optional. In this case, it means we can use the `input()` function without a *`prompt`* argument.

If a word inside the syntax is written in bold, like **`this`**, then it means it refers to a module. In the example **`random`**`.randint(`*`a, b`*`)`, random refers to the **`random`** module, randint corresponds to the `randint()` function and *`a`* and *`b`* are the arguments for the function.

One final note regarding the examples shown throughout the document. What you see written as comments in the images (`#written like this`) generally show what the output is for the code. In the example above regarding the `.format()` method, there are three outputs (one for each `print()`): the first one outputs `1 plus 2 equals 3`, the second one outputs `Hello World`, and the third one `2 plus 1 equals 3`. These comments can also contain explanations instead of just the output, which will be explicit.

7. Glossary

This chapter's purpose is to introduce the reader to some core concepts of Python, by defining important terms and concepts regarding the language and programming, in general, which will be essential to understanding the content of this Manual.

7.1. >>>

The default Python prompt of the interactive shell. Often seen for code examples which can be executed interactively in the interpreter. However, this notation will not be used in the examples shown throughout this Manual.

7.2. …

The default Python prompt of the interactive shell when entering code for an indented code block or within a pair of matching left and right delimiters (parentheses, square brackets or curly braces).

7.3. Argument

A value passed to a function (or method) when calling it. There are two kinds of arguments: keyword arguments and positional arguments.

Keyword arguments -> an argument preceded by an identifier (e.g. `name=`) in a function call or passed as a value in a dictionary preceded by `**`. For example, `3` and `5` are both keyword arguments in the following calls to `complex()`:

```
1   complex(real=3, imag=5)
2   complex(**{'real': 3, 'imag': 5})
```

Figure 2: Keyword arguments examples

Positional arguments-> an argument that is not a keyword argument. Positional arguments can appear at the beginning of an argument list and/or be passed as elements of an iterable preceded by `*`. For example, `3` and `5` are both positional arguments in the following calls:

```
complex(3, 5)
complex(*(3, 5))
```

Figure 3: Positional arguments examples

7.4. Attribute

A value associated with an *object* which is referenced by name using dotted notation. For example, if an object `o` has an attribute `a` it would be referenced as `o.a`.

7.5. BDFL

Benevolent Dictator For Life, also known as Guido van Rossum[2], Python's creator.

7.6. Binary file

A *file object* able to read and write bytes-like objects. Examples of binary files are files opened in binary mode (`'rb'`, `'wb'` or `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and `gzip.GzipFile`.

7.7. Class

A template for creating user-defined objects. Class definitions normally contain *method* definitions which operate on instances of the class.

7.8. Coercion

The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer `3`, but in `3+4.5`, each argument is of a different type (one integer, one floating-point), and both must be converted to the same type before they can be added or it will raise a `TypeError`.

Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

[2] A link to Guido van Rossum's personal home page: https://gvanrossum.github.io/

7.9. Complex Number

An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part.

Imaginary numbers are real multiples of the imaginary unit (the square root of -1), often written as *i* in mathematics or *j* in engineering.

Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a j suffix, e.g., `3+1j`. To get access to complex equivalents of the **`math`** module, use **`cmath`**.

Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.

7.10. Coroutine

Coroutines is a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the `async def` statement.

7.11. Coroutine Function

A function which returns a coroutine object. A coroutine function may be defined with the `async def` statement, and may contain `await`, `async for`, and `async with` keywords.

7.12. CPython

The canonical implementation of the Python programming language, as distributed on python.org. The term "CPython" is used when necessary to distinguish this implementation from others such as Jython or IronPython.

7.13. Descriptor

Any object which defines the `__get__()`, `__set__()`, or `__delete__()` methods.

When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using `a.b` to get, set or delete an attribute looks up the object named `b` in the class dictionary for `a`, but if `b` is a descriptor, the respective descriptor method gets called.

Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

7.14. Dictionary

An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods.

7.15. Dictionary View

The objects returned from `dict.keys()`, `dict.values()`, and `dict.items()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

To force the dictionary view to become a full list use `list(dictview_object)`.

7.16. Docstring

A string literal which appears as the first expression in a class, function or module.

While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing *class*, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

You can read more about documentation strings in its own section: **Documentation Strings** (Chapter 9.8).

7.17. Duck-typing

A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used ("If it looks like a duck and quacks like a duck, it must be a duck.").

By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution.

Duck-typing avoids tests using `type()` or `isinstance()`, though duck-typing can be complemented with abstract base classes. Instead, it typically employs `hasattr()` tests or EAFP (Easier to Ask for Forgiveness than Permission) programming.

7.18. EAFP

Easier to Ask for Forgiveness than Permission.

This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements.

This technique contrasts with the LBYL (Look Before You Leap) style common to many other languages such as C.

7.19. Expression

A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value.

In contrast to many other languages, not all language constructs are expressions. There are also statements which cannot be used as expressions, such as `if`. Assignments are also statements, not expressions.

7.20. File Object

An object exposing a file-oriented API (with methods such as `.read()` or `.write()`) to an underlying resource.

Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or streams.

There are actually three categories of file objects: raw binary files, buffered binary files and text files. Their interfaces are defined in the **`io`** module. The canonical way to create a file object is by using the `open()` function.

7.21. Floor Division

Mathematical division that rounds down to nearest integer.

The floor division operator is `//`. For example, the expression `11 // 4` evaluates to `2` in contrast to the `2.75` returned by float true division.

Note that `(-11) // 4` is `-3` because that is `-2.75` rounded downward.

7.22. Function

A series of statements which returns some value to a caller.

It can be passed zero or more arguments which may be used in the execution of the body.

7.23. Garbage Collection

The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles.

7.24. Generator

A function which returns a generator iterator. It looks like a normal function except that it contains `yield` expressions for producing a series of values usable in a `for`-loop or that can be retrieved one at a time with the `next()` function.

Usually refers to a generator function, but may refer to a generator iterator in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

7.25. Generator Iterator

An object created by a generator function.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending `try`-statements).

When the generator iterator resumes, it picks-up where it left-off (in contrast to functions which start fresh on every invocation).

7.26. Generator Expression

An expression that returns an iterator.

It looks like a normal expression followed by a `for` expression defining a loop variable, range, and an optional `if` expression.

The combined expression generates values for an enclosing function:

```
#sum of squares 0, 1, 4, ... 81
sum(i*i for i in range(10)) #285
```

Figure 4: Generator expression example

7.27. Hashable

An object is hashable if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

All of Python's immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not. Objects which are instances of user-defined classes are hashable by default.

They all compare unequal (except with themselves), and their hash value is derived from their `id()`.

7.28. IDLE

An Integrated Development Environment for Python.

IDLE is a basic editor and interpreter environment which ships with the standard distribution of Python.

7.29. Immutable

An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered.

A new object has to be created if a different value has to be stored.

They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

7.30. Importing

The process by which Python code in one module is made available to Python code in another module.

7.31. Importer

An object that both finds and loads a module; both a finder and loader object.

7.32. Iterable

An object capable of returning its members one at a time.

Examples of iterables include all sequence types (such as lists, strings, and tuples) and some non-sequence types like dictionaries, file objects, and objects of any classes you define with an `__iter__()` or `__getitem__()` method.

Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...).

When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values.

When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop.

7.33. Iterator

An object representing a stream of data.

Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data is available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raises `StopIteration` again.

Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted.

One notable exception is code which attempts multiple iteration passes. A container object (such as a list) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a for loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

7.34. Lambda

An anonymous inline function consisting of a single expression which is evaluated when the function is called.

The syntax to create a lambda function is `lambda [arguments]: expression`.

You can read more about lambda in its own section: **Anonymous Function (Lambda)** (Chapter 23.4).

7.35. LBYL

Look Before You Leap.

This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the EAFP (Easier to Ask for Forgiveness than Permission) approach and is characterized by the presence of many `if` statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between "the looking" and "the leaping". For example, the code `if key in mapping: return mapping[key]` can fail if another thread removes key from mapping after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

7.36. List

A built-in Python sequence. Despite its name it is more akin to an array in other languages than to a linked list since access to elements are of the time complexity O(1).

7.37. List Comprehension

A compact way to process all or part of the elements in a sequence and return a list with the results.

`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (`0x`) in the range of 0 to 255, inclusive.

The `if` clause is optional. If it was omitted, all elements in `range(256)` would be processed.

7.38. Metaclass

The class of a class.

Class definitions create a class name, a class dictionary, and a list of base classes. The *metaclass* is responsible for taking those three arguments and creating the class.

Most object-oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions.

They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

7.39. Method

A function which is defined inside a class body.

If called as an attribute of an instance of that class, the method will get the instance object as its first argument (which is usually called `self`).

7.40. Method Resolution Order

Method Resolution Order (MRO) is the order in which base classes are searched for a member during lookup.

7.41. Module

An object that serves as an organizational unit of Python code.

Modules have a namespace containing arbitrary Python objects. Modules are loaded into Python by the process of importing.

7.42. Mutable

Mutable objects can change their value but keep their `id()`.

7.43. Namespace

The place where a variable is stored.

Namespaces are implemented as dictionaries. There are the *local*, *global* and *built-in* namespaces as well as *nested* namespaces in objects (in methods).

Namespaces support modularity by preventing naming conflicts. For instance, the functions `builtins.open()` and `os.open()` are distinguished by their namespaces.

Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.islice()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

7.44. Object

Any data with state (attributes or value) and defined behavior (methods).

7.45. Package

A Python module which can contain submodules or recursively, subpackages.

Technically, a package is a Python module with a `__path__` attribute.

7.46. Parameter

A named entity in a function (or method) definition that specifies an argument (or in some cases, arguments) that the function can accept. There are five kinds of parameter:

- positional-or-keyword: specifies an argument that can be passed either positionally or as a keyword argument. This is the default kind of parameter, for example *foo* and *bar* in `def func(foo, bar=None): ...`.
- positional-only: specifies an argument that can be supplied only by position. Python has no syntax for defining positional-only parameters. However, some built-in functions have positional-only parameters (e.g. `abs()`).
- keyword-only: specifies an argument that can be supplied only by keyword. Keyword-only parameters can be defined by including a single var-positional parameter or bare `*` in the parameter list of the function definition before them, for example *kw_only1* and *kw_only2* in `def func(arg, *, kw_only1, kw_only2): ...`.
- var-positional: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `*`, for example *args* in `def func(*args, **kwargs): ...`.

- var-keyword: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `**`, for example `kwargs` in the example above.

Parameters can specify both optional and required arguments, as well as default values for some optional arguments.

7.47. Python 3000

Nickname for the Python 3.x release line (coined long ago when the release of version 3 was something in the distant future.) This is also abbreviated to "Py3k".

7.48. Pythonic

An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages.

For example, a common idiom in Python is to loop over all elements of an iterable using a `for` statement. Many other languages don't have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```
1   for i in range(len(food)):
2       print(food[i])
```

Figure 5: Non-pythonic code example

As opposed to the cleaner, Pythonic method:

```
1   for piece in food:
2       print(piece)
```

Figure 6: Pythonic code example

7.49. Reference Count

The number of references to an object.

When the reference count of an object drops to zero, it is deallocated.

Reference counting is generally not visible to Python code, but it is a key element of the CPython implementation. The `sys` module defines a `getrefcount()` function that programmers can call to return the reference count for a particular object.

7.50. Slice

An object usually containing a portion of a sequence.

A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in *`variable_name`*`[1:3:5]`. The bracket (subscript) notation uses slice objects internally.

7.51. Statement

A statement is part of a suite (a "block" of code).

A statement is either an expression or one of several constructs with a keyword, such as `if`, `while` or `for`.

7.52. Text Encoding

A codec which encodes Unicode strings to bytes.

7.53. Text File

A file object able to read and write string objects.

Often, a text file actually accesses a byte-oriented datastream and handles the text encoding automatically. Examples of text files are files opened in text mode (`'r'` or `'w'`), `sys.stdin`, `sys.stdout`, and instances of `io.StringIO`.

7.54. Triple-quoted String

A string which is bound by three instances of either a quotation mark (") or an apostrophe (').

While they don't provide any functionality not available with single-quoted strings, they are useful for a number of reasons. They allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.

7.55. Type

The type of a Python object determines what kind of object it is; every object has a type.

An object's type is accessible as its `__class__()` attribute or can be retrieved with `type(object)`.

8. Basic Information

This chapter presents some basic information regarding Python, such as variable assignment or how to write comments.

8.1. Variable Assignment

To assign a variable simply write the name for the variable, followed by an equal sign (=), and then the variable's value.

```
x = 1
y = 'Hello'
z = 2.5
```

Figure 7: Variable assignment

8.2. Indentation

Indentation is the number of spaces at the beginning of each line (normally 4 spaces).

```
#no indentation
    #1 tab of indentation
        #2 tabs of indentation
            #3 tabs of indentation
                #...
```

Figure 8: Indentation

8.3. 1-line Comment

Simply use a # to turn a line of code into a commentary.

```
#example of a 1-line commentary
```

Figure 9: 1-line comment

8.4. Multiple-lines Comment

To write comments that span multiple lines (or even if it's just a 1-line commentary), wrap the comment with triple quotes (''') as shown in the example below:

```
'''I
am
a
multiple
line
commentary
'''
```

Figure 10: Multiple lines comment

Note: these are interpreted as string so they need to be properly indented in the programs.

8.5. Concatenation

```
print("Life " + "of " "Brian") #Life of Brian

print("This " + "is " "concatenation.") #This is concatenation

print("Spaces" + "matter" + "in" + "concatenation") #Spacesmatterinconcatenation

print("Try " + "using " + "numbers " + str(2)) #Try using numbers 2
```

Figure 11: Concatenation

As shown in the examples above, it is necessary to leave a space before ending each string, otherwise when using concatenation Python won't leave a space between each string (word) and the output will be a single word just like in the third example.

To use non-strings in concatenation just simply use the `str()` function on the non-string objects, like in the fourth example of Figure 11.

8.6. Output-Formatting

There are a couple of ways to format the outputs in Python: `%`-formatting, `str.format()` and in a recent version of Python *f-strings* were introduced.

8.6.1. %-formatting

You can simply substitute parts of a string with other variables, data, etc. by simply using the syntax shown in Figure 12.

For more information about the different format specifiers available (the character next to the percentage symbol in the strings), read the section **%-formatting Format Specifiers** (Chapter 35.1).

```
a = 1
b = 2
c = "Hello"
d = "World"

print("%d plus %d equals 3." %(a,b)) #1 plus 2 equals 3.
print("%s %s." %(c, d)) #Hello World.
```

Figure 12: %-formatting examples

8.6.2. str.format()

`string.format(*args)`

Substitutes the curly braces for each argument from `.format()`.

```
a = 1
b = 2
c = "Hello"
d = "World"
#1 plus 2 equals 3.
print("{} plus {} equals 3.".format(a,b))
#Hello World.
print("{} {}.".format(c,d))
#2 plus 1 equals 3.
print("{1} plus {0} equals 3.".format(a,b))
```

Figure 13: str.format() formatting examples

By default, it substitutes by order of appearance, but as shown in the example indexes can be assigned inside the curly braces so the substitution occurs per the order designed by the numbers (starts at zero).

For more information about the `.format()` please read the **Format String Syntax** section (Chapter 9.10).

8.6.3. f-string

With Python 3.6 came a brand-new way to format strings: *f-string*. The use logic for it is similar to the `.format()` method but without the need to call any method at the end of the string. Instead, you just need to prefix the string with an `f` and inside of the empty curly brackets you were going to use for `.format()` you write the content that was going to be parsed to the method.

Here is a simple comparison between `.format()` and an f-string:

```
1  day = '2'
2  month = 'January'
3  year = '2018'
4
5  print('Today is {} {}nd, {}'.format(month, day, year))
6  print(f'Today is {month} {day}nd, {year}')
7  #Today is January 2nd, 2018
```

Figure 14: f-string formatting examples

We obtained the exact same result, but in less space using the f-string. Though, even if it is a better solution for general purposes, an f-string has its own caveats: you can't escape characters, that is, you can't use backslashes (\) to escape characters, for instance, apostrophes, and as stated in the beginning, it's a feature only available in versions of Python released starting with Python 3.6, which is one of the most recent ones. But this is not all there is to know about the power of f-strings. If you want to read the full documentation you can read "PEP 498 – Literal String Interpolation" at https://www.python.org/dev/peps/pep-0498/.

All in all, f-strings are a powerful tool that make your code shorter and easier to read and understand.

8.6.4. Output in the Same Line

There are a couple of options available in Python to output data in the same line.

8.6.4.1. Using Commas

```
a = 'world'
b = 1
c = 2

print("hello", a) #hello word
print('test', b, c) #test 1 2
print('test', b * c) #test 2
```

Figure 15: Output in the same line using commas

8.6.4.2. Using print() Parameters

`print(item, end='\n')`

A simple way to output, for example, each character of a string in the same line would be to assign something to the *end* argument inside the `print()` callout.

Each *item* will be followed by the string assigned to *end* in the output.

```
a = 'marble'

for char in a:
    print(char, end = '')
#marble

for char in a:
    print(char, end = ' ')
#m a r b l e

for char in a:
    print(char, end = '-')
#m-a-r-b-l-e-

for char in a:
    print(char, end = '&')
#m&a&r&b&l&e&
```

Figure 16: Output in the same line with print()'s end keyword argument

9. Useful Information

This chapter goes slightly beyond the "basic" information given in the previous **Basic Information** chapter (Chapter 8). It details, for example, how to convert integers from different bases, or how to have access to the contents of a module.

9.1. How To Use Apostrophes Inside Strings

To use an apostrophe inside a string it needs to be escaped, since it's a special character used to denote strings. Therefore, to use an apostrophe or any other special character, you escape it using a backslash, followed the apostrophe, like this:

```
2   "That\'s cool"
3   "It\'s amazing"
4   "\'X\' marks the place"
```

Figure 17: Escape apostrophes in strings

9.2. Boolean Operators' Priority

Boolean operators are evaluated in the following ascending priority order: `or`, `and`, `not`.

9.3. Output the Contents of a Module

To print (output) the full contents of a module, simply import the module and call the dir() function on it.

```
1   import module
2   print(dir(module))
```

Figure 18: Output a module's contents

9.4. Syntax for Extending a Line of Code

The backslash symbol (\) is used to show that a line of code extends for more than one line.

```
a = 1
b = 2
c = a + \
    b
d = c * \
    b

print(c) #3
print(d) #6
```

Figure 19: Line of code that spans more than one line

9.5. Syntax to Write Base 2, 8 or 16 Integers

This section covers the syntax for writing integers in base 2 (binary), base 8 (octal) and base 16 (hexadecimal).

9.5.1. Base 2 Integers

To create a binary number (base 2 integer), write the number in binary then prefix it with `0b` to denote it is written in base 2.

```
0b1 #base 2 integer 1
0b10 #base 2 integer 10
0b11 #base 2 integer 11
```

Figure 20: Binary numbers' syntax

9.5.2. Base 8 Integers

To create an octal number (base 8 integer), write the number in octal then prefix it with `0o` to denote it is written in base 8.

```
0o1 #base 8 integer 1
0o2 #base 8 integer 2
0o3 #base 8 integer 3
```

Figure 21: Octal number' syntax

9.5.3. Base 16 Integers

To create a hexadecimal number (base 16 integer), write the number in hexadecimal then prefix it with `0x` to denote it is written in base 16.

```
1   0x1 #base 16 integer 1
2   0x2 #base 16 integer 2
3   0x3 #base 16 integer 3
```

Figure 22: Hexadecimal numbers' syntax

9.6. Arbitrary Argument Lists

Specify that a function can be called with an arbitrary number of arguments using `*args` as parameter in the function definition. These arguments will be wrapped up in a tuple. Before the variable number of arguments, zero or more normal arguments may occur:

```
1   def write_multiple_items(file, separator, *args):
2       file.write(separator.join(args))
```

*Figure 23: Function definition using *args*

Normally, these arguments will be last in the list of formal parameters, because they scoop up all remaining input arguments that are passed to the function.

Any formal parameters which occur after the `*args` parameter are ‘keyword-only’ arguments, meaning that they can only be used as keywords rather than positional arguments.

```python
def example1(*args, sep = '-'):
    return sep.join(args)
print(example1(str(1), str(2), str(3))) #1-2-3
print(example1("123")) #123
print(example1("1", "2", "3")) #1-2-3
print(example1("1", "2", "3", sep = "=")) #1=2=3

def example2(*args, sep = "/"):
    return sep.join(args)
print(example2("earth", "mars", "venus")) #earth/mars/venus
print(example2("earth", "mars", "venus", sep = ".")) #earth.mars.venus
```

*Figure 24: Practical examples of functions using *args*

9.7. Unpacking Argument Lists

This situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments.

For instance, the built-in `range()` function expects separate *`start`* and *`stop`* arguments. If they are not available separately, write the function call with the `*`-operator to unpack the arguments out of a list or tuple:

```python
a = list(range(3, 6))
#[3, 4, 5]
args = [3, 6]
b = list(range(*args))
#[3, 4, 5]
```

Figure 25: Unpacking argument lists example

In the same way, dictionaries can deliver keyword arguments with the `**`-operator:

```
def parrot(voltage, state='a stiff', action='voom'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.", end=' ')
    print("E's", state, "!")

d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
parrot(**d)
#-- This parrot wouldn't VOOM if you put four million volts through it. \
#E's bleedin' demised !
```

Figure 26: Unpacking argument dictionaries

9.8. Documentation Strings

Some conventions about the content and formatting of documentation strings.

The first line should always be a short, concise summary of the object's purpose. For brevity, it should not explicitly state the object's name or type, since these are available by other means (except if the name happens to be a verb describing a function's operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation if desired. This is done using the following convention:

- The first non-blank line after the first line of the string determines the amount of indentation for the entire documentation string (the first line can't be used for this since it is generally adjacent to the string's opening quotes, which means its indentation is not apparent in the string literal)
- Whitespace "equivalent" to this indentation is then stripped from the start of all lines of the string. Lines that are indented less should not occur, but if they occur all their leading whitespace should be stripped. Equivalence of whitespace should be tested after expansion of tabs (to 8 spaces, normally).

Here is an example of a docstring:

```
def my_function():
    """Do nothing, but document it.

    No, really, it doesn't do anything.
    """
    pass

print(my_function.__doc__)
#Do nothing, but document it.
#
#    No, really, it doesn't do anything.
```

Figure 27: Docstring example

9.9. Print Python's Built-In Elements

Using the syntax `print(locals()['__builtins__'])` prints a dictionary containing all the built-in exceptions and other objects in Python.

Here's a short snippet of the output:

```
print(locals()['__builtins__'])

#(part of the) output
'''
{'__name__': 'builtins', '__doc__': "Built-in functions, exceptions, and other
objects.\n\nNoteworthy: None is the `nil' object; Ellipsis represents `...'
in slices.", '__package__': '', '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__spec__': ModuleSpec(name='builtins', loader=<class '_frozen_importlib.BuiltinImporter'>),
'__build_class__': <built-in function __build_class__>, '__import__':
<built-in function __import__>, 'abs': <built-in function abs>, 'all':
<built-in function all>, 'any': <built-in function any>, 'ascii': <built-in function ascii>,
'''
#do note this is just the beginning, the full output is very extensive
```

Figure 28: Snippet of Python's built-in objects

9.10. Format String Syntax

As you've read in previous chapters, the `str.format()` method is pretty handy to format outputs. However, so far you've only learned about the simple practice of using placeholder curly braces as *replacement fields*.

Before going any further into this let me refresh your memory with an example of `.format()`:

```
a = 1
b = 2
c = "Hello"
d = "World"

print("{} plus {} equals 3.".format(a, b)) #1 plus 2 equals 3.
print("{} {}.".format(c, d)) #Hello World.
```

Figure 29: str.format() basic usage

Replacement fields can go far beyond than this. A "complete" replacement field will look like this: `{[field_name] ["!" conversion] [":" format_spec]}`.

9.10.1. field_name

First off, a replacement field can start with an optional *`field_name`* argument. This specifies the object whose value is to be formatted and replaced in the output.

The *`field_name`* itself begins with an *`arg_name`* that can either be a number or a keyword. If it is a number, then it refers to a positional argument, but if it is a keyword then it refers to a named keyword argument. If the numerical *`arg_name`*s in a format string are written in sequence (e.g. 0,1,2,…), then they can be fully omitted (not just some), and the numbers will be automatically inserted in order.

Since *`arg_name`* is not quote-delimited, it isn't possible to specify arbitrary dictionary keys within a format string. *`arg_name`* can be followed by any number of index or attribute expressions. An expression of the *`.name`* form selects the named attribute using the `getattr()` function, while an expression of the form `[`*`index`*`]` looks for an index using `.__getitem__()`.

```
"First, thou shalt count to {0}"
#References first positional argument
"Bring me a {}"
#Implicitly references the first positional argument
"From {} to {}"
#Same as "From {0} to {1}"
"My quest is {name}"
#References keyword argument 'name'
"Weight in tons {0.weight}"
#'weight' attribute of first positional arg
"Units destroyed: {players[0]}"
#First element of keyword argument 'players'.
```

Figure 30: str.format()'s field_name examples

9.10.2. conversion

The second optional argument is *`conversion`*, which is always preceded by an exclamation point.

The *`conversion`* argument causes a type coercion before formatting. Normally, it is the `.__format__()` method that is responsible for formatting a value. However, in some cases, it is needed to force a type to be formatted as a string (for example), overriding its own definition of formatting.

By converting the value to a string before calling `.__format__()`, the normal formatting logic is bypassed. For this, there are currently three supported conversion flags: `!s` which calls `str()` on the value, `!r` which calls `repr()` and `!a` which calls `ascii()`.

```
"Harold's a clever {0!s}"
#Calls str() on the argument
"Bring out the holy {name!r}"
#Calls repr() on the argument
"More {!a}"
#Calls ascii() on the argument
```

Figure 31: str.format()'s conversion examples

9.10.3. format_spec

Lastly, there's a third optional argument: *format_spec*. This last argument is always preceded by a colon.

The *format_spec* argument contains a specification of how the value should be formatted, including details such as width, alignment, padding, decimal precision and more.

Each value type can define its own "formatting mini-language" or interpretation of the *format_spec*. Most built-in types support a common formatting language.

A *format_spec* can also include nested replacement fields within it. These nested fields may contain a field name, conversion flag and format specification, but deeper nesting is not allowed.

The replacement fields within *format_spec* are substituted before the *format_spec* string is interpreted. This allows the formatting of a value to be dynamically specified.

```
1  print('{:<30}'.format('left aligned'))
2  #'left aligned                  '
3  print('{:>30}'.format('right aligned'))
4  #'                 right aligned'
5  print('{:^30}'.format('centered'))
6  #'           centered           '
7  print('{:*^30}'.format('centered'))
8  #use '*' as a fill char
9  #'***********centered***********'
```

Figure 32: str.format()'s format_spec examples

The "formatting mini-language" will be further explained in the next section (Chapter 9.11).

9.11. Format Specification Mini-Language

As seen in the previous section (Chapter 9.10.3), format specifications are used within replacement fields contained within a format string to define how individual values are presented.

They can also be passed directly to the built-in `.format()` method. Each formattable type may define how the format specification is to be interpreted.

Most built-in types implement the following options for format specifications, although some of the formatting options are only supported by the numeric types.

A general convention is that an empty format string (`""`) produces the same result as if `str()` was called on the value. A non-empty format string typically modifies the result.

A "complete" *`format_spec`* argument will look like this:

`[[fill]align][sign][#][0][width][grouping_option][.precision][type]`

9.11.1. align

If a valid *`align`* value is specified, it can be preceded by a *`fill`* character that can be any character and defaults to a space if omitted.

It is not possible to use a literal curly brace (`{` or `}`) as the *`fill`* character in a formatted string literal or when using the `.format()` method.

However, it is possible to insert a curly brace with a nested replacement field. This limitation doesn't affect the mentioned method.

The possible *`align`*ment options are:

Option	Meaning
`<`	Forces the field to be left-aligned (this is the default for most objects).
`>`	Forces the field to be right-aligned (this is the default for numbers).
`=`	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form `+000000120`. This alignment option is only valid for numeric types. Adopts the default behavior when `0` immediately precedes the field *`width`*.
`^`	Forces the field to be centered within the available space.

Table 1: str.format()'s format_spec align options

Note that unless a minimum field *`width`* is defined, the field *`width`* will always be the same size as the data to fill it, so the *`align`*ment option will have no meaning in this case.

9.11.2. sign

The `sign` option is only valid for number types, and can be one of the following:

Option	Meaning
`+`	Indicates that a sign should be used for both positive as well as negative numbers.
`-`	Indicates that a sign should be used only for negative numbers (this is the default behavior).
literal space	Indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers.

Table 2: str.format()'s format_spec sign options

9.11.3. Stylistic Options

Currently, there are three stylistic options available for format specifications: `#` (octothorpe), `,` (comma) and `_` (underscore).

9.11.3.1. Octothorpe (#)

The `#` option causes the "alternate form" to be used for the conversion.

The alternate form is defined differently for different types. This option is only valid for integer, float, complex and decimal types.

For integers, when binary, octal, or hexadecimal output is used, this option adds the respective prefix `0b`, `0o`, or `0x` to the output value.

For floats, complex and decimal, the alternate form causes the result of the conversion to always contain a decimal-point character, even if no digits follow it. Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.

In addition, for `g` and `G` conversions, trailing zeros are not removed from the result.

9.11.3.2. Comma (,)

The `,` option signals the use of a comma for a thousands separator. For a locale aware separator, use the `n` integer presentation type instead.

9.11.3.3. Underscore (_)

The `_` option signals the use of an underscore for a thousands separator for floating point presentation types and for integer presentation type `d`. For integer presentation types `b`, `o`, `x`, and `X`, underscores will be inserted every 4 digits. For other presentation types, specifying this option is an error.

9.11.4. width

`width` is a decimal integer defining the minimum field width. If not specified, then the field *`width`* will be determined by the content.

When no explicit *`align`*ment is given, preceding the *`width`* field with a zero enables sign-aware zero-padding for numeric types. This is equivalent to a *`fill`* character of `0` with an alignment type of `=`.

9.11.5. .precision

The *`.precision`* is a decimal number indicating how many digits should be displayed after the decimal point for a floating-point value formatted with `f` or `F`, or before and after the decimal point for a floating-point value formatted with `g` or `G`.

For non-number types the field indicates the maximum field size: put in other words, how many characters will be used from the field content. *`.precision`* is not allowed for integer values.

9.11.6. type

Finally, *`type`* determines how the data should be presented.

For information on string, integer and floating-point and decimal presentation types, refer to their tables further ahead (Chapter 35.5, Chapter 35.6 and Chapter 35.7, respectively).

In addition to the integer presentation types, integers can be formatted with the floating-point presentation types (except `n` and `None`). When doing so, `float()` is used to convert the integer to a floating-point number before formatting.

9.12. Test an Object's Memory Consumption

To easily test how much memory a certain object in your program is consuming, use the `getsizeof()` function from the **`sys`** module.

Note: the unit used for the values shown in the following examples is bytes.

```
import sys
x=1
print(sys.getsizeof(x)) #28
x = 12
print(sys.getsizeof(x)) #28
x=1.2
print(sys.getsizeof(x)) #24
x='1.2'
print(sys.getsizeof(x)) #52
x=[1,2]
print(sys.getsizeof(x)) #80
x=[]
print(sys.getsizeof(x)) #64
x=[1]
print(sys.getsizeof(x)) #72
x={}
print(sys.getsizeof(x)) #240
x={'1':1}
print(sys.getsizeof(x)) #240
x={'1':1, '2':2}
print(sys.getsizeof(x)) #240
x = [i**2 for i in range(10)]
print(sys.getsizeof(x)) #192
x = [i**2 for i in range(5)]
print(sys.getsizeof(x)) #128
class Test:
    pass
print(sys.getsizeof(Test)) #1056
```

Figure 33: Test an object's memory consumption

For more information about this function, please read its own section (Chapter 12.14.2).

10. Notable Built-In Functions

In this chapter, only built-in functions of Python will be included. For more functions from certain modules or data types, those will have its own separate chapters.

Here are presented only functions that I found out to be quite useful for the programs I've created so far, that is, functions which I've experience with.

10.1. abs()

`abs(argument)`

Returns the absolute value of a number.

`argument` may be an integer or a float.

If `argument` is a complex number, its magnitude is returned.

```
1  a = 1.5
2  b = -2
3
4  print(abs(a)) #1.5
5  print(abs(b)) #2
```

Figure 34: abs() examples

10.2. bin()

`bin(x)`

Convert an integer `x` to a binary string (base 2).

```
1  print(bin(1)) #turns the base 10 integer 1 into a binary string, 0b1
2  print(bin(0o1)) #turns the base 8 integer 0o1 into a binary string, 0b1
3  print(bin(0x1)) #turns the base 16 integer 0x1 into a binary string, 0b1
```

Figure 35: bin() examples

10.3. bool()

`bool([x])`

Returns a Boolean value: `True` or `False`.

`x` is converted using the standard truth testing procedure.

If *`x`* is `False` or omitted, returns `False`; otherwise it returns `True`.

The `bool` class is a subclass of the integer class `int`. It cannot be "subclassed" further.

Its only instances are `False` and `True`.

```
a = 'a'
b = 'b'

print(bool(a == b)) #False
print(bool(1 < 2)) #True
print(bool(len(list(range(3))) == 3)) #True
```

Figure 36: bool() examples

10.4. complex()

`complex([real [,imag]])`

Returns a complex number with the value *`real + imag * 1j`* or converts a string or a number to a complex number. Both arguments can be any numeric type (including complex).

If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter.

If both arguments are omitted, it returns `0j`.

Due note that when converting from a string, the string must not contain whitespace around the central `+` and `-` operators. For example, `complex('1+2j')` is fine, but `complex('1 +2j')` raises `ValueError`.

```
z = complex(1, 3)
print(z) #1 + 3j
print(z.real) #1.0
print(z.imag) #3.0

a = complex('2+5j')
print(a) #2 + 5j
print(a.real) #2.0
print(a.imag) #5.0

b = complex('5j')
print(b) #5j
print(b.real) #0.0
print(b.imag) #5.0
```

Figure 37: complex() examples

10.5. dict()

`dict(iterable)`

Return a new dictionary, with key-value pairs created from pairs contained in *`iterable`*.

```
a = dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
b = dict([('a', 1), ('b', 2), ('c', 3)])
c = dict([('a',[1,2,3]),('b',[4,5,6])])

print(a) #{'sape': 4139, 'guido': 4127, 'jack': 4098}
print(b) #{'a': 1, 'b': 2, 'c': 3}
print(c) #{'a': [1, 2, 3], 'b': [4, 5, 6]}
```

Figure 38: dict() examples

10.6. enumerate()

`enumerate(iterable, [start=int])`

`iterable` must be a sequence, an iterator, or some other object which supports iteration.

Returns a tuple containing a count (starting the count at *`int`* if given, otherwise defaults to `0`) and the values obtained from iterating over *`iterable`*.

Instead of modifying the already existing *iterable*, `enumerate()` simply returns a tuple version of *iterable*, following the count given at *start*. In other words, `enumerate()` does not modify *iterable*.

```
seasons = ['Spring', 'Summer', 'Fall', 'Winter']
print(list(enumerate(seasons)))
#[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
print(list(enumerate(seasons, start=1)))
#[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
print(list(enumerate(seasons, start=2)))
#[(2, 'Spring'), (3, 'Summer'), (4, 'Fall'), (5, 'Winter')]
```

Figure 39: enumerate() examples

10.7. eval()

`eval(`*expression[, globals=None, locals=None]*`)`

expression is parsed and evaluated as a Python expression (technically speaking, a *condition list*) using the *globals* and *locals* dictionaries as the global and local namespace.

expression is a required argument string, with *globals* and *locals* being optional. If provided, *globals* must be a dictionary. If provided, *locals* can be any mapping object.

If the *globals* dictionary is present and lacks `__builtins__`, the current *globals* are copied into *globals* before the *expression* is parsed. This means that *expression* normally has full access to the standard **builtins** module and restricted environments are propagated.

If the *locals* dictionary is omitted, it defaults to the *globals* dictionary.

If both dictionaries are omitted, the expression is executed in the environment where `eval()` is called. The return value is the result of the evaluated *expression*. Syntax errors are reported as exceptions.

```
x = 1
print(eval('x+1')) #2
#the above expression is evaluated simply as if it was print(x+1)
```

Figure 40: eval() example

This function can also be used to execute arbitrary code objects (such as those created by `compile()`). In this case pass a code object instead of a string. If the code object has been compiled with `'exec'` as the mode argument, `eval()`'s return value will be `None`.

Dynamic execution of statements is supported by the `exec()` function. The `globals()` and `locals()` functions return the current global and local dictionary, respectively, which may be useful to pass around for use by `eval()` or `exec()`.

10.8. filter()

`filter(function, iterable)`

Return an iterator from those elements of *`iterable`* for which *`function`* returns `True`.

`iterable` may be either a sequence, a container which supports iteration, or an iterator.

If *`function`* is `None`, the *identity function* is assumed, that is, all elements of *`iterable`* that return `False` are removed.

Note that `filter(function, iterable)` is equivalent to the generator `expression item for item in iterable if function(item)` if *`function`* is not `None` and `item for item in iterable if item` in case *`function`* is `None`.

```
1  my_list = range(16)
2  print(list(filter(lambda x: x % 3 == 0, my_list)))
3  #[0, 3, 6, 9, 12, 15]
4
5  languages = ['HTML', 'Python', 'Ruby']
6  print(list(filter(lambda x: x == 'Python', languages)))
7  #['Python']
```

Figure 41: filter() examples

10.9. float()

`float(argument)`

Return a floating-point number constructed from *`argument`*.

```
print(float(1)) #1.0
print(float(1.5)) #1.5
print(float(-2)) #-2.0
print(float('1')) #1.0
print(float('-3')) #-3.0
```

Figure 42: float() examples

10.10. hex()

`hex(x)`

Convert an integer *x* to a hexadecimal string (base 16).

```
print(hex(1)) #turns the base 10 integer 1 into an hexadecimal string, 0x1
print(hex(0b1)) #turns the base 2 integer 0b1 into a hexadecimal string, 0x1
print(hex(0o1)) #turns the base 8 integer 0o1 into a hexadecimal string, 0x1
```

Figure 43: hex() examples

10.11. input()

`input([prompt])`

This function is used to receive input from the user.

In case *prompt* is written then it will be outputted and the program will then wait for input from the user.

The input will be automatically turned into a string.

prompt must be a string.

```
a = input('Please enter a word: ')
#outputs Plese write a word:
#then waits for input from the user

b = input()
#outputs nothing; but still waits \
#for input from the user
```

Figure 44: input() examples

10.12. int()

`int(x[, base = y])`

Converts `x` to a base 10 integer if base is not given (default behavior).

```
print(int(1.9)) #1
print(int(1)) #1
print(int(1.1)) #1
print(int('15')) #15
```

Figure 45: int() basic examples

If `base` is given, then it must be 0 (zero) or an integer between 2 and 36, inclusive, and `x` a number written as a string.

`base = y` represents the base which was used to write `x`.

If omitted, `base = y` defaults to base 10 (decimal).

```
print(int('111',2)) #7
print(int('0b111',2)) #7
print(int(str(111),2)) #7
print(int('1a',16)) #26
print(int('0x1a',16)) #26
```

Figure 46: int() advanced examples

10.13. isinstance()

`isinstance(object, classinfo)`

Returns `True` if the *`object`* argument is an instance of the *`classinfo`* argument or an instance of a (direct, indirect or virtual) subclass thereof.

If *`object`* is not an object of the given type, the function always returns `False`.

If *`classinfo`* is a tuple of type objects (or recursively, other such tuples), return `True` if *`object`* is an instance of any of the types.

If *`classinfo`* is not a type or tuple of types and such tuples, a `TypeError` exception is raised.

```
1  print(isinstance(1, int)) #True
2  print(isinstance(1.5, int)) #False
3  print(isinstance(3, str)) #False
4  print(isinstance('str', str)) #True
```

Figure 47: isinstance() examples

10.14. issubclass()

`issubclass(`*`class, classinfo`*`)`

Returns `True` if class is a subclass (direct, indirect or virtual) of *`classinfo`*.

A class is considered a subclass of itself.

`classinfo` may be a tuple of `class` objects, in which case every entry in *`classinfo`* will be checked; in any other case, a `TypeError` exception is raised.

```
1  print(issubclass(int, int)) #True
2  print(issubclass(float, int)) #False
3  print(issubclass(complex, int)) #False
4  print(issubclass(bool, int)) #True
```

Figure 48: issubclass() examples

10.15. iter()

`iter(`*`object[, sentinel]`*`)`

Returns an iterator object.

The first argument is interpreted very differently depending on the presence of the second argument.

Without a second argument, *object* must be a collection object which supports the iteration protocol (the `.__iter__()` method), or it must support the sequence protocol (the `.__getitem__()` method with integer arguments starting at 0). If it does not support either of those protocols, `TypeError` is raised.

If the second argument, *sentinel*, is given, then *object* must be a callable object. The iterator created in this case will call *object* with no arguments for each call to its `.__next__()` method; if the value returned is equal to *sentinel*, `StopIteration` will be raised, otherwise the value will be returned.

```
1   s = 'abc'
2   it = iter(s)
3   print(it)
4   #<str_iterator object at 0x04E5C750>
5   print(next(it)) #a
6   print(next(it)) #b
7   print(next(it)) #c
8   print(next(it))
9   #Traceback (most recent call last):
10  #    print(next(it))
11  #StopIteration
```

Figure 49:iter() example

One useful application for giving the *sentinel* argument is to read lines of a file until a certain line is reached. The following example reads a file until the `readline()` method returns an empty string:

```
1   with open('mydata.txt') as fp:
2       for line in iter(fp.readline, ''):
3           process_line(line)
```

Figure 50: iter() example using sentinel

10.16. len()

`len(iterable)`

Returns the length (number of items) of *`iterable`*.

`iterable` must be an iterable such as a string or a list.

```
a = "string"
len(a) #6

b = [1,2,3,4,5,6,7,8,9,10]
len(b) #10
```

Figure 51: len() examples

10.17. list()

`list([iterable])`

The constructor builds a list whose items are the same and in the same order as *`iterable`*'s items.

`iterable` may be either a sequence, a container that supports iteration, or an iterator object.

If *`iterable`* is already a list, a copy is made and returned.

If no argument is given, the constructor creates a new empty list.

```
print(list('abc')) #[a, b, c]
print(list((1,2,3))) #[1, 2, 3]
print(list()) #[]
```

Figure 52: list() examples

10.18. max()

`max(iterable)`

Return the largest item in an *`iterable`* or the largest of two or more arguments passed to the function call.

```
a = ["t", "test"]
print(max(a)) #test

b = [1,2,3,4,5]
print(max(b)) #5

print(max(1,5,4,7)) #7
```

Figure 53: max() examples

10.19. min()

`min(iterable)`

Return the smallest item in an *`iterable`* or the smallest of two or more arguments passed to the function call.

```
a = ["t", "test"]
print(min(a)) #t

b = [1,2,3,4,5]
print(min(b)) #1

print(min(1,5,4,7)) #1
```

Figure 54: min() examples

10.20. next()

`next(iterator[, default])`

Retrieve the next item from *`iterator`* by calling its `.__next__()` method.

If *`default`* is given, it is returned if *`iterator`* is exhausted, otherwise `StopIteration` is raised.

```
s = 'abc'
it = iter(s)
print(it)
#<str_iterator object at 0x04E5C750>
print(next(it)) #a
print(next(it)) #b
print(next(it)) #c
print(next(it))
#Traceback (most recent call last):
#    print(next(it))
#StopIteration
```

Figure 55: next() examples

10.21. oct()

`oct(x)`

Convert an integer *x* to an octal string (base 8).

```
print(oct(1)) #turns the base 10 integer 1 into an octal string, 0o1
print(oct(0b1)) #turns the base 2 integer 0b1 into a octal string, 0o1
print(oct(0x1)) #turns the base 16 integer 0x1 into a octal string, 0o1
```

Figure 56: oct() examples

10.22. print()

`print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)`

`print()` is mainly used to output some information (*objects*). However it can do it better using the other available arguments for the function.

Print/output *objects* to the text stream file, separated by *sep* and followed by *end*.

sep, *end*, *file* and *flush*, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be `None`, which uses the default values.

If no *objects* are given, `print()` will just write *end*.

The *file* argument must be an object with a `.write(string)` method; if it is not present or `None`, `sys.stdout` will be used.

Since printed arguments are converted to text strings, `print()` cannot be used with binary mode file objects. For these, use `file.write()` instead.

Whether output is buffered is usually determined by *file*, but if the *flush* keyword argument is `True`, the stream is forcibly flushed.

```
a = 'abc'
b = 'cde'

print(a) #abc
print(a, b) #abc, cde
print(a, b, sep = '*') #abc*cde
print(a, end = '=>')
print(b) #abc=>cde
```

Figure 57: print() examples

10.23. property()

`property(fget=None, fset=None, fdel=None, doc=None)`

Returns a property attribute.

fget is a function for getting an attribute value (a getter). *fset* is a function for setting an attribute value (a setter). *fdel* is a function for deleting an attribute value (a deleter). And *doc* creates a docstring for the attribute.

If `c` is an instance of `C`, `c.x` will invoke the getter, `c.x = value` will invoke the setter and `del c.x` the deleter.

If given, *doc* will be the docstring of the property attribute. Otherwise, the property will copy *fget*'s docstring (if it exists).

```
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```

Figure 58: property() examples

In short, using `property()` is the same as using a `@property` decorator. This topic will be tackled further ahead in the **property() and @property** chapter (Chapter 27.11).

10.24. range()

`range([start,] stop[, step])`[3]

The arguments to the `range` constructor must be integers.

If `step` is omitted, it defaults to 1.

If `start` is omitted, it defaults to 0.

If `step` is zero, `ValueError` is raised.

As long as the `step` argument is written accordingly, `range()` can create either a growing or decreasing list of integers.

Do note that in order to actually create a list, the `list()` function should be used (as shown in the examples below).

[3] Rather than being a function or a method, `range()` is an immutable sequence type. However, since `range()` still appears under the Functions section of the official Python documents, `range()` is included in this section of this Manual.

Pertaining the *start* and *stop* arguments, the sequence will start with *start* and end with *stop*-1.

```
a = list(range(5)) #[0,1,2,3,4]
b = list(range(5,0,-1)) #[5,4,3,2,1]
c = list(range(1,5)) #[1,2,3,4]
d = list(range(0,10,2)) #[0,2,4,6,8]
```

Figure 59: range() examples

10.25. round()

round(*number[, ndigits]*)

Return *number* rounded to *ndigits* precision after the decimal point.

If *ndigits* is omitted or is None, it returns the nearest integer to its input.

For the built-in types supporting round(), values are rounded to the closest multiple of 10 to the power of minus *ndigits*.

If two multiples are equally close, rounding is done toward the even choice (so, for example, both round(0.5) and round(-0.5) are 0, and round(1.5) is 2).

Any integer value is valid for *ndigits* (positive, zero, or negative); the return value is an integer if called with one argument, otherwise of the same type as *number*.

```
print(round(10/3, 3)) #3.333
print(round(1.0,3)) #1.0
print(round(10/3)) #3
print(round(2.675, 2)) #2.67
```

Figure 60: round() examples

Note: The behavior of round() for floats can be surprising: for example, round(2.675, 2) returns 2.67 instead of the expected 2.68. This is not a bug: it's a result of the fact that most

decimal fractions can't be represented exactly as a float. Read Floating Point Arithmetic: Issues and Limitations[4] for more information on the matter.

10.26. sorted()

`sorted(iterable, [key], [reverse])`

Return a new sorted list from *`iterable`*.

`key` and *`reverse`* are two optional arguments which must be specified as keyword arguments.

`key` specifies a function that is used to extract a comparison key from each element of *`iterable`* (the default value is `None`, which means it compares the elements directly).

`reverse` is a Boolean value: if set to `True` then the result of using `sorted()` will be the same, but in reverse order.

```
1   a = [3,1,2]
2   print(sorted(a)) #[1,2,3]
3
4   b = ['b', 'c', 'a']
5   print(sorted(b)) #['a', 'b', 'c']
6
7   c = {'c': 3, 'a': 1, 'b': 2}
8   print(sorted(c)) #['a', 'b', 'c']
9
10  d = {'c': 3, 'a': 1, 'b': 2}
11  print(sorted(d, reverse = True)) #['c', 'b', 'a']
```

Figure 61: sorted() examples

10.27. str()

`str(argument)`

Returns a string version of *`argument`*.

If *`argument`* is not given, returns the empty string `''`.

[4] Available at: https://docs.python.org/3/tutorial/floatingpoint.html#tut-fp-issues

```
a = 1
str(a) #"1"

b = 1.5
str(b) #"1.5"

str() #empty string ("")
```

Figure 62: str() examples

10.28. super()

`super([type[, object-or-type]])`

Return a proxy object that delegates method calls to a parent or sibling class of *type*.

This is useful for accessing inherited methods that have been overridden in a class.

The search order is same as that used by `getattr()` except that the *type* itself is skipped.

If the second argument is omitted, the super object returned is unbound.

If the second argument is an object, `isinstance(obj, type)` must be `True`.

If the second argument is a type, `issubclass(type2, type)` must be `True` (this is useful for class methods).

There are two typical use cases for super(): in a class hierarchy with single inheritance, super() can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable.

The second use case is to support *cooperative multiple inheritance* in a dynamic execution environment. This use case is unique to Python and is not found in statically compiled languages or languages that only support single inheritance. This makes it possible to implement "diamond diagrams" where multiple base classes implement the same method.

For both use cases, a typical superclass call looks like this (general syntax followed by a practical example):

```python
#general syntax
class C(B):
    def method(self, arg):
        super().method(arg)
    #This does the same thing as:
    #super(C, self).method(arg)
```

Figure 63: super() equivalence to "manual code"

```python
class Employee(object):
    def __init__(self, employee_name):
        self.employee_name = employee_name

    def calculate_wage(self, hours):
        self.hours = hours
        return hours * 20.00

class PartTimeEmployee(Employee):
    #PartTimeEmployee inherits from Employee
    def part_time_wage(self, hours):
        #this accesses the superclass of PartTimeEmployee, Employee,
        #and executes the calculate_wage method
        return super(PartTimeEmployee, self).calculate_wage(hours)

milton = PartTimeEmployee("Milton")
print(milton.part_time_wage(10))
```

Figure 64: super() example

Note that, aside from the zero-argument form, `super()` is not limited to be used inside methods. The two-argument form specifies the arguments exactly and makes the appropriate references. The zero-argument form only works inside a class definition, as the compiler fills in the necessary details to correctly retrieve the class being defined, as well as accessing the current instance for ordinary methods.

10.29. type()

`type(`*`argument`*`)`

Returns the class of *`argument`*.

```
print(type(1)) #outputs <class 'int'>
print(type('a')) #outputs <class 'str'>
print(type(1.5)) #outputs <class 'float'>
```

Figure 65: type() examples

10.30. zip()

`zip(*iterables)`

Make an iterator that aggregates elements from each of the *`iterables`*.

Returns an iterator of tuples, stopping when the shortest input *`iterable`* is exhausted.

With a single *`iterable`* argument, it returns an iterator of 1-tuples.

With no arguments, it returns an empty iterator.

The left-to-right evaluation order of the *`iterables`* is guaranteed (if *`iterables`* aren't dictionaries).

(Continued in the next page)

```python
a = [3,9,17, 15]
b = [2,4,8,10,30, 40,50,60]

c = list(zip(a,b))
print(c)
#[(3, 2), (9, 4), (17, 8), (15, 10)]

d = [21,23, 45,39,2]

e = list(zip(a,b,d))
print(e)
#[(3, 2, 21), (9, 4, 23), (17, 8, 45), (15, 10, 39)]

f = list(zip(a))
print(f)
#[(3,), (9,), (17,), (15,)]

dict_a = {'a':1, 'b':2}
dict_b = {'c':3, 'd':4}

g = list(zip(dict_a,dict_b))
print(g)
#[('b', 'd'), ('a', 'c')]
#because it's a dictionary the order of iteration
#will be randomized
```

Figure 66: zip() examples

11. Notable Built-In Statements

As with the **Notable Built-In Functions** chapter (Chapter 10), this **Notable Built-In Statements** chapter doesn't contain all statements available in Python, it contains the ones I found out to be most important for my use-cases or that are a precious resource for particular cases.

11.1. break

`break`

`break` may only occur syntactically nested in a `for` or `while` loop, but not nested in a function or class definition within that loop.

It terminates the nearest enclosing loop, skipping the optional `else` clause if the loop has one.

If a `for` loop is terminated by `break`, the loop control target keeps its current value.

```
1   string = 'string'
2   for char in string:
3       print(char)
4       break
5   #s
6
7
8   while 1 < 2:
9       print("Infinite Loop!")
10      break
11  #Infinite Loop!
```

Figure 67: break statement examples

11.2. continue

`continue`

`continue` may only occur syntactically nested in a `for` or `while` loop, but not nested in a function or class definition or `finally` clause within that loop. It continues with the next cycle of the nearest enclosing loop.

When `continue` passes control out of a `try` statement with a `finally` clause, that `finally` clause is executed before starting the next loop cycle.

Put in a simple way, the `continue` statement returns the control to the beginning of the loop. The `continue` statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

```
for num in range(2, 6):
    if num % 2 == 0:
        print("Found an even number", num)
        continue
    print("Found a number", num)
#outputs
#Found an even number 2
#Found a number 3
#Found an even number 4
#Found a number 5

for letter in 'Python':
   if letter == 'h':
      continue
   print('Current Letter :', letter)
#P'\n'y'\n't'\n'o'\n'n
#in this case, the continue statement makes the loop "ignore" the letter 'h'; \
#if letter == 'h', the loop simply restarts, else it prints a small phrase
```

Figure 68: continue statement examples

11.3. del

`del` *`expression`*

The `del` statement can be used to delete elements from a list or dictionary, or even delete complete variables.

```
a = [1,2,3,4]
b = {'a': 1, 'b': 2, 'c': 3}
c = 'variable'
del a[1::2]
del b['b']
del c
print(a) #[1, 3]
print(b) #{'a': 1, 'c': 3}
print(c) #Raises an NameError exception
```

Figure 69: del statement examples

11.4. global

`global` *`identifier`*

The `global` statement is a declaration which holds for the entire current code block. It means that the listed identifiers are to be interpreted as having a global scope. It would be impossible to assign to a global variable without `global`, although free variables may refer to *globals* without being declared global.

Names listed in a `global` statement must not be used in the same code block textually preceding that `global` statement.

Names listed in a `global` statement must not be defined as formal parameters or in a `for` loop control target, class definition, function definition, `import` statement, or variable annotation.

You can read more about the global statement in the **Variable Binding** section (Chapter 27.1.4).

11.5. import

`import` *`modulename`*

`import` is used to import modules. The different types of imports are explained in its own section, **Imports** (Chapter 25).

Though there's one small detail specified here: a specific function or method imported from *`modulename`* can be attributed a different name, using the syntax:

```
#general syntax
from modulename import functionname as newfunctionname

#examples
from math import log as logarithm
from random import randint as random_integer
```

Figure 70: import statement examples

This doesn't change the original name in the module, however it allows you to use the functions with a name of your choice, possibly avoiding function overwriting in your programs.

11.6. nonlocal

`nonlocal` *`identifier`*

The `nonlocal` statement causes the listed identifiers to refer to previously bound variables in the nearest enclosing scope excluding *globals*. This is important because the default behavior for binding is to search the local namespace first.

The statement allows encapsulated code to rebind variables outside of the local scope besides the **`global`** (module) scope.

Names listed in a `nonlocal` statement, unlike those listed in a `global` statement, must refer to pre-existing bindings in an enclosing scope (the scope in which a new binding should be created cannot be determined unambiguously).

Names listed in a `nonlocal` statement must not collide with pre-existing bindings in the local scope.

11.7. pass

`pass`

The `pass` keyword simply serves as a placeholder for code. For example, you can define a function or a class with its content simply being a `pass` statement, no need for anything more. This means `pass` is a null operation: when it is executed nothing happens.

```
class Example1:
    pass
    #this class does nothing

def example1():
    pass
    #this function does nothing
```

Figure 71: pass statement examples

11.8. raise

`raise` *`expressions`*

If no expressions are present, `raise` re-raises the last exception that was active in the current scope. If no exception is active in the current scope, a `RuntimeError` exception is raised indicating that this is an error.

Otherwise, `raise` evaluates the first expression as the exception object. It must be either a subclass or an instance of `BaseException`. If it is a class, the exception instance will be obtained when needed by instantiating the class with no arguments.

The type of the exception is the exception instance's class, the value is the instance itself.

A traceback object is normally created automatically when an exception is raised and attached to it as the `__traceback__` attribute, which is writable. You can create an exception and set your own traceback in one step using the *`exception`*`.with_traceback()` exception method (which returns the same exception instance, with its traceback set to its argument), like so:

```
raise Exception("foo occurred").with_traceback(tracebackobj)
```

Figure 72: raise stament/with_traceback() example

11.8.1. from

The `from` clause is used for exception chaining: if given, the second expression must be another exception class or instance, which will then be attached to the raised exception as the `__cause__` attribute (which is writable).

If the raised exception is not handled, both exceptions will be printed:

```
try:
    print(1 / 0)
except Exception as exc:
    raise RuntimeError("Something bad happened") from exc


#Traceback (most recent call last):
#  File "<stdin>", line 2, in <module>
#ZeroDivisionError: division by zero

#The above exception was the direct cause of the following exception:

#Traceback (most recent call last):
#  File "<stdin>", line 4, in <module>
#RuntimeError: Something bad happened
```

Figure 73: raise/from statements example

A similar mechanism works implicitly if an exception is raised inside an exception handler or a `finally` clause: the previous exception is then attached as the new exception's `__context__` attribute:

```
try:
    print(1 / 0)
except:
    raise RuntimeError("Something bad happened")

#Traceback (most recent call last):
#  File "<stdin>", line 2, in <module>
#ZeroDivisionError: division by zero

#During handling of the above exception, another exception occurred:

#Traceback (most recent call last):
#  File "<stdin>", line 4, in <module>
#RuntimeError: Something bad happened
```

Figure 74: Exception raised inside an exception handler

Exception chaining can be explicitly suppressed by specifying `None` in the `from` clause:

```
try:
    print(1 / 0)
except:
    raise RuntimeError("Something bad happened") from None

#Traceback (most recent call last):
#  File "<stdin>", line 4, in <module>
#RuntimeError: Something bad happened
```

Figure 75: Exception chaining suppression

Changed in Python 3.3: `None` is now permitted as *`Y`* in `raise X from Y` and the `__suppress_context__` attribute to suppress automatic display of the exception context.

11.9. return

`return` *`expression`*

`return` may only occur syntactically nested in a function definition, not within a nested class definition.

If an expression list is present, it is evaluated, else `None` is substituted. `return` then leaves the current function call with the *`expression`* (or `None`) as the return value.

When `return` passes control out of a `try` statement with a `finally` clause, that `finally` clause is executed before really leaving the function.

In a generator function, the `return` statement indicates that the generator is done and will cause `StopIteration` to be raised.

The returned value (if any) is used as an argument to construct `StopIteration` and becomes the `StopIteration.`*`value`* attribute.

```
1  def example1(x):
2      y = 2
3      return bool(x > y)
4  #if x is bigger than y then example1 will return True, \
5  #else it returns False
6
7  print(example1(1)) #False; 1 > 2 is False
8  print(example1(3)) #True; 3 > 2 is True
9  print(example1(5)) #True; 5 > 2 is True
```

Figure 76: return statement example

11.10. try

`try`

In Python, exceptions can be handled using a `try` statement.

11.10.1. try/except

A critical operation which can raise an exception is placed inside the `try` clause and the code that handles the exception is written in the `except` clause.

It is up to the programmer, what operations to perform once the exception has been "caught". Here is a simple example[5]:

(Continued in the next page)

[5] A note about the mathematical term 'reciprocal' used in the example of Figure 77. The reciprocal of a number n is 1 divided by n, or put other way, 1/n or n^{-1}. 'Reciprocal' can also be called 'multiplicative inverse'.

```python
# import sys to get the type of exception
import sys

randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print('The entry is:', entry)
        r = 1/int(entry)
        break
    except:
        print('Oops!',sys.exc_info()[0],'occured.')
        print('Next entry.')
        print()
print('The reciprocal of', entry,'is', r)
```

Figure 77: try/except statements example

```python
#outputs:

#The entry is a
#Oops! <class 'ValueError'> occured.
#Next entry.

#The entry is 0
#Oops! <class 'ZeroDivisionError'> occured.
#Next entry.

#The entry is 2
#The reciprocal of 2 is 0.5
```

Figure 78: Figure 77's code output

In this example, the program iterates through the *`randomList`* list until it tries an *`entry`* with a valid reciprocal value. The portion that can cause an exception is placed inside the `try` block. Do note that as soon as the loop finds a valid value it will be terminated (a simple example, if the `2` came before the `0`, the loop wouldn't try the `0` for its reciprocal value because the `2` would have already tested successfully, thanks to the usage of `break`).

If no exception occurs, the `except` block is skipped. But if any exception occurs, it is caught by the `except` block. When it happens, the name of the exception is printed using the `exc_info()` function from the **`sys`** module and proceed to test the next *`entry`*. We can see that the values *`'a'`* causes `ValueError` and *`0`* causes `ZeroDivisionError`.

Let's move on to using exceptions inside the `except` clause. Though do keep in mind that this is not a good programming practice as it will catch all exceptions and handle every case in the same way. However, it's possible to specify which exceptions an `except` clause will catch.

A `try` clause can have any number of `except` clauses to handle them differently but only one will be executed in case an exception occurs.

It's possible to use a tuple of values to specify multiple exceptions in an `except` clause. Here is a generic example:

```
try:
    pass

except ValueError:
    #handle ValueError exception
    pass

except (TypeError, ZeroDivisionError):
    #handle multiple exceptions
    #TypeError and ZeroDivisionError
    pass

except:
    #handle all other exceptions
    pass
```

Figure 79: try statement with multiple except clauses

In Python programming, exceptions are raised when corresponding errors occur at run time, but we can forcefully raise errors using the `raise` keyword, as seen in Chapter 11.8.

You can also optionally use a string as an argument for the error, to further explain why the exception was raised:

```
raise MemoryError("This is an argument")
#Traceback (most recent call last):
#  File "python", line 3, in <module>
#MemoryError: This is an argument
```

Figure 80: raise an exception passing it an error argument

11.10.2. try/finally

There's still one more clause pertaining the `try` statement: `finally`. This clause is executed no matter what and is generally used to release external resources.

For a real-world example, you may be connected to a remote data center through the network, working with a file or working with a Graphical User Interface (GUI). In all these circumstances, the resource must be cleaned up once used, whether it was successful or not. Either of these actions are performed under the `finally` clause to guarantee execution.

Here is an example of the `finally` clause in file operations:

```
try:
    #Opens a test.txt file in read mode
    f = open("test.txt")
finally:
    #Closes the file
    f.close()
```

Figure 81: try/finally statements example

This way the file will be closed no matter what code was executed after the `try` statement and before the `finally` clause.

11.11. with

`with`

The `with` statement is used to wrap the execution of a block with methods defined by a context manager. This allows common `try/except/finally` usage patterns to be encapsulated for convenient reuse.

The execution of the `with` statement with one "item" proceeds as follows:

1. The context expression (the expression given in the `with_item`) is evaluated to obtain a context manager;
2. The context manager's `__exit__()` is loaded for later use;
3. The context manager's `__enter__()` method is invoked;
4. If a target was included in the `with` statement, the return value from `__enter__()` is assigned to it;
5. The suite is executed;
6. The context manager's `__exit__()` method is invoked. If an exception caused the suite to be exited, its type, value, and traceback are passed as arguments to `__exit__()`. Otherwise, three `None` arguments are supplied;
7. If the suite was exited due to an exception, and the return value from the `__exit__()` method was `False`, the exception is reraised. If the return value was `True`, the exception is suppressed, and execution continues with the statement following the `with` statement;
8. If the suite was exited for any reason other than an exception, the return value from `__exit__()` is ignored, and execution proceeds at the normal location for the kind of exit that was taken.
9. With more than one item, the context managers are processed as if multiple `with` statements were nested:

```
with A() as a, B() as b:
    suite

#is equivalent to

with A() as a:
    with B() as b:
        suite
```

Figure 82: with statement examples

11.12. yield

`yield` *`expression`*

The `yield` keyword is essential for generator functions in Python.

Citing PEP 255[6]: "If a `yield` statement is encountered, the state of the function is frozen, and the value of *`expression`* is returned to `.next()`'s caller. By "frozen" we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, and the internal evaluation stack: enough information is saved so that the next time `.next()` is invoked, the function can proceed exactly as if the `yield` statement were just another external call.

On the other hand, when a function encounters a `return` statement, it returns to the caller along with any value proceeding the `return` statement and the execution of such function is complete for all intent and purposes. One can think of `yield` as causing only a temporary interruption in the executions of a function."

Just like what is shown in the following example, you should take into account that in order to get the values from using a `yield` in a function, you'll need to iterate through those values, using a `for` loop. Then you can store them as you need (in this case we output them right away, and store the values in a list).

(Continues in the next page)

[6] PEP 255: Simple Generators, available at: https://www.python.org/dev/peps/pep-0255/

```python
#define a generator function
#in this case the function generates the numbers in the fibbonaci \
#sequence, from the first until a max value
def fib(max):
    a, b = 0, 1
    while a < max:
        yield a
        a, b = b, a + b

print(fib(10))
#<generator object fib at 0x7fadd5d285c8>
#in order to get the actual values you'll need to loop through the \
#generator with a for loop
for item in fib(10):
    print(item) #0'\n'1'\n'1'\n'2'\n'3'\n'5'\n'8

#or you can create an iterator from the generator
example = iter(fib(10))
#and print it, for example, as a list
print(list(example)) #[0, 1, 1, 2, 3, 5, 8]
```

Figure 83: yield statement example

12. Notable Built-In Modules

This chapter contains only the modules I've come across during my time programming in Python or that are related to some parts of this Manual.

For each module, I will start with a brief description followed by all the content available in the module. In some cases, I will also describe some functions and/or methods and/or classes worth mentioning pertaining to that module.

One last note: this chapter only contemplates content from the Python Standard Library.

12.1. calendar

The **`calendar`** module allows the output of calendars, like the Unix cal program, and additional useful functions related to calendars.

By default, these calendars have Monday as the first day of the week and Sunday as the last (the European convention). This can be modified by setting the first day of the week to Sunday (`6`) or to any other with the `setfirstweekday()` function.

It's worth noting parameters that specify dates are given as integers.

```
1  import calendar
2  print(dir(calendar))
3  '''['Calendar', 'EPOCH', 'FRIDAY', 'February', 'HTMLCalendar',
4  'IllegalMonthError', 'IllegalWeekdayError', 'January',
5  'LocaleHTMLCalendar', 'LocaleTextCalendar', 'MONDAY',
6  'SATURDAY', 'SUNDAY', 'THURSDAY', 'TUESDAY', 'TextCalendar',
7  'WEDNESDAY', '_EPOCH_ORD', '__all__', '__builtins__', '__cached__',
8  '__doc__', '__file__', '__loader__', '__name__', '__package__',
9  '__spec__', '_colwidth', '_locale', '_localized_day',
10 '_localized_month', '_spacing', 'c', 'calendar', 'datetime',
11 'day_abbr', 'day_name', 'different_locale', 'error',
12 'firstweekday', 'format', 'formatstring', 'isleap', 'leapdays',
13 'main', 'mdays', 'month', 'month_abbr', 'month_name',
14 'monthcalendar', 'monthrange', 'prcal', 'prmonth', 'prweek',
15 'repeat', 'setfirstweekday', 'sys', 'timegm', 'week', 'weekday',
16 'weekheader']'''
```

Figure 84: calendar module contents

12.1.1. calendar.monthcalendar()

`calendar.monthcalendar(year, month)`

Returns a matrix (2D list) representing a *month*'s calendar. Each row represents a week, with days outside of the *month* being represented by zeros. By default, each week begins with Mondays unless a different weekday has been set with `setfirstweekday()`.

```
import calendar

print(calendar.monthcalendar(2018,1))
'''
[
    [1, 2, 3, 4, 5, 6, 7],
    [8, 9, 10, 11, 12, 13, 14],
    [15, 16, 17, 18, 19, 20, 21],
    [22, 23, 24, 25, 26, 27, 28],
    [29, 30, 31, 0, 0, 0, 0]
]
'''
```

Figure 85: calendar.monthcalendar() example

12.1.2. calendar.monthrange()

`calendar.monthrange(year, month)`

Returns a tuple of two integers: the first corresponds to the weekday of the first day in *month* (by default, `0` is Monday and `6` is Sunday), with the second being the number of days in *month*.

year is given just to specify to which year *month* refers to.

```
import calendar

#(First weekday, number of days in the month)
print(calendar.monthrange(2018, 1))
#(0, 31); (Starts on a Monday; Has 31 days)
print(calendar.monthrange(2018,2))
#(3, 28); (Starts on a Thursday; Has 28 days)
```

Figure 86: calendar.monthrange() examples

12.1.3. calendar.setfirstweekday()

`calendar.setfirstweekday(weekday)`

Sets the *`weekday`* (`0` being Monday and `6` Sunday) to start each week.

For convenience, the values for each day are provided, that is, to set the first weekday to Sunday you can simply use:

```
1  import calendar
2  calendar.setfirstweekday(calendar.SUNDAY)
3  print(calendar.firstweekday()) #6; == Sunday
4
5  #You can also just change it using integer notation
6  calendar.setfirstweekday(2)
7  print(calendar.firstweekday()) #2; == Wednesday
```

Figure 87:calendar.setfirstweekday() examples

12.2. cmath

The **`cmath`** module is always available. It provides access to mathematical functions for complex numbers. The functions in this module accept integers, floating-point numbers or complex numbers as arguments.

They will also accept any Python object that has either a `__complex__()` or a `__float__()` method: these methods are used to convert the object to a complex or floating-point number, respectively, and the function is then applied to the result of the conversion.

(Continued in the next page)

```
import cmath
print(dir(cmath))

'''
['__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atanh',
'cos', 'cosh', 'e', 'exp', 'inf', 'infj', 'isclose', 'isfinite',
'isinf', 'isnan', 'log', 'log10', 'nan', 'nanj', 'phase', 'pi',
'polar', 'rect', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau']
'''
```

Figure 88: cmath module contents

12.3. collections

The **`collections`** module implements specialized container data types providing alternatives to Python's general purpose built-in containers: dictionaries, lists, sets, and tuples.

```
import collections
print(dir(collections))

'''
['AsyncGenerator', 'AsyncIterable', 'AsyncIterator', 'Awaitable',
'ByteString', 'Callable', 'ChainMap', 'Collection', 'Container',
'Coroutine', 'Counter', 'Generator', 'Hashable', 'ItemsView',
'Iterable', 'Iterator', 'KeysView', 'Mapping', 'MappingView',
'MutableMapping', 'MutableSequence', 'MutableSet', 'OrderedDict',
'Reversible', 'Sequence', 'Set', 'Sized', 'UserDict', 'UserList',
'UserString', 'ValuesView', '_Link', '_OrderedDictItemsView',
'_OrderedDictKeysView', '_OrderedDictValuesView', '__all__',
'__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__path__', '__spec__', '_chain',
'_class_template', '_collections_abc', '_count_elements', '_eq',
'_field_template', '_heapq', '_iskeyword', '_itemgetter', '_proxy',
'_recursive_repr', '_repeat', '_repr_template', '_starmap', '_sys',
'abc', 'defaultdict', 'deque', 'namedtuple']
'''
```

Figure 89: collections module content

12.3.1. collections.Counter()

`collections``.Counter(`*`[iterable-or-mapping]`*`)`

A *Counter* is a dictionary subclass for counting hashable objects.

It is an unordered collection where elements are stored as dictionary keys and their counts are stored as dictionary values.

Counts are allowed to be any integer value including zero or negative counts.

The Counter class is similar to *bags* or *multisets* in other programming languages.

Elements are counted from an `iterable` or initialized from another `mapping` (or Counter):

```
c = Counter() #a new, empty counter
c = Counter('gallahad') #a new counter from an iterable
c = Counter({'red': 4, 'blue': 2}) #a new counter from a mapping
c = Counter(cats = 4, dogs = 8) #a new counter from keyword args
```

Figure 90: Forms of creating Counter objects

`Counter` objects have a dictionary interface except that returns a zero count for missing items instead of raising a `KeyError`:

```
from collections import Counter
c = Counter(['eggs', 'ham'])
print(c['bacon']) #0; count of a missing element is zero
```

Figure 91: Counter objects' dictionary interface

Setting a count to zero does not remove an element from a counter. Use `del` to remove it entirely:

```
c['sausage'] = 0 #counter entry with a zero count
del c['sausage'] #del actually removes the entry
```

Figure 92: Delete an element from a Counter

Counter objects support three methods beyond those available for all dictionaries: `.elements()`, `.subtract()` and `.update()` and some methods related to dictionaries.

12.3.1.1. CounterObject.elements()

`collections.CounterObject.elements()`

A `Counter` class' method.

Return an iterator over elements repeating each as many times as its count.

Elements are returned in arbitrary order.

If an element's count is less than one, `.elements()` will ignore it.

```
1  from collections import Counter
2  c = Counter(a=4, b=2, c=0, d=-2)
3  print(sorted(c.elements()))
4  #['a', 'a', 'a', 'a', 'b', 'b']
```

Figure 93: collections.CounterObject.elements() example

12.3.1.2. dict.fromkeys()

`dict.fromkeys(iterable)`

While this method is available for dictionary objects, it is not implemented for `Counter` objects.

12.3.1.3. CounterObject.most_common()

`collections.CounterObject.most_common([n])`

A `Counter` class' method.

Return a list of the `n` most common elements and their counts from the most common to the least.

If `n` is omitted or `None`, `.most_common()` returns all elements in the counter.

Elements with equal counts are ordered arbitrarily:

```
from collections import Counter
print(Counter('abracadabra').most_common(3))
#[('a', 5), ('r', 2), ('b', 2)]
```

Figure 94: collections.CounterObject.most_common() example

12.3.1.4. CounterObject.subtract()

`collections``.`*`CounterObject`*`.subtract(`*`[iterable-or-mapping]`*`)`

A `Counter` class' method.

Elements are subtracted from an *`iterable`* or from another *`mapping`* (or counter).

Like *`dict`*`.update()` but subtracts counts instead of replacing them.

Both inputs and outputs may be zero or negative.

```
from collections import Counter
c = Counter(a=4, b=2, c=0, d=-2)
d = Counter(a=1, b=2, c=3, d=4)
c.subtract(d)
print(c)
#Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

Figure 95: collections.CounterObject.subtract() example

12.3.1.5. CounterObject.update()

`collections``.`*`CounterObject`*`.update(`*`[iterable-or-mapping]`*`)`

A `Counter` class' method.

Elements are counted from an *`iterable`* or added-in from another *`mapping`* (or counter).

Like *`dict`*`.update()` but adds counts instead of replacing them.

Also, the *`iterable`* is expected to be a sequence of elements, not a sequence of key-value pairs.

12.3.1.6. Counter objects practical examples

```
sum(c.values()) #total of all counts
c.clear() #reset all counts
list(c) #list unique elements
set(c) #convert to a set
dict(c) #convert to a regular dictionary
c.items() #convert to a list of (elem, cnt) pairs
Counter(dict(list_of_pairs)) #convert from a list of (elem, cnt) pairs
c.most_common()[:-n-1:-1] #n least common elements
+c #remove zero and negative counts
```

Figure 96: Counter objects' practical examples

Several mathematical operations are provided for combining `Counter` objects to produce *multisets* (`Counter`s that have counts greater than zero).

Addition and subtraction combine counters by adding or subtracting the counts of corresponding elements.

Intersection and union return the minimum and maximum of corresponding counts.

Each operation can accept inputs with signed counts, but the output will exclude results with counts of zero or less.

```
from collections import Counter
c = Counter(a=3, b=1)
d = Counter(a=1, b=2)
print(c + d) #add two counters together: c[x] + d[x]
#Counter({'a': 4, 'b': 3})

print(c - d) #subtract (keeping only positive counts)
#Counter({'a': 2})

print(c & d) #intersection: min(c[x], d[x])
#Counter({'a': 1, 'b': 1})

print(c | d) #union: max(c[x], d[x])
#Counter({'a': 3, 'b': 2})
```

Figure 97: Counter objects' operations examples

12.4. datetime

The **`datetime`** module supplies classes for manipulating dates and times in both simple and complex ways.

While date and time arithmetic is supported, the focus of the implementation is on efficient attribute extraction for output formatting and manipulation.

```
import datetime
print(dir(datetime))

'''
['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__', '__spec__',
'_divide_and_round', 'date', 'datetime', 'datetime_CAPI', 'time',
'timedelta', 'timezone', 'tzinfo']
'''
```

Figure 98: datetime module contents

12.4.1. datetime.datetime.now()

`datetime``.datetime.now(`*`tz = None`*`)`

A `datetime` class' method.

Return the current local date and time.

If the optional argument *`tz`* is `None` or not specified, this is like `today()`, but, if possible, supplies more precision than can be gotten from going through a **`time`**`.time()` timestamp.

If *`tz`* is not `None`, it must be an instance of a `tzinfo` subclass, and the current date and time are converted to *`tz`*'s time zone. In this case the result is equivalent to `tz.fromutc(datetime.utcnow().replace(tzinfo=tz))`.

You can then access the `.now()` method's properties for specific elements of date and time, such as the current year, hour, or second.

```python
from datetime import datetime

print(datetime.now()) #2017-06-29 20:20:00.479275

#you can also get particular elements this way
time = datetime.now()
print(time.year) #MINYEAR <= year <= MAXYEAR
print(time.month) #1 <= month <= 12
print(time.day) #1 <= day <= number of days in the given month and year
print(time.hour) #0 <= hour < 24
print(time.minute) #0 <= minute < 60
print(time.second) #0 <= second < 60
```

Figure 99: datetime.datetime.now() example

12.4.2. datetime.timedelta()

`datetime.timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0)`

A `timedelta` object represents a duration, that is, the difference, between two dates or times.

All arguments are optional and default to `0`. Arguments may be integers or floating-points and can be either positive or negative.

Note that only `days`, `seconds` and `microseconds` are stored internally. The other arguments are converted to those units:

- A millisecond is converted to 1000 microseconds
- A minute is converted to 60 seconds
- An hour is converted to 3600 seconds
- A week is converted to 7 days

`days`, `seconds` and `microseconds` are then normalized so that their representation remains unique:

- `0 <= microseconds < 1000000`
- `0 <= seconds < 3600*24` (the number of seconds in a day)

- `-999999999 <= days <= 999999999`

Here is an example usage of the `timedelta` class to advance time in a Python program, in this particular example, to advance from a given day in January to the first day of the following month, February 1st. Regarding the `calendar.monthrange()` function used to get the number of days in the month, that function and more about the `calendar` module were already tackled in the previous **calendar** section (Chapter 12.1).

```
1   import datetime, calendar
2
3   #Today's date
4   current_date = datetime.datetime.today() #2018-01-20 15:45:22.317166
5   print(current_date)
6   #Get the number of days in the current month
7   num_days = calendar.monthrange(current_date.year, current_date.month)[1]
8   #Calculate how many days are left in the current month
9   days_left = num_days - current_date.day
10  #Move on to the first day of the next month
11  #For this, we add the number of days left in this month to today's day,\
12  #plus 1 extra day, so we land on the first day of the next month
13  next_month = current_date + datetime.timedelta(days = days_left + 1)
14  print(next_month) #2018-02-01 15:45:05.719285
```

Figure 100: datetime.deltatime() example

You can still take advantage of some `deltatime` class attributes such as these:

```
16  delta = datetime.timedelta(days = days_left + 1)
17  print(delta) #12 days, 0:00:00
18  #You can extract how many days that object equates to by calling the '.days' attribute
19  print(delta.days)
20  #Or even '.seconds' for seconds or '.microseconds' for microseconds
21  print(delta.seconds, delta.microseconds) #0 0; this deltatime uses exactly 12 days
22  #You can use the '.total_seconds()' method to see how many seconds\
23  #that timedelta object contains. You can then use simple maths to convert units
24  print(delta.total_seconds()/60/(24*60)) #12.0; converted to days
```

Figure 101: deltatime class useful attributes

12.4.3. datetime.datetime.today()

`datetime``.datetime.today()`

Return the current local date and time (with `tzinfo` as `None`).

```
import datetime

print(datetime.datetime.today())
#2018-01-20 15:00:39.130879
```

Figure 102: datetime.datetime.today() example

12.5. dummy_threading

This module provides a duplicate interface to the **`threading`** module. It is meant to be imported when the **`_thread`** module is not provided on a platform.

```
import dummy_threading
print(dir(dummy_threading))

'''
['Barrier', 'BoundedSemaphore', 'BrokenBarrierError',
'Condition', 'Event', 'Lock', 'RLock', 'Semaphore',
'TIMEOUT_MAX', 'Thread', 'ThreadError', 'Timer',
'__all__', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__',
'__spec__', 'active_count', 'current_thread', 'enumerate',
'get_ident', 'local', 'main_thread', 'setprofile',
'settrace', 'stack_size', 'threading']
'''
```

Figure 103: dummy_threading module contents

The suggested usage here is:

```
try:
    import threading
except ImportError:
    import dummy_threading as threading
```

Figure 104: Suggested usage for the dummy_threading module

Though be careful to not use this module where deadlock might occur from a thread being created that blocks waiting for another thread to be created. This often occurs with blocking I/O.

12.6. hashlib

The hashlib module implements a common interface to many different secure hash and message digest algorithms.

Included are the FIPS secure hash algorithms SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512 (defined in FIPS 180-2) as well as RSA's MD5 algorithm (defined in Internet RFC 1321[7]).

The terms "secure hash" and "message digest" are interchangeable. Older algorithms were called message digests. The modern term is secure hash.

There is one constructor method named for each type of hash available in the module. All of these return a hash object with the same interface.

For example, `sha256()` is used to create a SHA-256 hash object. Then you can feed bytes-like objects (normally bytes) using the `.update()` method. At any point, you can digest the concatenation of the data fed to the object, using either the `.digest()` or `.hexdigest()` methods.

```
1  import hashlib
2  print(dir(hashlib))
3  '''['__all__', '__builtin_constructor_cache',
4  '__builtins__', '__cached__', '__doc__',
5  '__file__', '__get_builtin_constructor',
6  '__loader__', '__name__', '__package__',
7  '__spec__', '_hashlib', 'algorithms_available',
8  'algorithms_guaranteed', 'blake2b', 'blake2s',
9  'md5', 'new', 'pbkdf2_hmac', 'sha1', 'sha224',
10 'sha256', 'sha384', 'sha3_224', 'sha3_256',
11 'sha3_384', 'sha3_512', 'sha512', 'shake_128',
12 'shake_256']
```

Figure 105: hashlib module contents

12.6.1. hashlib.new()

hashlib.new(*name[, data]*)

[7] RFC 1321 is available online at https://tools.ietf.org/html/rfc1321.html

A generic constructor for the available hash algorithms.

The first parameter, *name*, is a string with the name of the desired algorithm.

The second parameter, *data*, is optional and needs to be a bytes-like object, generally bytes.

Here's an example using `new()` to create an `sha256()` object.

```
1   import hashlib
2
3   #Create a SHA-256 hash object
4   m = hashlib.new('sha256')
5   #Update the hash object with a buffer of bytes
6   m.update(b"Nobody inspects")
7   #Update the object a second time
8   m.update(b" the spammish repetition")
9   #Print the digest using hexadecimal characters
10  print(m.hexdigest())
11  #031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b9406
```

Figure 106: hashlib.new() example

All those lines could be condensed to:

```
13  print(hashlib.new('sha256', b'Nobody inspects the spammish repetition').hexdigest())
14  #031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b9406
```

Figure 107: Condensed hashlib.new() example

12.6.1.1. hashobject.digest()

hashlib.*hashobject*.digest()

Return the digest of the data passed to the `.update()` method so far. This is a bytes object of size `.digest_size` which may contain bytes in the whole range from 0 to 255.

12.6.1.2. hashobject.digest_size

hashlib.*hashobject*.digest_size

The size of the resulting hash, in bytes.

12.6.1.3. hashobject.hexdigest()

`hashlib.hashobject.hexdigest()`

Similar to `.digest()`, except the digest is returned as a string object of double length, containing only hexadecimal characters.

This may be used to exchange the value safely in e-mail or other non-binary environments.

12.6.1.4. hashobject.update()

`hashlib.hashobject.update(arg)`

Update the *hashobject* with the object *arg*, which must be interpretable as a buffer of bytes.

Repeated calls are equivalent to a single call with the concatenation of all the arguments, that is, given an hash object `m`, `m.update(a)` followed by `m.update(b)` is equivalent to `m.update(a+b)`.

12.7. io

The **io** module provides Python's main facilities for dealing with various types of I/O (Input/Output).

There are three main types of I/O: text I/O, binary I/O and raw I/O. These are generic categories, and various backing stores can be used for each of them. A concrete object belonging to any of these categories is called a file object. Other common terms are stream *and file-like* object.

Independently of its category, each concrete stream object will also have various capabilities: it can be read-only, write-only, or read-write. It can also allow arbitrary random access (seeking forwards or backwards to any location), or only sequential access (for example in the case of a socket or pipe).

All streams are careful about the type of data you give to them. For example, giving a string object to the `.write()` method of a binary stream will raise a `TypeError`. So will giving a bytes object to the `.write()` method of a text stream.

```
import io
print(dir(io))

'''
['BlockingIOError', 'BufferedIOBase', 'BufferedRWPair',
'BufferedRandom', 'BufferedReader', 'BufferedWriter',
'BytesIO', 'DEFAULT_BUFFER_SIZE', 'FileIO', 'IOBase',
'IncrementalNewlineDecoder', 'OpenWrapper', 'RawIOBase',
'SEEK_CUR', 'SEEK_END', 'SEEK_SET', 'StringIO',
'TextIOBase', 'TextIOWrapper', 'UnsupportedOperation',
'__all__', '__author__', '__builtins__', '__cached__',
'__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', '_io', 'abc', 'open']
'''
```

Figure 108: io module contents

12.8. itertools

The **`itertools`** module implements a number of iterator building blocks inspired by constructs from APL, Haskell, and SML. Each has been recast in a form suitable for Python.

```
import itertools
print(dir(itertools))

'''
['__doc__', '__loader__', '__name__', '__package__', '__spec__',
'_grouper', '_tee', '_tee_dataobject', 'accumulate', 'chain',
'combinations', 'combinations_with_replacement', 'compress',
'count', 'cycle', 'dropwhile', 'filterfalse', 'groupby',
'islice', 'permutations', 'product', 'repeat', 'starmap',
'takewhile', 'tee', 'zip_longest']
'''
```

Figure 109: itertools module contents

12.9. json

The **`json`** module exposes an API familiar to users of the standard library **`marshal`** and **`pickle`** modules.

With this module, serializing and deserializing JSON data, possibly to and from files, becomes much simpler.

```
import json
print(dir(json))

'''
['JSONDecodeError', 'JSONDecoder', 'JSONEncoder', '__all__',
'__author__', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__',
'__path__', '__spec__', '__version__', '_default_decoder',
'_default_encoder', 'codecs', 'decoder', 'detect_encoding',
'dump', 'dumps', 'encoder', 'load', 'loads', 'scanner']
'''
```

Figure 110: json module contents

12.9.1. json.dump()

json.dump(*obj, fp, *, skipkeys = False, ensure_ascii = True, check_circular = True, allow_nan = True, cls = None, indent = None, separators = None, default = None, sort_keys = False, **kw*)

Serialize *obj* as a JSON formatted stream to *fp* (a `.write()`-supporting object file) using the JSON conversion table.

If *skipkeys* is `True` (`False` by default), then dictionary keys that are not of a basic type (string, integer, float, Boolean, `None`) will be skipped instead of raising a `TypeError`.

The **json** module always produces string objects, not bytes objects. Therefore, *fp*`.write()` must support string input.

If *ensure_ascii* is `True` (the default value), the output is guaranteed to have all incoming non-ASCII characters escaped. Else, these characters will be output as-is.

If *check_circular* is `False` (`True` is the default value), then the circular reference check for container types will be skipped and a circular reference will result in an `OverflowError` (or worse).

If *allow_nan* is False (True is the default value), then it will be a ValueError to serialize out of range float values (nan, inf, -inf) in strict compliance of the JSON specification.

If *allow_nan* is True, their JavaScript equivalents (NaN, Infinity, -Infinity) will be used.

If *indent* is a non-negative integer or string, then JSON array elements and object members will be formatted with that indent level. An indent level of 0, negative, or "" will only insert newlines.

None (the default value) selects the most compact representation.

Using a positive integer *indent* indents that many spaces per level. If *indent* is a string (such as "\t"), that string is used to indent each level.

If specified, *separators* should be an *(item_separator, key_separator)* tuple. The default is (',', ':') if *indent* is *None* and (',', ': ') otherwise. To get the most compact JSON representation, you should specify (',', ':') to eliminate whitespace (which is the default value for *indent* = None).

If specified, *default* should be a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a TypeError. If not specified, TypeError is raised.

If *sort_keys* is True (False is the default value), then the output of dictionaries will be sorted by key.

To use a custom JSONEncoder subclass (e.g. one that overrides the .default() method to serialize additional types), specify it with the cls kwarg; otherwise JSONEncoder is used.

Please read the **JSON Conversion Table** section (Chapter 35.3) to learn more about it.

12.9.2. json.dumps()

json.dumps(*obj, *, skipkeys = False, ensure_ascii = True, check_circular = True, allow_nan = True, cls = None, indent = None, separators = None, default = None, sort_keys = False, **kw*)

Serialize *obj* to a JSON formatted string using the JSON Conversion Table.

The arguments have the same meaning as in **json**.dump().

Due note that keys in key/value pairs of JSON are always of the string type. When a dictionary is converted into JSON, all the keys of the dictionary are coerced to strings. As a result, if a dictionary is converted into JSON and then back into a dictionary, the dictionary may not equal the original one. That is, loads(dumps(*x*)) != *x* if *x* has non-string keys.

Please read the **JSON Conversion Table** section (Chapter 35.3) to learn more about it.

Check out a couple of examples of **json**.dumps() in action. **json**.dump() acts the same way, instead it serializes the converted data directly to an object file.

```
1   import json
2   a = (1,2,3,4,5)
3   b = json.dumps(a)
4   print(b) #[1, 2, 3, 4, 5]
5   #converted a Python tuple to a JSON array
6
7   c = None
8   d = json.dumps(c)
9   print(d) #true
10  #converted Python's None value to JSON's null value
```

Figure 111: json.dumps() examples

12.9.3. json.load()

json.load(*fp, *, cls = None, object_hook = None, parse_float = None, parse_int = None, parse_constant = None, object_pairs_hook = None, **kw*)

Deserialize *fp* (a .read()-supporting object file containing a JSON document) to a Python object using the JSON conversion table.

object_hook is an optional function that will be called with the result of any object literal decoded (a dictionary).

The return value of *object_hook* will be used instead of the dictionary. This feature can be used to implement custom decoders (e.g. JSON-RPC class hinting).

object_pairs_hook is an optional function that will be called with the result of any object literal decoded with an ordered list of pairs.

The return value of *object_pairs_hook* will be used instead of the dictionary.

This feature can be used to implement custom decoders that rely on the order that the key-value pairs are decoded (for example, **collections**.OrderedDict() will remember the order of insertion).

If *object_hook* is also defined, the *object_pairs_hook* takes priority.

parse_float, if specified, will be called with the string of every JSON float to be decoded. By default, this is equivalent to float(*num_str*). This can be used to use another datatype or parser for JSON floats (e.g. **decimal**.Decimal).

parse_int, if specified, will be called with the string of every JSON integer to be decoded. By default, this is equivalent to int(*num_str*). This can be used to use another datatype or parser for JSON integers (e.g. float).

parse_constant, if specified, will be called with one of the following strings: '-Infinity', 'Infinity', 'NaN'. This can be used to raise an exception if invalid JSON numbers are encountered.

Changed in Python 3.1: *parse_constant* doesn't get called on 'null', 'true', 'false' anymore.

To use a custom JSONDecoder subclass, specify it with the cls kwarg; otherwise JSONDecoder is used.

Additional keyword arguments will be passed to the constructor of the *class*.

If the data being *deserialized* is not a valid JSON document, a JSONDecodeError will be raised.

Changed in Python 3.6: All optional parameters are now keyword-only.

Please read the **JSON Conversion Table** section (Chapter 35.3) to learn more about it.

12.9.4. json.loads()

`json.loads(s, *, encoding = None, cls = None, object_hook = None, parse_float = None, parse_int = None, parse_constant = None, object_pairs_hook = None, **kw)`

Deserialize `s` (a string, bytes or bytearray instance containing a JSON document) to a Python object using the JSON conversion table.

The other arguments have the same meaning as in `load()`, except `encoding` which is ignored and deprecated.

If the data being deserialized is not a valid JSON document, a JSONDecodeError will be raised.

Changed in Python 3.6: `s` can now be of type bytes or bytearray. The input encoding should be UTF-8, UTF-16 or UTF-32.

Please read the **JSON Conversion Table** section (Chapter 35.3) to learn more about it.

Check out a couple of examples deserializing JSON data to Python data. Following what happened with `json.dumps()` and `json.dump()`, the behavior with `json.loads()` is similar to the behavior of `json.load()`, except the latter deserializes data directly from a file.

```
1   import json
2   a = '[1,2,3,4,5]'
3   b = json.loads(a)
4   print(b) #[1, 2, 3, 4, 5]
5   #converted a JSON array to a Python list
6
7   c = 'null'
8   d = json.loads(c)
9   print(d) #true
10  #converted JSON's null value to Python's None value
```

Figure 112: json.loads() examples

12.10. math

The `math` module is always available. It provides access to the mathematical functions defined by the C standard.

```
import math
print(dir(math))

'''
['__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf',
'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod',
'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
'''
```

Figure 113: math module contents

12.10.1. math.ceil()

`math.ceil(x)`

Return the ceiling of `x`: the smallest integer greater than or equal to `x`.

If `x` is not a float, delegates to `x.__ceil__()`, which should return an Integral value.

```
from math import ceil

print(ceil(1.5)) #2
print(ceil(1.75)) #2
print(ceil(1005.22)) #1006
print(ceil(0.45)) #1
print(ceil(3.22)) #4
```

Figure 114: math.ceil() examples

12.10.2. math.floor()

`math.floor(x)`

Returns the floor of `x`, the largest integer less than or equal to `x`.

```
from math import floor

print(floor(1.2)) #1
print(floor(1.05)) #1
print(floor(1.9)) #1
print(floor(-0.9)) #-1
print(floor(-1.5)) #-2
```

Figure 115: math.floor() examples

12.10.3. math.log()

`math.log(x[, base])`

With one argument, return the natural logarithm of `x` (to base *e*).

With two arguments, return the logarithm of `x` to the given `base`, calculated as *log(x)/log(base)*.

```
import math

print(math.log(49, 7)) #2.0
print(math.log(64, 4)) #3.0
print(math.log(90, 13)) #1.7543...
```

Figure 116: math.log() examples

12.10.4. math.sqrt()

`math.sqrt(x)`

Returns the square root of `x`.

```
from math import sqrt

print(sqrt(4)) #2.0
print(sqrt(16)) #4.0
print(sqrt(20)) #4.47213595499958
print(sqrt(39)) #6.244997998398398
```

Figure 117: math.sqrt() examples

12.11. os

The **os** module provides a portable way of using operating system dependent functionality.

Note: I am not showing a picture of this module's contents due to its extension.

12.11.1. os.chdir()

`os.chdir(path)`

Change the current working directory to `path`.

`path` must be a string.

In Windows, because `path` is a string, each backslash used in the string must be doubled, because by nature a single backslash inside a string in Python is the way to "escape" the following character (e.g. using a backslash to include an apostrophe like in `"That\'s pretty great."`). As an alternative, simply prefix the path with an `r` to convert the string to a raw string, so that characters can't be escaped.

12.11.2. os.getcwd()

`os.getcwd()`

Return a string representing the current working directory.

12.11.3. os.system()

`os.system(command)`

Execute *command* (a string) in a subshell.

Changes to **sys.stdin**, etc. are not reflected in the environment of the executed *command*. If *command* generates any output, it will be sent to the interpreter standard output stream.

On Unix, the return value is the exit status of the process encoded in the format specified for `wait()`.

On Windows, the return value is that returned by the system shell after running *command*. The shell is given by the Windows environment variable COMSPEC: it is usually cmd.exe, which returns the exit status of the command run; on systems using a non-native shell, consult your shell documentation.

One useful example for **os.system()** would be to run it using `cls`: **os.system('cls')**, which would clear the terminal's screen (the output screen).

```
1  import os, time
2
3  print('Clearing screen in two seconds!')
4  #Sleep for two seconds so the previous print isn't cleared immediately
5  time.sleep(2)
6  #Then clear the screen in the command line using the cls command
7  os.system('cls')
8  #After clearing, print a new statement
9  print('I\'m back.')
```

Figure 118: os.system() example

12.12. random

The **random** module implements pseudo-random number generators for various distributions.

(Continued in the next page)

```
import random
print(dir(random))

'''
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random',
'SG_MAGICCONST', 'SystemRandom', 'TWOPI', '_BuiltinMethodType',
'_MethodType', '_Sequence', '_Set', '__all__', '__builtins__',
'__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', '_acos', '_bisect', '_ceil', '_cos',
'_e', '_exp', '_inst', '_itertools', '_log', '_pi', '_random',
'_sha512', '_sin', '_sqrt', '_test', '_test_generator',
'_urandom', '_warn', 'betavariate', 'choice', 'choices',
'expovariate', 'gammavariate', 'gauss', 'getrandbits',
'getstate', 'lognormvariate', 'normalvariate', 'paretovariate',
'randint', 'random', 'randrange', 'sample', 'seed', 'setstate',
'shuffle', 'triangular', 'uniform', 'vonmisesvariate',
'weibullvariate']
'''
```

Figure 119: random module contents

12.12.1. random.choice()

`random.choice(seq)`

Return a random element from the non-empty sequence `seq`.

If `seq` is empty, raises `IndexError`.

```
from random import choice
list1 = [1,2,3,4,5,6,7,8,9,10]
list2 = ['a', 'b', 'c', 'd', 'e']
#prints a random integer from list1
print(choice(list1))
#prints a random string from list2
print(choice(list2))
```

Figure 120: random.choice() examples

12.12.2. random.random()

`random.random()`

Return a random floating-point number in the range [0.0, 1.0). This means the returned number `N` will always be in the range `0.0 <= N < 1.0`.

```
import random

print(random.random()) #0.6301958652541833
print(random.random()) #0.5441614276129321
```

Figure 121: random.random() examples

12.12.3. random.rantint()

`random.randint(a, b)`

From the **`random`** module.

Generate an integer `N` such that *`a`* `<= N <=` *`b`* (both *`a`* and *`b`* must be integers).

```
import random

print(random.randint(100, 200)) #outputs an integer N; N >= 100 and N <= 200
print(random.randint(1, 10)) #outputs an integer N; N >= 1 and N <= 10
```

Figure 122: random.randint() examples

12.12.4. random.sample()

`random.sample(population, k)`

Return a *`k`* length list of unique elements chosen from the *`population`* sequence or set. Used for random sampling without replacement.

Returns a new list containing elements from *`population`* while leaving the original *`population`* unchanged. The resulting list is in selection order so that all sub-slices will also be valid random samples.

If *`population`* contains repeats, then each occurrence is a possible selection in the sample.

To choose a sample from a range of integers, use a `range()` object as an argument. This is especially fast and space efficient for sampling from a large population (e.g.: `sample(range(10000000), k=60)`).

If the sample size is larger than the *`population`* size, a `ValueError` is raised.

```
1  import random
2
3  print(random.sample(range(10), k=3)) #[4, 0, 5]
4  print(random.sample(['a', 'b', 'c', 'd', 'e'], k=2)) #['a', 'c']
5  print(random.sample({'1', '2', 3, 4.0, -5}, k=4)) #['2', 3, 4.0, -5]
```

Figure 123: random.sample() examples

12.12.5. random.shuffle()

`random.``shuffle(`*`x[, random]`*`)`

Shuffle the sequence *`x`* in place.

The optional argument *`random`* is a 0-argument function returning a random float in [0.0, 1.0); by default, this is the function `random()`.

To shuffle an immutable sequence and return a new shuffled list, use `sample(`*`x`*`, k=len(`*`x`*`))` instead.

Note that even for small `len(`*`x`*`)`, the total number of permutations of *`x`* can quickly grow larger than the period of most random number generators. This implies that most permutations of a long sequence can never be generated. For example, a sequence of length 2080 is the largest that can fit within the period of the Mersenne Twister random number generator.

```
from random import shuffle
list1 = [1,2,3,4,5,6,7,8,9,10]
list2 = ['a', 'b', 'c', 'd', 'e']
#shuffles the order of list1's contents
shuffle(list1)
print(list1)
#example output: [4, 8, 2, 5, 3, 1, 9, 10, 6, 7]
#shuffles the order of list2's contents
shuffle(list2)
print(list2)
#example output: ['c', 'b', 'd', 'e', 'a']
```

Figure 124: random.shuffle() examples

12.12.6. random.uniform()

`random.uniform(a, b)`

Return a random floating-point number `N` such that `a <= N <= b` for `a <= b` and `b <= N <= a` for `b < a`.

The end-point value `b` may or may not be included in the range depending on floating-point rounding in the equation `a + (b-a) * random()`.

```
import random

print(random.uniform(1,2)) #1.889361759074644
print(random.uniform(1,10)) #4.7172790445321215
print(random.uniform(50,100)) #67.75231392485257
```

Figure 125: random.uniform() examples

12.13. re

The **`re`** module provides *regular expressions* (regex) matching operations.

Regular expressions specify a set of strings that match it, using normalized syntax.

One recommendation to keep in mind while using this module is to use *raw strings* for regex (prefix your strings containing a regex with an `r`, such as `r"your_regex"`), so that

backslashes aren't escaped and you can use regex syntax freely (in raw strings characters can't be escaped).

Before getting into the contents of the module, I would also like to point out that this Manual does not include a section dedicated to regular expressions. As such, in the **Notes and specific information** chapter there's a table containing the most used syntax for regex (Chapter 35.8).

In case you'd like a tutorial for this topic, I can point you in the direction of Corey Schafer's video tutorials. He has created two tutorials: one for regular expressions in general[8] and one for regex applied to Python[9] (the `re` module). You can also try the online interactive tutorial from RegexOne[10].

```
import re
print(dir(re))
'''
['A', 'ASCII', 'DEBUG', 'DOTALL', 'I', 'IGNORECASE',
'L', 'LOCALE', 'M', 'MULTILINE', 'RegexFlag', 'S',
'Scanner', 'T', 'TEMPLATE', 'U', 'UNICODE', 'VERBOSE',
'X', '_MAXCACHE', '__all__', '__builtins__', '__cached__',
'__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', '__version__', '_alphanum_bytes',
'_alphanum_str', '_cache', '_compile', '_compile_repl',
'_expand', '_locale', '_pattern_type', '_pickle', '_subx',
'compile', 'copyreg', 'enum', 'error', 'escape', 'findall',
'finditer', 'fullmatch', 'functools', 'match', 'purge',
'search', 'split', 'sre_compile', 'sre_parse', 'sub',
'subn', 'template']
'''
```

Figure 126: re module contents

12.13.1. re.compile()

`re.compile(pattern, flags=0)`

[8] You can find the first vídeo tutorial at: https://www.youtube.com/watch?v=sa-TUpSx1JA&ab_channel=CoreySchafer

[9] You can find the second vídeo tutorial at: https://www.youtube.com/watch?v=K8L6KVGG-7o&ab_channel=CoreySchafer

[10] This tutorial is available at: https://regexone.com/

Compiles a regular expression *pattern* into a `regular expression` object, which can be used, for instance, to look for characters that match that *pattern*, using `.finditter()` or `.findall()`.

The expression's behavior can be modified by specifying a *flag*'s value. These values can be any of the variables stated in Chapter 35.9. These variables can then be combined using the bitwise OR operation (the | operator).

```
1   import re
2
3   search_string = "abcdefabcghicba"
4   #Compiles the 'abc' regex pattern, with no flags
5   pattern = re.compile('abc')
6   #Obtain a lisf of all the matches of 'pattern' in 'search_string'
7   matches = pattern.findall(search_string)
8   print(matches) #['abc', 'abc']
9   #While it did contain 'cba' at the end, that did not count as\
10  #a match since it had to be an exact match to the pattern
```

Figure 127: re.compile() example

An important note regarding this function: what we did in the last example is equivalent to using:

```
12  matches = re.findall('abc', search_string)
13  print(matches) #['abc', 'abc']
```

Figure 128: re.compile() example alternative

The advantage of using **`re.compile()`** is that it becomes possible to save the resulting regular expression object in a variable, so it can later be used to look for that pattern in a different string (e.g. programs where the same expression will be used to look up data in different files).

12.13.2. re.findall()

`re.``findall(`*`pattern,string,flags=0`*`)`

Return all non-overlapping matches of *pattern* in *string*, as a list of strings.

string is scanned from left to right and matches are returned in the order found. If one or more groups[11] are present in the *pattern*, return a list of groups: a list of tuples if the *pattern* has more than one group. Empty matches are included in the result.

```
import re

search_string = '''
615-555-7164,
800-555-5669,
560-555-5153,
900-555-9340
'''
```

Figure 129: Data to be searched with re.findall()

```
#Pattern to be searched: 3 sequences of digits, separated by dashes (-)
pattern = r'\d\d\d-\d\d\d-\d\d\d\d'
#Find the 'pattern' in 'search_string'
matches = re.findall(pattern, search_string)
print(matches)
#['615-555-7164', '800-555-5669', '560-555-5153', '900-555-9340']
#Notice how it did not match the commas, only the digits and dashes
```

Figure 130: re.findall() example

12.13.3. re.finditer()

re.finditer(*pattern, string, flag=0*)

Returns an iterator yielding `match` objects over all non-overlapping matches for the regular expression *pattern*.

string is scanned from left to right and matches are returned in the order found. Empty matches are included in the result.

[11] A group, in the context of regular expressions, is defined as any subpattern inside a pair of parenthesis.

By looping through the returned iterator, it becomes possible to extract each match found, and even the starting and ending indexes of the match in *`string`*. These can be accessed using the `.start()` and `.end()` methods, respectively.

```
import re

search_string = '''
615-555-7164,
800-555-5669,
560-555-5153,
900-555-9340
'''
```

Figure 131: Data to be searched with re.finditer()

```
#Pattern to be searched: 3 sequences of digits, separated by dashes (-)
pattern = r'\d\d\d-\d\d\d-\d\d\d\d'
#Find the 'pattern' in 'search_string'
matches = re.finditer(pattern, search_string)
for match in matches:
    print(match, match.start(), match.end())
#<_sre.SRE_Match object; span=(1, 13), match='615-555-7164'> 1 13
#<_sre.SRE_Match object; span=(16, 28), match='800-555-5669'> 16 28
#<_sre.SRE_Match object; span=(31, 43), match='560-555-5153'> 31 43
#<_sre.SRE_Match object; span=(46, 58), match='900-555-9340'> 46 58
```

Figure 132: re.finditer() example

12.13.3.1.matchobject.start()

`re.`*`matchobject`*`.start(`*`[group]`*`)`

Return the index of the start of the substring matched by *`group`*. *`group`* defaults to zero (meaning the whole matched substring). Returns `-1` if *`group`* exists but did not contribute to the match.

For a *`matchobject`* `m`, `str(m)[m.start()]` will return the first character that matched in the substring. `str(m)[m.start():m.end()]` returns the whole match.

12.13.3.2.matchobject.end()

`re.matchobject.end([group])`

Return the index of the end of the substring matched by *group*. *group* defaults to zero (meaning the whole matched substring). Returns `-1` if *group* exists but did not contribute to the match.

For a *matchobject* `m`, `str(m)[m.end()]` will return the last character that matched in the substring. `str(m)[m.start():m.end()]` returns the whole match.

12.14. sys

The **sys** module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available.

```
import sys
print(dir(sys))

'''
['__displayhook__', '__doc__', '__excepthook__',
'__interactivehook__', '__loader__', '__name__',
'__package__', '__spec__', '__stderr__', '__stdin__',
'__stdout__', '_clear_type_cache', '_current_frames',
'_debugmallocstats', '_getframe', '_git', '_home',
'_xoptions', 'abiflags', 'api_version', 'argv',
'base_exec_prefix', 'base_prefix', 'builtin_module_names',
'byteorder', 'call_tracing', 'callstats', 'copyright',
'displayhook', 'dont_write_bytecode', 'exc_info',
'excepthook', 'exec_prefix', 'executable', 'exit', 'flags',
'float_info', 'float_repr_style', 'get_asyncgen_hooks',
'get_coroutine_wrapper', 'getallocatedblocks', 'getcheckinterval',
'getdefaultencoding', 'getdlopenflags', 'getfilesystemencodeerrors',
'getfilesystemencoding', 'getprofile', 'getrecursionlimit',
'getrefcount', 'getsizeof', 'getswitchinterval', 'gettrace',
'hash_info', 'hexversion', 'implementation', 'int_info', 'intern',
'is_finalizing', 'maxsize', 'maxunicode', 'meta_path', 'modules',
'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix',
'set_asyncgen_hooks', 'set_coroutine_wrapper', 'setcheckinterval',
'setdlopenflags', 'setprofile', 'setrecursionlimit', 'setswitchinterval',
'settrace', 'stderr', 'stdin', 'stdout', 'thread_info', 'version',
'version_info', 'warnoptions']
'''
```

Figure 133: sys module contents

12.14.1. sys.exc_info()

`sys.exc_info()`

Returns a tuple of three values that gives information about the exception that is currently being handled. The information returned is specific to the current thread and to the current stack frame.

If no exception is being handled anywhere on the stack, a tuple containing three `None` values is returned. Otherwise, the values are returned are (*`type`*, *`value`*, *`traceback`*). *`type`* gets

the type of exception being handled (a subclass of `BaseException`), *`value`* gets the exception instance (an instance of the exception type) and *`traceback`* gets a `traceback` object.

In the example

```
import sys

a = ['a', 0, 2]

for item in a:
    print(item)
    try:
        result = 1/item
        print('The result is', result)
    except:
        print(sys.exc_info()[0], 'was raised.')
        print(sys.exc_info())
```

Figure 134: sys.exc_info() example

The following output would be obtained:

```
#a
#<class 'TypeError'> was raised.
#(<class 'TypeError'>, TypeError("unsupported operand type(s) for /: 'int' and 'str'",),\
#<traceback object at 0x05675418>)
#0
#<class 'ZeroDivisionError'> was raised.
#(<class 'ZeroDivisionError'>, ZeroDivisionError('division by zero',),\
#<traceback object at 0x05675418>)
#2
#The result is 0.5
```

Figure 135: sys.exc_info() example output

12.14.2. sys.getsizeof()

`sys``.getsizeof(`*`object[, default]`*`)`

Return the size of *`object`* in bytes.

`object` can be any type of object. All built-in objects will return correct results, but this does not have to hold true for third-party extensions as it is implementation specific.

Only the memory consumption directly attributed to the *object* is accounted for, not the memory consumption of objects it refers to.

If given, *default* will be returned if *object* does not provide means to retrieve the size. Otherwise a `TypeError` will be raised.

`getsizeof()` calls the object's `__sizeof__` method and adds an additional garbage collector overhead if the object is managed by the garbage collector.

(Continued in the next page)

```
import sys
x=1
print(sys.getsizeof(x)) #28
x = 12
print(sys.getsizeof(x)) #28
x=1.2
print(sys.getsizeof(x)) #24
x='1.2'
print(sys.getsizeof(x)) #52
x=[1,2]
print(sys.getsizeof(x)) #80
x=[]
print(sys.getsizeof(x)) #64
x=[1]
print(sys.getsizeof(x)) #72
x={}
print(sys.getsizeof(x)) #240
x={'1':1}
print(sys.getsizeof(x)) #240
x={'1':1, '2':2}
print(sys.getsizeof(x)) #240
x = [i**2 for i in range(10)]
print(sys.getsizeof(x)) #192
x = [i**2 for i in range(5)]
print(sys.getsizeof(x)) #128
class Test:
    pass
print(sys.getsizeof(Test)) #1056
```

Figure 136: sys.getsizeof() examples

12.15. threading

The **`threading`** module constructs higher-level threading interfaces on top of the lower level **`_thread`** module.

```
import threading
print(dir(threading))

'''
['Barrier', 'BoundedSemaphore', 'BrokenBarrierError',
'Condition', 'Event', 'Lock', 'RLock', 'Semaphore',
'TIMEOUT_MAX', 'Thread', 'ThreadError', 'Timer',
'WeakSet', '_CRLock', '_DummyThread', '_MainThread',
'_PyRLock', '_RLock', '__all__', '__builtins__',
'__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', '_active',
'_active_limbo_lock', '_after_fork', '_allocate_lock',
'_count', '_counter', '_dangling', '_deque',
'_enumerate', '_format_exc', '_islice', '_limbo',
'_main_thread', '_newname', '_pickSomeNonDaemonThread',
'_profile_hook', '_set_sentinel', '_shutdown',
'_start_new_thread', '_sys', '_time', '_trace_hook',
'activeCount', 'active_count', 'currentThread',
'current_thread', 'enumerate', 'get_ident', 'local',
'main_thread', 'setprofile', 'settrace', 'stack_size']
'''
```

Figure 137: threading module contents

To read more about this module and threading as a subject in Python, please read its own chapter: **Concurrent Execution (Threads)** (Chapter 20).

12.16. time

The **`time`** module provides various time-related functions. For related functionality, read the **calendar** (Chapter 12.1) and **datetime** (Chapter 12.4) sections for more information on those modules.

```
import time
print(dir(time))
'''['_STRUCT_TM_ITEMS', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'altzone', 'asctime', 'clock',
'ctime', 'daylight', 'get_clock_info', 'gmtime', 'localtime',
'mktime', 'monotonic', 'perf_counter', 'process_time', 'sleep',
'strftime', 'strptime', 'struct_time', 'time', 'timezone',
'tzname']'''
```

Figure 138: time module contents

One point worth noting about this module is that while the module itself is always available, not all functions are available on all platforms. Most of the functions defined here call platform C library functions with the same name. Thus, it may sometimes be helpful to consult the platform documentation, because the semantics of these functions vary among platforms.

12.16.1. time.sleep()

`time.sleep(secs)`

Suspend execution of the calling thread for the `secs` seconds. `secs` may be a floating-point number to indicate a more precise sleep time.

As you can read in the **Concurrent Execution (Threads)** chapter (Chapter 32), this function is essential for this process.

12.17. urllib package

The `urllib` is a package that collects several modules for working with URLs:

- `urllib.request` for opening and reading URLs;
- `urllib.error` containing the exceptions raised by `urllib.request`;
- `urllib.parse` for parsing URLs;
- `urllib.robotparser` for parsing *robots.txt* files.

12.17.1. urllib.error

The `urllib.error` module defines the exception classes for exceptions raised by `urllib.request`. The base exception class is `URLError`.

```
import urllib.error
print(dir(urllib.error))

'''
['ContentTooShortError', 'HTTPError',
'URLError', '__all__', '__builtins__',
'__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__',
'__spec__', 'urllib']
'''
```

Figure 139: urllib.error module contents

12.17.2. urllib.parse

This module defines a standard interface to break Uniform Resource Locator (URL) strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a "relative URL" to an absolute URL given a "base URL."

The module has been designed to match the Internet RFC on Relative Uniform Resource Locators. It supports the following URL schemes: `file`, `ftp`, `gopher`, `hdl`, `http`, `https`, `imap`, `mailto`, `mms`, `news`, `nntp`, `prospero`, `rsync`, `rtsp`, `rtspu`, `sftp`, `shttp`, `sip`, `sips`, `snews`, `svn`, `svn+ssh`, `telnet`, `wais`, `ws`, `wss`.

The **`urllib.parse`** module defines functions that fall into two broad categories: URL parsing and URL quoting.

(Continued in the next page)

```
import urllib.parse
print(dir(urllib.parse))

'''
['DefragResult', 'DefragResultBytes', 'MAX_CACHE_SIZE',
'ParseResult', 'ParseResultBytes', 'Quoter', 'ResultBase',
'SplitResult', 'SplitResultBytes', '_ALWAYS_SAFE',
'_ALWAYS_SAFE_BYTES', '_DefragResultBase',
'_NetlocResultMixinBase', '_NetlocResultMixinBytes',
'_NetlocResultMixinStr', '_ParseResultBase',
'_ResultMixinBytes', '_ResultMixinStr',
'_SplitResultBase', '__all__', '__builtins__',
'__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', '_asciire', '_coerce_args',
'_decode_args', '_encode_result', '_hexdig', '_hextobyte',
'_hostprog', '_implicit_encoding', '_implicit_errors', '_noop',
'_parse_cache', '_portprog', '_safe_quoters', '_splitnetloc',
'_splitparams', '_typeprog', 'clear_cache', 'collections',
'namedtuple', 'non_hierarchical', 'parse_qs', 'parse_qsl',
'quote', 'quote_from_bytes', 'quote_plus', 're',
'scheme_chars', 'splitattr', 'splithost', 'splitnport',
'splitpasswd', 'splitport', 'splitquery', 'splittag',
'splittype', 'splituser', 'splitvalue', 'sys', 'to_bytes',
'unquote', 'unquote_plus', 'unquote_to_bytes', 'unwrap',
'urldefrag', 'urlencode', 'urljoin', 'urlparse', 'urlsplit',
'urlunparse', 'urlunsplit', 'uses_fragment', 'uses_netloc',
'uses_params', 'uses_query', 'uses_relative']
'''
```

Figure 140: urllib.parse module contents

12.17.3. urllib.request

The **`urllib.request`** module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world — basic and digest authentication, redirections, cookies and more.

Note: I am not showing a picture of this module's contents due to its extension.

12.17.4. urllib.robotparser

This module provides a single *class*, `RobotFileParser`, which answers questions about whether or not a particular user agent can fetch a URL on the Web site that published the *robots.txt* file. For more details on the structure of *robots.txt* files, see http://www.robotstxt.org/orig.html.

```
import urllib.robotparser
print(dir(urllib.robotparser))

'''
['Entry', 'RobotFileParser', 'RuleLine',
'__all__', '__builtins__', '__cached__',
'__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__',
'collections', 'urllib']
'''
```

Figure 141: urllib.robotparser module contents

12.17.5. urllib.response

The **`urllib.response`** module defines functions and classes which define a minimal file-like interface, including `.read()` and `.readline()`. The typical response object is an `addinfourl` instance, which defines an `.info()` method and that returns headers and a `.geturl()` method that returns the URL. Functions defined by this module are used internally by the **`urllib.request`** module.

```
import urllib.response
print(dir(urllib.response))

'''
['__all__', '__builtins__', '__cached__',
'__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'addbase',
'addclosehook', 'addinfo', 'addinfourl',
'tempfile']
'''
```

Figure 142: urllib.response module contents

12.18. http.client

The **`http.client`** module defines classes which implement the client side of the HTTP and HTTPS protocols. It is normally not used directly — the module **`urllib.request`** uses it to handle URLs that use HTTP and HTTPS.

Note: HTTPS support is only available if Python was compiled with SSL support (through the **`ssl`** module).

Note: I am not showing a picture of this module's contents due to its extension.

13. Numeric Operations

This chapter presents the numeric operations available in Python and how to use them.

Please note these are ordered by ascending priority and all numeric operations have a higher priority than comparison operators.

Operations	Result
`x + y`	Sum of `x` and `y`.
`x - y`	Difference of `x` and `y`.
`x * y`	Product of `x` and `y`.
`x / y`	Quotient of `x` and `y`.
`x // y` [12]	Floored quotient of `x` and `y`.
`x % y` [13]	Remainder of `x / y`.
`-x`	`x` negated.
`+x`	`x` unchanged.
`abs(x)`	Absolute value or magnitude of `x`.
`int(x)` [6]	`x` converted to integer.
`float(x)` [14,6]	`x` converted to floating point.
`complex(re,im)` [15]	A complex number with real part `re`, imaginary part `im`; `im` defaults to zero.
`c.conjugate()`	Conjugate of the complex number `c`.
`divmod(x,y)` [4]	The pair `(x // y, x % y)`.
`pow(x,y)` [16]	`x` to the power of `y`.
`x ** y` [7]	`x` to the power of `y`.

Table 3: Numeric Operations

[12] Also referred to as integer division; the resultant value is a whole integer, though the result's type is not necessarily an integer; the result is always rounded towards minus infinity: `1//2 == 0`, `(-1)//2 == -1`, `1//(-2) == -1`, and `(-1)//(-2) == 0`

[13] Not for complex numbers; instead convert to floats using `abs()` if appropriate

[14] float also accepts the strings `"nan"` and `"inf"` with an optional prefix `"+"` or `"-"` for Not a Number (`NaN`) and positive or negative infinity

[15] The numeric literals accepted include the digits 0 to 9 or any Unicode equivalent

[16] Python defines `pow(0, 0)` and `0 ** 0` to be `1`, as is common for programming languages

14. Data Types

This chapter presents the most commonly used data types in Python. Some of these types will have their own chapters further ahead.

14.1. str

Textual data in Python is handled with `str` (string) objects. To create a string, you can use single, double or triple quotes, with the latter option being generally called multi-line comments or even docstrings depending on the case.

```
'string'
"string"
'''str
ing'''
"""str
ing"""
```

Figure 143: str examples

Read the **Strings** chapter (Chapter 15) for more information on this data type.

14.2. int

`int` (integers) are "normal" integers from Mathematics, that is, `1`, `2` or `3` are examples of integers in Python. This means integers have unlimited precision.

```
1
2
3
4
```

Figure 144: int examples

14.3. float

`float` represents floating point numbers. Put simply, these are numbers that have a decimal point with digits after it.

```
1.5
2.3
4.0
10.29
7.33
```

Figure 145: float examples

14.4. complex

`complex` numbers, as you'd expect, have a real and imaginary part, each being a floating-point number. To extract each part from a `complex` number `z`, you can extract them by calling the `z.real` and `z.imag` attributes to extract the real and the imaginary parts, respectively.

```
z = complex(1, 3)
#complex(real_part, imag_part)
print(z)
print(z.real) #1.0
print(z.imag) #3.0
```

Figure 146: complex example

14.5. bool

`bool` (Boolean values) can only be one of two values: `True` or `False`.

It is worth noting that Booleans are a subtype of integers.

14.6. list

As a general definition, `list` is a mutable sequence type, typically used to store various items. The syntax for a `list` is to wrap the items with square brackets.

```
list_1 = [1,2,3,4,5,]
list_2 = [] #empty list
list_3 = ['a', 'b', 'c']
list_4 = [1, 2,'a', 3, 'b', 'c']
```

Figure 147: list examples

Read the **Lists** chapter (Chapter 16) for more information on this data type.

14.7. dict

`dict` (dictionaries) are an unordered mutable mapping type in Python, that is, it maps keys to values, creating key-value pairs. The syntax for a `dict` is to wrap the pairs with curly brackets.

Because of its unordered nature, neither keys nor values can be accessed by indexing as in `list` or `str` objects. Instead you access them using the keys.

```
1  dict_1 = {'a':1, 'b':2, 'c':3}
2  dict_2 = {
3      'd':4,
4      'e':5,
5      'f':6
6  }
7  dict_3 = {} #empty dictionary
```

Figure 148: dict examples

Read the **Dictionaries** chapter (Chapter 17) for more information on this data type.

14.8. tuple

`tuple` (tuples) are an immutable sequence type in Python, typically used to store collections of heterogeneous data. The syntax for a `tuple` is to wrap the `tuple`'s data in parenthesis, but you can simply assign the data to a single variable and it will be automatically converted to using parenthesis.

```
1  #example of a tuple
2  t = 12345, 54321, 'hello!'
3  print(t[0]) #12345
4  print(t) #(12345, 54321, 'hello!')
```

Figure 149: tuple example

Read the **Tuples** chapter (Chapter 18) for more information on this data type.

14.9. set

`set` (sets) is an unordered collection of distinct objects, that is, in a `set` each item is unique (there are no duplicates in a `set`). To create a `set` you also wrap the data using curly brackets, but instead of having key-value pairs as in `dict` objects, you only have single values such as strings.

```
1  basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
2  #show that duplicates have been removed
3  print(basket) #{'apple', 'orange', 'pear', 'banana'}
```

Figure 150: set example

Read the **Sets** chapter (Chapter 19) for more information on this data type.

15. Strings

15.1. Basic Information

Textual data in Python is handled with `str` objects (strings). Strings are immutable sequences of Unicode code points. String literals are written in a variety of ways:

- Single quotes: `'allows embedded "double" quotes'`
- Double quotes: `"allows embedded 'single' quotes"`
- Triple quoted: `'''Three single quotes'''`, `"""Three double quotes"""`

Triple quoted strings may span multiple lines - all associated whitespace will be included in the string literal.

String literals that are part of a single expression and have only whitespace between them will be implicitly converted to a single string literal. That is, `("spam " "eggs")` is equivalent to `"spam eggs"`.

Strings may also be created from other objects using the `str()` constructor.

Since there is no separate “character” type, indexing a string produces strings of length 1. That is, for a non-empty string `s`, `s[0]` is equivalent to `s[0:1]`.

Then this means strings can be indexed (subscripted), with the first character having index zero. So, for example, for a string `s` such that `s = “++Python++"`, you can access the `Python` portion using `s[2:7]`.

There is also no mutable string type, but `str.join()` can be used to efficiently construct strings from multiple fragments.

15.2. Notable String Methods

The following are a handful of important string methods used commonly for operations with strings.

15.2.1. str.capitalize()

`str.capitalize()`

Returns *str* with its first character capitalized, and the rest in lowercase.

```
a = "CAPS"
b = "no caps"

print(a.capitalize()) #Caps
print(b.capitalize()) #No caps
```

Figure 151: str.capitalize() examples

15.2.2. str.format()

str.format(*argument*)

At first use, .format() can simply be used to replace the placeholder curly braces, {}, inside *str* with *argument* in the output.

```
a = 1
b = 2
c = "Hello"
d = "World"

print("{} plus {} equals 3.".format(a, b)) #1 plus 2 equals 3.
print("{} {}.".format(c, d)) #Hello World.
```

Figure 152: str.format() examples

Note: in case you actually need to use either { or } in a string, just simply use double braces: {{ or }}.

However, this method is used for much more than this, if the string formatting syntax is applied to it. Please read Chapter 9.10 for more information on the subject.

15.2.3. str.join()

str.join(*iterable*)

Return a new string which is the concatenation of the strings in *iterable*.

A TypeError will be raised if there are any non-string values in *iterable*, including bytes objects.

The separator between elements is the *str* providing this method.

```
string = 'coding is cool'
a = string.split()
print(a) #['coding', 'is', 'cool']
print(' '.join(a)) #coding is cool
print(''.join(a)) #codingiscool
print('-'.join(a)) #coding-is-cool
```

Figure 153: str.join() examples

15.2.4. str.isalpha()

str.isalpha()

Returns True if all the characters in *str* are alphabetic and there is at least one character, else it returns False.

```
a = 'aaa'
b = '1a'
c = ''

print(a.isalpha()) #True
print(b.isalpha()) #False
print(c.isalpha()) #False
```

Figure 154: str.isalpha() examples

Note: alphabetic characters are those characters defined in the Unicode character database as “Letter”.

15.2.5. str.isdigit()

str.isdigit()

Return True if all characters in *str* are digits and there is at least one character, False otherwise.

```
print('a'.isdigit()) #False
print('1'.isdigit()) #True
print(''.isdigit()) #False
```

Figure 155: str.isdigit() examples

15.2.6. str.lower()

`str.lower()`

Returns all characters of `str` in lowercase.

`str` must be a string.

```
a = "Hello World"
a.lower() #"hello world"

b = "ALL CAPS"
b.lower() #"all caps"
```

Figure 156: str.lower() examples

15.2.7. str.split()

`str.split([sep] [,maxsplit = integer])`

Return a list of the words in `str`, using `sep` as the delimiter (`sep` must be a string).

If `maxsplit` is given, at most `maxsplit` splits are done (thus, the list will have at most `maxsplit+1` elements). If `maxsplit` is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made). `maxsplit` must be `0` or any other positive integer.

If `sep` is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings. The `sep` argument may consist of multiple characters.

If `sep` is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if `string` has leading or trailing whitespace; consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns an empty list.

```
print('1,2,3'.split(',')) #['1', '2', '3']
print('1,2,3'.split(',', maxsplit = 1)) #['1', '2,3']
print('1,2,,3,'.split(',')) #['1', '2', '', '3', '']
print('1<>2<>3'.split('<>')) #['1', '2', '3']
print('1, 2, 3'.split(maxsplit = 1)) #['1,', '2, 3']
print(''.split()) #[]
print('1 2 3'.split()) #['1', '2', '3']
print('1 2 3'.split(maxsplit = 1)) #['1', '2 3']
print('   1   2   3   '.split()) #['1', '2', '3']
```

Figure 157: str.split() examples

15.2.8. str.strip()

`str.strip([chars])`

Returns a copy of `str` with the leading and trailing `char`acters removed.

`chars` is a string that specifies the characters to be removed. If omitted or `None`, `chars` defaults to removing whitespaces.

The `chars` argument is not a prefix or suffix, rather, all combinations of its values are stripped. The outermost leading and trailing `chars` argument values are stripped from the string.

Characters are removed from the leading end until reaching a `str` character that is not contained in the set of characters in `chars`. A similar action takes place on the trailing end.

(Continued in the next page)

```
a = '     Lots of leading whitespace!'
print(a.strip())
#Lots of leading whitespace!

b = 'Lots of trailing whitespace!     '
print(b.strip())
#Lots of trailing whitespace!

c = '    Lots of whitespace!      '
print(c.strip())
#Lots of whitespace!

d = 'dddddLots of leading "d"s!'
print(d.strip('d'))
#Lots of leading "d"s!

e = 'Lots of trailing "e"s and "."!....eeeeee'
print(e.strip('e.'))
#Lots of trailing "e"s and "."!
```

Figure 158: str.strip() examples

15.2.9. str.upper()

`str.upper()`

Returns all characters of `str` in uppercase.

`str` must be a string.

```
a = "Hello World"
a.upper() #"HELLO WORLD"

b = "no caps"
b.upper() #"NO CAPS"
```

Figure 159: str.upper() examples

16. Lists

`list` (lists) is a data type used to store information as a sequence, all under the same variable (as shown in the next example).

```
1   list_1 = [1,2,3,4,5,]
2   list_2 = [] #empty list
3   list_3 = ['a', 'b', 'c']
4   list_4 = [1, 2,'a', 3, 'b', 'c']
```

Figure 160: list examples

16.1. How to Create Lists

Lists can be created through several ways:

- Using a pair of square brackets to denote the empty list

```
4   list_2 = [] #empty list
```

Figure 161: Create empty lists

- Using square brackets, separating items with commas

```
1   list_1 = ['a', 'b', 'c']
2   list_2 = [1, 2,'a', 3, 'b', 'c']
```

Figure 162: Create populated lists

- Using a *list comprehension* (Chapter 16.3)

```
1   list_1 = [x for x in range(5)] #[0,1,2,3,4]
```

Figure 163: Create lists using list comprehension

- Using the type constructor

```
1   list_4 = list(range(5)) #[0,1,2,3,4]
```

Figure 164: Create lists using list()

16.2. Lists' Operations and Methods

The following points will be operations lists, such as overwriting items from a list or adding new items, and what syntax or list-specific methods can be used to do so.

16.2.1. Item Callout

To call a specific item from a list the same process (syntax) is applied as when using just a portion of a string. This process can be called list slicing, which uses subscription.

```
1  list_1 = [1,2,3,4,5]
2  print(list_1[0]) #1
3  print(list_1[0] + 9) #10
4  print(list_1[2:]) #[3,4,5]
5  print(list_1[:4]) #[1,2,3,4]
```

Figure 165: Item callout in lists

16.2.2. Item Overwriting

To modify (overwrite) a specific item from a list, the syntax to be used is the following:

```
1  #list_item[index] = new_value
2
3  list_1 = [1,2,3,4,5]
4  list_1[4] = 10 #5 -> 10
5  list_1[2] = 6 #3 -> 6
6  print(list_1) #[1,2,6,4,10]
```

Figure 166: Item overwriting in lists

16.2.3. list.append() and list.insert()

You can add items to a list using two different methods: `list.append()` will append an item to the end of the list, while `list.insert()` will insert an item at the designated index.

16.2.3.1. list.append()

`list.append(x)`

Add (append) `x` to the end of `list`.

```
list_1 = [1,2,3,4,5]
list_1.append(6)
list_1.append(7)
print(list_1) #[1,2,3,4,5,6,7]
```

Figure 167: list.append() examples

16.2.3.2. list.insert()

`list.insert(i, x)`

Insert an item in a list at a given position: `i` is the index of the element before which to insert, so `list.insert(0, x)` inserts at the front of `list`, while `list.insert(len(list), x)` is equivalent to `list.append(x)`.

```
list_1 = ['ant', 'boar', 'cat']
list_1.insert(1, 'dog')
list_1.insert(len(list_1), 'bee')
print(list_1)
#['ant', 'dog', 'boar', 'cat', 'bee']
```

Figure 168: list.insert() examples

16.2.4. Group Lists

It's possible to just group (sum) different lists, for example, sum `list_a` and `list_b`, just as if they were simple integers to obtain a new list that contains all items from both lists:

```
list_a = [1, 2]
list_b = [3, 4]
list_c = [5, 6]
list_d = list_a + list_b + list_c
print(list_d) #[1, 2, 3, 4, 5, 6]
```

Figure 169: How to group lists

16.2.5. Check an Item's Index

You can see what's the index of a certain item from a list using the `list.index()` method.

`list.index(x[, start[, end]])`

Return the zero-based index in `list` of the first item whose value is `x`.

The optional arguments `start` and `end` are interpreted as in the *slice notation* and are used to limit the search to a particular subsequence of `list`.

The returned index is computed relative to the beginning of the full sequence rather than the `start` and `end` arguments.

```
1   list_1 = ['ant', 'boar', 'cat']
2   print(list_1.index('cat')) #2
```

Figure 170: list.index() example

16.2.6. Sort a List

To sort a list, simply use the `list.sort()` method:

`list.sort([reverse])`

Modifies `list` so that its items are sorted in a growing order.

If the items are numbers, `list.sort()` will simply compare each number and sort it by growing order. For strings, it will sort them by alphabetical order.

If `reverse` is set to `True`, then the order of the sorted list will be reversed.

```
1   a = [4,5,6,2,1,3]
2   a.sort()
3   print(a) #[1,2,3,4,5,6]
4
5   b = ['fish', 'eagle', 'sloth']
6   b.sort(reverse = True)
7   print(b) #['sloth', 'fish', 'eagle']
8
9   b = ['fish', 'eagle', 'sloth']
10  b.sort()
11  print(b) #['eagle', 'fish', 'sloth']
```

Figure 171: list.sort() examples

16.2.7. Occurrences of an Item

To know how many times a certain item occurs in a list (how many of the same item the list contains) you can simply use the `list.count()` method.

`list.count(x)`

Total number of occurrences of `x` in `list`.

```
1   a = [1,2,3,1,4,1,5,1,1]
2   print(a.count(1)) #5
3
4   b = ['a', 'b', 'a', 'c', 'a']
5   print(b.count('a')) #3
```

Figure 172: list.count() examples

16.2.8. Delete/Remove Items from a List

You can delete or remove items from a list using four different ways, each with its own idiosyncrasies: `list.pop()`, `list.remove()`, `del` and `list.clear()`.

16.2.8.1. list.pop()

`list.pop([i])`

Remove the item at index `i` in `list` and return it.

If no index `i` is specified, `list.pop()` removes and returns the last item in `list`.

```
4   a = [1,2,3,1,4]
5
6   print(a.pop(1)) #2
7   print(a) #[1,3,1,4]
```

Figure 173: list.pop() example

16.2.8.2. list.remove()

`list.remove(x)`

Remove the first item from `list` whose value is `x`.

```
#general syntax
list_name.remove(x)

a = [1,2,3,1,4]

a.remove(1)
print(a) #[2,3,1,4]
```

Figure 174: list.remove() example

16.2.8.3. del

`del list[i]`

Given the index `i` from `list`, delete the item(s) that correspond to that index (`i` can either be a single index, a `list` slice or the complete `list` (`del list`)).

```
#general syntax
del list_name[list_index]

a = [-1, 1, 66.25, 333, 333, 1234.5]

del a[0]
print(a) #[1, 66.25, 333, 333, 1234.5]

del a[2:4]
print(a) #[-1, 1, 333, 1234.5]

del a[:]
print(a) #[]
```

Figure 175: del list[i] examples

16.2.8.4. list.clear()

`list.clear()`

This method removes (clears) every item from `list`, returning only an empty `list`, that is, it modifies `list`, does not create a new empty list.

```
4   list_1 = [1,2,3]
5   list_1.clear()
6   print(list_1) #[]
```

Figure 176: list.clear() example

16.2.9. Flatten Lists (1D Lists)

Sometimes you may have a 2D list, a list that contains other lists. If you want to turn them into a "normal" 1D list (flatten the list) you have two ways of doing it (actually, there's a third one, but we'll get to that in a bit).

16.2.9.1. Using Nested for Loops

The first way to flat a list it to use nested `for` loops. As usual you'd start by looping through the list, but because the list contains other lists (sub-lists) you want to extract (flatten) the contents of those sub-lists. That's where you use a nested `for` loop. Each time there's a list inside another list you'll need to use another `for` loop.

This way of obtaining a flat list isn't very practical because it will not be as readable as the other methods and it is a case-by-case situation, for example, a list `list_1` may only need one nested `for` loop but `list_2` might need two nested `for` loops. My point is, you can't automatize the process simply using `for` loops.

Take a look at the example below:

(Continued in the next page)

```python
a = [[1,2,3], ['a', 'b', 'c']]

print(a[0]) #[1,2,3]
print(a[1]) #['a','b','c']

def complt_list(lst):
    final = []
    for num in lst:
        for itemm in num:
            final.append(itemm)
    print(final)

print(complt_list(a)) #[1,2,3,'a','b','c']
```

Figure 177: Flatten lists using for loops

16.2.9.2. Using list.extend()

The second method still involves using a `for` loop, but this time we add the `list.extend()` method to the mix. While looping through the original list, if you find an item that is a list you can simply call the `list.extend()` method on it and append the contents of that sub-list to a separate list (probably a list that will contain all the content from the original list, but this one is flat).

Take a look at the method in depth below using the same list from the previous example (Figure 177):

`list.extend(iterable)`

Extend `list` by appending all the items from `iterable`. Equivalent to `a[len(a):] = iterable`.

(Continued in the next page)

```
a = [[1,2,3], ['a','b','c']]
#will hold the contents from 'a', except this is a flat version
flat_list = []

#loop through the contents of 'a'
for item in a:
    #if item is a list then call the extend() method on it
    if type(item) is list:
        flat_list.extend(item)

print(flat_list)
#[1, 2, 3, 'a', 'b', 'c']
```

Figure 178: Flatten lists using list.extend()

16.2.9.3. Using Recursion

Finally, we arrive at the third method: recursion. I've shown you how to flatten lists using both nested `for` loops and using a combination of a `for` loop and the `list.extend()` method.

But, in both examples, we only had a sub-list inside the original list, we didn't have "sub-sub-lists" (or in a more technical way a list with depth 3 or more). Well, in this case it'd be a bother to make a specific number of nested `for` loops for each specific case (yes, without using recursion you'd still need nested for loops even using `list.extend()`).

Then let's stick with the `list.extend()` method, but use it in a recursive function context. To raise the bar even higher, let's have two new list examples: one with depth 3 and one with depth 4. Using the same function for both we'll be able to flatten both nonetheless, which proves the automatization of the process.

(Continued in the next page)

```python
a = [[1,[2],3], ['a','b','c']] #depth 3
b = [[1,[2,3]], [['a',['b']],'c']] #depth 4

def recursive_flat(lst):
    #will be a flat version of the input lists
    flat_lst = []
    #loop through 'lst'
    for item in lst:
        #if 'item' is a list
        if type(item) is list:
            #then recursively extend() it to 'flat_lst' until item is a flat list
            flat_lst.extend(recursive_flat(item))
        #if 'item' is not a list then simply append it to 'flat_lst'
        else:
            flat_lst.append(item)
    return flat_lst

print(recursive_flat(a))
#[1, 2, 3, 'a', 'b', 'c']
print(recursive_flat(b))
#[1, 2, 3, 'a', 'b', 'c']
```

Figure 179: Flatten lists using recursion

16.3. List Comprehension

List comprehensions provide a concise way to create lists.

Common applications are to make new lists where each element is the result of some operation applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition[17].

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more `for` or (optional) conditional clauses. The result will be a new list created from evaluating the expression in the context of the `for` and the conditional clauses (if used).

[17] For example, in `for x in iterable:`, when using this it means any previous variable called `x` from outside the loop will be overwritten. If, instead, you use a list comprehension you won't have to worry about this happening, plus its far more concise and readable.

```python
#general syntax
list_comp = [expresion for variable in sequence or iterable]

example_1 = [i**2 for i in range(10)]
print(example_1) #[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

example_1 = []
for i in range(10):
    example_1.append(i**2)
print(example_1) #[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]


example_2 = [i**2 for i in range(12) if i % 2 == 0]
print(example_2) #[0, 4, 16, 36, 64, 100]

example_2 = []
for i in range(12):
    if i % 2 == 0:
        example_2.append(i**2)
print(example_2) #[0, 4, 16, 36, 64, 100]


neg_num = [-1,-2,-3,-4,-5,-6,-7,-8,-9,-10]
pos_num = [abs(i) for i in neg_num]
print(pos_num) #[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

pos_num = []
for i in neg_num:
    pos_num.append(abs(i))
print(pos_num) #[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Figure 180: List comprehension examples

17. Dictionaries

A dictionary is another data type built into Python. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type (strings and numbers can always be keys).

```
dict_1 = {'a':1, 'b':2, 'c':3}
dict_2 = {
    'd':4,
    'e':5,
    'f':6
}
dict_3 = {} #empty dictionary
```

Figure 181: Dictionaries examples

17.1. Basic Information

Tuples can be used as keys if they contain only strings, numbers, or tuples (if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key). Lists can't be used as keys since these can be modified using different methods.

It is best to think of a dictionary as an unordered set of key-value pairs, with the requirement that the keys are unique (within the same dictionary). A pair of braces creates an empty dictionary (like in the example of Figure 181).

Because dictionaries are an unordered set of key-value pairs, we should keep two things in mind: it's not possible to access a specific value by its index as in a list, but rather you access it using its key from the key-value pair, and when you print a dictionary's content it won't always be the same, what I wrote in one of the examples ahead is just one of the possible outputs for that dictionary, but the next time I print the same dictionary the result will likely not be the same.

17.2. Dictionaries' Operations

The main operations on a dictionary are storing a value with some key and extracting the value given the corresponding key. If you store a value using a key that is already in use, the old value associated with it will be overwritten.

Next are some practical examples of operations using dictionaries and the syntax for these operations. Note that only figures with code will be shown, any explanation needed will be included as a comment in the code itself.

17.2.1. Output a Dictionary's Content

```
dict_1 = {'a':1, 'b':2, 'c':3}

dict_2 = {
    'd':4,
    'e':5,
    'f':6
}

dict_3 = {} #empty dictionary

print(dict_1) #{'a':1, 'b':2, 'c':3}
print(dict_2) #{'d':4, 'e':5, 'f':6}
print(dict_3) #{}
```

Figure 182: Print whole dictionaries

```
dict_1 = {
    'fish': ['a', 'b', 'c', 'd'],
    'cash': -483,
    'luck': 'good'
}

#Alternatively, loop through a dictionary\
#and extract key-value pairs using dict.items()
for k, v in dict_1.items():
    print('Key:', k, 'Value:', v)
#Key: fish Value: ['a', 'b', 'c', 'd']
#Key: cash Value: -483
#Key: luck Value: good
```

Figure 183: Print whole dictionaries using a loop

17.2.2. Output a Key's Value

```
#general syntax
print(dict_example['key_name'])

dict_1 = {'a':1, 'b':2, 'c':3}

dict_2 = {
    'd':4,
    'e':5,
    'f':6
}

print(dict_1['b']) #2
print(dict_2['d']) #4
```

Figure 184: Output a key's value

17.2.3. Add New Keys to a Dictionary

```
#general syntax
dict_example['key_name'] = 'key_value'

dict_1 = {'a':1, 'b':2, 'c':3}

dict_2 = {
    'd':4,
    'e':5,
    'f':6
}

dict_1['z'] = 26
dict_2['g'] = 7

print(dict_1) #{'b': 2, 'c': 3, 'a': 1, 'z': 26}
print(dict_2) #{'f': 6, 'g': 7, 'e': 5, 'd': 4}
#do note the order of the keys in the output is randomised
```

Figure 185: Add new keys to a dictionary

17.2.4. Delete Specific Key-Value Pairs (del)

```
#general syntax
del dict_name[key_name]

dict_1 = {'a':1, 'b':2, 'c':3}

del dict_1['a']
print(dict_1) #{'b':2, 'c':3}
```

Figure 186: Delete key-value pairs

17.2.5. Overwrite a Key's Value

```
#general syntax
dict_example['key'] = new_value

dict_1 = {'a':1, 'b':2, 'c':3}

dict_1['a'] = 5
print(dict_1) #{'a':5, 'b':2, 'c':3}
```

Figure 187: Overwrite a key's value

17.2.6. Output a Dictionary's Keys

```
dict_1 = {
    'fish': ['a', 'b', 'c', 'd'],
    'cash': -483,
    'luck': 'good'
}

for key in dict_1:
    print(key)
#fish
#cash
#luck
#Alternatively, call the dict.keys() method
print(dict_1.keys())
#dict_keys(['fish', 'cash', 'luck'])
```

Figure 188: Output all keys from a dictionary

17.2.7. Output a Dictionary's Values

```
dict_1 = {
    'fish': ['a', 'b', 'c', 'd'],
    'cash': -483,
    'luck': 'good'
}

for key in dict_1:
    print(dict_1[key])
#['a', 'b', 'c', 'd']
#-483
#good
#Alternatively, call the dict.values() method
print(dict_1.values())
#dict_values([['a', 'b', 'c', 'd'], -483, 'good'])
```

Figure 189: Output all values from a dictionary

17.2.8. Create a Dictionary with dict()

```
a = dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
b = dict([('a', 1), ('b', 2), ('c', 3)])
c = dict([('a',[1,2,3]),('b',[4,5,6])])

print(a) #{'sape': 4139, 'guido': 4127, 'jack': 4098}
print(b) #{'a': 1, 'b': 2, 'c': 3}
print(c) #{'a': [1, 2, 3], 'b': [4, 5, 6]}
```

Figure 190: Create a dictionary using dict()

For more about the `dict()` function, read its own section (Chapter 10.5).

17.3. Dictionary Comprehension

Just like with lists, there's also dictionary comprehension. These work and are written fundamentally in the same way:

```python
#general syntax
#dict_comp = {key: value for (key, value) in iterable}

example_1 = {i: i**2 for i in (2, 4, 6)}
print(example_1) #{2: 4, 4: 16, 6: 36}

example_2 = {i: x for (i,x) in zip(['a','b','c'],(range(1,4)))}
print(example_2) #{'a': 1, 'b': 2, 'c': 3}
```

Figure 191: Dictionary comprehension examples

18. Tuples

Tuples are very similar to lists. However, there are some key differences, most notably the fact that tuples are immutable (its elements cannot be changed).

18.1. Theory

Look at Figure 192 for some examples of tuples in action:

```
#example of a tuple
t = 12345, 54321, 'hello!'
print(t[0]) #12345
print(t) #(12345, 54321, 'hello!')

# Tuples may be nested:
u = t, (1, 2, 3, 4, 5)
print(u) #((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))

#Tuples are immutable:
t[0] = 88888
#Traceback (most recent call last):
#  File "python", line 10, in <module>
#TypeError: 'tuple' object does not support item assignment


#but they can contain mutable objects:
v = ([1, 2, 3], [3, 2, 1])
print(v) #([1, 2, 3], [3, 2, 1])
```

Figure 192: Tuples examples

As you can see from the examples, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression).

It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.

A special problem is the construction of tuples containing zero or one items: the syntax has some extra quirks to accommodate these.

Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses).

For example:

```
empty = ()
singleton = 'hello', #note trailing comma
print(len(empty)) #0
print(len(singleton)) #1
print(singleton) #('hello',)
```

Figure 193: Empty tuples and 1-tuples examples

The statement `t = 12345, 54321, 'hello!'` is an example of *tuple packing*: the values `12345, 54321 and 'hello!'` are packed together in a tuple. The reverse operation is also possible:

```
x, y, z = t
```

Figure 194: Sequence unpacking example

This is called, appropriately enough, *sequence unpacking* and works for any sequence on the right-hand side. Sequence unpacking requires that there are as many variables on the left side of the equals sign as there are elements in the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.

18.2. Advantages of Tuples Over Lists

To finish off the **Tuples** chapter, let's enumerate some reasons to use tuples over lists:

- Tuples are generally used to contain different data types while lists are primarily used to contain similar data types;
- Since tuples are immutable, it's faster to iterate through a tuple than through a list, which leads to a slight boost in performance;
- Tuples that contain immutable elements can be used as keys for dictionaries, something not possible with lists;

- If you have data that doesn't change, implementing it as a tuple will guarantee that it remains write-protected.

19. Sets

Python also includes a data type for *sets*. A set is an unordered collection with no duplicate elements.

19.1. Theory

Basic uses include membership testing and eliminating duplicate entries.

Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the `set()` function can be used to create sets.

Note: to create an empty set you have to use `set()`, not `{}`; the latter creates an empty dictionary.

Here are some practical examples of a set and its operations:

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banna'}
#Show that duplicates have been removed
print(basket) #{'orange', 'pear', 'apple', 'banna'}
print(type(set())) #<class 'set'>; Empty set
print(type({})) #<class 'dict'>; Empty dictionary
```

Figure 195: Set examples

19.2. Set's Operations

Sets also support a wide variety of operations such as comparing two variables to see what contents are in one but are not in the other, or contents that are in both.

Take a look at the examples below:

(Continued in the next page)

```
#fast membership testing
print('orange' in basket) #True
print('crabgrass' in basket) #False


# Demonstrate set operations on unique letters from two words
a = set('abracadabra')
b = set('alacazam')
print(a) #unique letters in a
#{'d', 'c', 'a', 'r', 'b'}

print(a - b) #letters in a but not in b
#{'d', 'b', 'r'}

print(a | b) #letters in either a or b
#{'m', 'z', 'd', 'c', 'a', 'r', 'l', 'b'}

print(a & b) #letters in both a and b
#{'c', 'a'}

print(a ^ b) #letters in a or b but not both
#{'m', 'z', 'd', 'r', 'l', 'b'}
```

Figure 196: Set operations

19.3. Set Comprehension

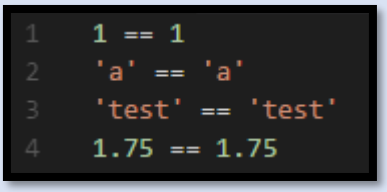
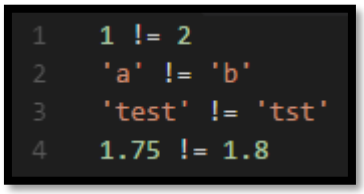
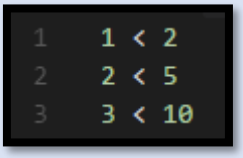
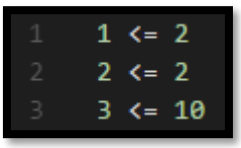
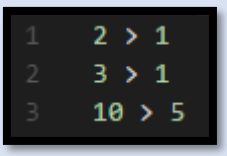
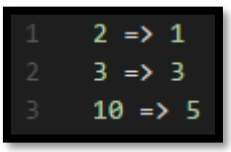
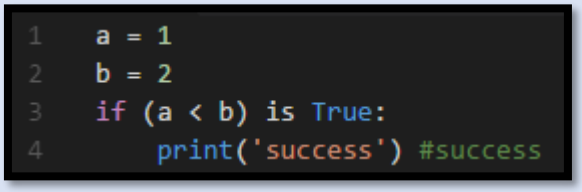
Similarly to list and dictionary comprehensions, set comprehensions are also supported:

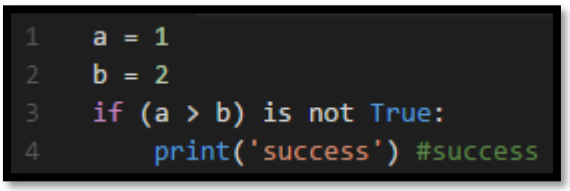
```
a = {x for x in 'abracadabra' if x not in 'abc'}
print(a) #{'r', 'd'}
```

Figure 197: Set comprehension

20. Comparison Operators

This chapter presents and explains the comparison operators available in Python.

Operator	Name	Example
`==`	Equal to (different from `=` which is used to assign a variable).	
`!=`	Different from/ not equal to.	
`<`	Lesser than.	
`<=`	Lesser or equal to.	
`>`	Bigger than.	
`>=`	Bigger or equal to.	
`is`	Object identity.	

`is not`	Negated object identity.	

```
a = 1
b = 2
if (a > b) is not True:
    print('success') #success
```

Table 4: Comparison operators

21. Boolean Operators

This chapter presents and explains the three Boolean operators of Python: `or`, `and` and `not`.

21.1. or

`or`[18]

Tests two arguments: if at least one argument is `True`, then returns `True`, else returns `False`.

```
1   if 1 > 0 or 2 > 1:
2       print(True) #True
3
4   if (1 + 1) != 1 or 2 != 2:
5       print(True) #True
6
7   if 1 < 0 or 2 > 3:
8       print(True)
9   else:
10      print(False) #False
```

Figure 198: or Boolean operator examples

21.2. and

`and`[19]

Tests two arguments: returns `True` if both arguments are `True`, else returns `False`.

(Continued in the next page)

[18] It only evaluates the second argument if the first one is `False`.
[19] It only evaluates the second argument if the first one is `True`.

```
if 1 > 0 and 2 > 1:
    print(True) #True

if (1 + 1) != 1 and 2 == 2:
    print(True) #True

if 1 > 2 and 2 > 3:
    print(True)
else:
    print(False) #False
```

Figure 199: and Boolean operator examples

21.3. not

`not`[20]

Tests one argument: if the argument is `True` returns `False`, else returns `True`.

```
a = 1 > 2
print(not(a)) #True

print(not(1+1 == 2)) #False
```

Figure 200: not Boolean operator examples

[20] `not` has a lower priority than non-Boolean operators, so `not a == b` is interpreted as `not (a == b)`, and `a == not b` is a syntax error.

22. Conditional Clauses

This chapter tackles the conditional clauses found in Python: namely `if`, `elif` and `else`.

22.1. if

`if expression:`

Verifies whether the clause is `True` or `False`: if `True` it executes the following code; if `False` the code is ignored.

A conditional chain always starts with an `if` clause, never with an `elif` or `else` clauses.

```
1  a = 1
2  b = 2
3
4  if a < b:
5      print(a) #outputs 1 because a < b is True
6
7  if a > b:
8      print(a) #outputs nothing because a > b is False
```

Figure 201: if clause examples

In a conditional chain, only the initial `if` clause is needed, both the `elif` (no matter how many) and the final `else` clauses are optional.

In case the `if` returns `True` then the following `elif` and `else` clauses will be completely ignored.

22.2. elif

`elif expression:`

Short for *else if*. Always written after the `if` clause and before the `else` clause.

There can be more than one `elif` clause inside the same conditional chain.

Just like the `if` clause, it verifies whether the condition is `True` or `False`: if `True` executes the following code; if `False` the code is ignored.

For an `elif` clause to be verified it means the preceding conditional clauses were evaluated as `False`.

```
a = 1
b = 2

if a < b:
    #This code is executed since a < b is True
    print(a)
elif a == 1:
    print(b)
    #Even though the condition is True, the condition\
    #is not executed because the preceding condition\
    #(the if) was already executed
```

Figure 202: if/elif example

```
a = 1
b = 2

if a > b:
    #Not executed because the condition is False
    print(a)
elif a == b:
    #Not executed because the condition is False
    print('Equal')
elif a < b:
    #Executed because this is the first clause in the\
    #chain to return True
    print(b)
```

Figure 203: Conditional chain with multiple elif clauses

22.3. else

`else:`

The last clause of a conditional chain. `else` is only executed if the preceding `if` condition and optional `elif` condition(s) returned `False`.

This means the `else` isn't tested at all, that is, if the program reaches the `else` it means the previous conditions returned `False`, therefore the code for the `else` is just executed.

```
a = 1
b = 2

if a >= b:
    #Not executed because the condition is False
    print(a)
else:
    #Executed because the if condition is False
    print(b)
```

Figure 204: if/else example

```
a = 1
b = 2

if a > b:
    #Not executed because the condition is False
    print(a)
elif a == b:
    #Not executed because the condition is False
    print('Equal')
else:
    #Executed because the if condition is False
    print(b)
```

Figure 205: if/elif/else example

23. Functions

This chapter tackles the thematic of functions in Python. It explains some of the theory behind the concept and how it is handled in a practical way in this programming language.

23.1. Theory

The concept of a function is one of the most important ones in mathematics. A common usage of functions in computer languages is to implement mathematical functions. Such a function is computing one or more results, which are entirely determined by the parameters passed to it.

In the most general sense, a function is a structuring element in programming languages to group a set of statements so they can be utilized more than once in a program. The only way to accomplish this without functions would be to reuse code by copying it and adapting it for each case. Using functions usually enhances the comprehensibility and quality of the program. It also lowers the cost for development and maintenance of the software.

It's worth noting functions are known under various names in programming languages: subroutines, routines, procedures, methods and subprograms.

23.2. Function Definition

In Python, to define a new function we use the `def` keyword, followed by the function's name and the parameters (wrapped in parenthesis).

Here's an example of a generic function definition in Python:

```
1   def function_name(parameters):
2       statements a.k.a body of the function
```

Figure 206: Generic function definition

And now some practical examples of function definitions:

```
1   def hello_world():
2       print("Hello World!") #outputs Hello World!
3
4   hello_world()
```

Figure 207: Function definition example (1)

```
def print_evens(x):
    if (x % 2) == 0:
        print(x)
    else:
        print('Not even.')

print_evens(10) #outputs 10
print_evens(3) #outputs Not even.
print_evens(4) #outputs 4
```

Figure 208: Function definition example (2)

From the examples shown above we conclude the following:

- A function may or may not take in parameters/arguments;
- To call a function in a program just write the name of the function as shown in the examples:
    - If the function takes in arguments, then when calling for the function it needs to be given as many arguments as written in the function definition (in the example, `def print_evens(x):`, the function takes one argument, `x`, so each time the function is called it needs to passed one);
    - A function may be called as many times as necessary.

23.3. Recursion

Recursion is the process of defining something in terms of itself. Applying this concept in the context of Python, we obtain recursive functions: function that call themselves.

Take a look at a simple example[21] of a recursive function:

[21] Factorial of a number x is the sum of the product of all the integers from 1 to x inclusive.

```python
#a function to calculate the factorial of a given number

def calc_factorial(x):
    #if x is 1 then immediately return 1, \
    #and no more actions are needed to obtain its factorial
    if x == 1:
        return 1
    #else, we need to multiply x * (x-1) * (x-2)*...* 1
    else:
        return (x * calc_factorial(x-1))
num = 3
print("The factorial of", num, "is", calc_factorial(num))
#The factorial of 3 is 6

#in this case the following steps are executed:
#calc_factorial(3) -> \
#3 * calc_factorial(2) -> \
#3 * (2 * calc_factorial(1)) -> \
#3 * (2 * 1) -> \
#3 * 2 -> \
#6
```

Figure 209: Recursive function example

In this example, the recursion ends when the number is reduced to 1. This is called the base condition.

Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

Advantages of recursion:

- Recursive functions make the code look clean and elegant;
- Complex tasks can be broken down into simpler “sub-tasks” using recursion;
- Sequence generation is easier with recursion than with nested iteration.

Disadvantages of recursion.

- The logic in recursion is sometimes hard to follow;
- Recursive calls are inefficient: they consume a lot of memory and time;

- Recursive functions are hard to debug.

23.4. Anonymous Function (Lambda)

Anonymous functions, also known as lambda expressions, are small functions created with the `lambda` keyword.

`lambda` functions can be used wherever function objects are required. However, they are syntactically restricted to a single expression.

The syntax `lambda arguments: expression` yields an unnamed function object. The unnamed object behaves like a function object defined in the "normal" way:

```
1   def lambda_function(arguments):
2       return expression
```

Figure 210: Generic function definition

which is the usual syntax you're used to see when defining functions.

For the syntax to define a `lambda` function, it's the following:

```
1   #general syntax
2   lambda arguments: expression
```

Figure 211: Generic lamda function definition

Below you can see some practical uses and comparisons between using `lambda` functions and "normal" functions:

(Continued in the next page)

```python
example1 = range(16)

print(list(filter(lambda x: x%3 == 0, example1)))
#[0, 3, 6, 9, 12, 15]

def example1_function(x):
    x_filtered = []
    for item in x:
        if item%3 == 0:
            x_filtered.append(item)
    return x_filtered

print(example1_function(example1))
#[0, 3, 6, 9, 12, 15]
```

Figure 212: lambda functions example (1)

```python
example2 = ['HTML', 'Python', 'Ruby']

print(list(filter(lambda x: x == 'Python', example2)))
#['Python']

def example2_function(y):
    y_filtered = []
    for item in y:
        if item == 'Python':
            y_filtered.append(item)
    return y_filtered

print(example2_function(example2))
#['Python']
```

Figure 213: lambda functions example (2)

23.5. Closure Functions

In some cases, there will be a need to have nested functions, that is, functions defined inside functions. These nested functions have access to variables of the enclosing/outer function. In Python, these non-local variables are read only by default and must be declared explicitly as non-local in order to modify them (using the `nonlocal` statement).

For example, in this case:

```
def print_msg(msg): #The outer function
    def printer(): #The nested function
        #The nested function accesses the nonlocal\
        #variable 'msg'
        print(msg)
    #After defining the nested function, it is called
    printer()

#Call the 'print_msg' function passing it a string
print_msg("Hello") #Hello
```

Figure 214: Nested function definition example

The nested function, `printer()`, accesses the argument (`msg`) of the outer function, `print_msg()`, and prints it.

However, what if instead of calling `printer()` at the end, it returned `printer` instead?

```
def print_msg(msg): #The outer function
    def printer(): #The nested function
        #The nested function accesses the nonlocal\
        #variable 'msg'
        print(msg)
    #After defining the nested function, it is returned
    return printer

#Save the 'print_msg' call to a variable, passing it an argument
alternative = print_msg("Hello")
#Then call the 'alternative' function object
alternative() #Hello
```

Figure 215: Closure function example (1)

Wait, we still got `Hello` as the output? The `print_msg()` function was first called with the string `Hello` and the returned function was bound or assigned to `alternative`. On calling `alternative()`, the content was still remembered although the `print_msg()` had already finished executing.

This technique by which some data (in this case the string `Hello`) gets attached to the code is called *closure* in Python. This value in the enclosing scope is remembered even when the variable goes out of scope or the function itself is removed from the current namespace.

When do we have closure though? As seen in the previous example (Figure 215), there's closure when a nested function references a value in its enclosing scope. From this, we can extract three necessary points that need to be met to have closure:

- There must be a nested function (a function inside a function)
- The nested function must refer to a value defined in the enclosing function
- The enclosing function must return the nested function

Still, after learning what a closure function is, what is it actually used for? Well, for one, they can avoid the use of global values and thus provide some form of data hiding. On the other hand, in cases that involve few methods (generally one method) to be implemented in a class, closures can provide an alternate and more elegant solution. Though, as the number of attributes and methods increase, implementing a class clearly becomes the better option.

Here is an example where a closure function might be more preferable than defining a class and creating objects. Of course, which one to use is all up to you.

```
1   def make_multiplier_of(n):
2       def multiplier(x):
3           return x * n
4       return multiplier
5
6   #Call 'make_multiplier_of()' twice and bound it to variables passing\
7   #different values
8   times3 = make_multiplier_of(3)
9   times5 = make_multiplier_of(5)
10
11  #Then call those function objects passing values to the nested funtion
12  print(times3(9)) #27
13  print(times5(3)) #15
14  print(times5(times3(2))) #30
```

Figure 216: Closure function example (2)

It is worth noting *decorators* in Python make extensive use of closure as well (you can read more about decorators in its own chapter, Chapter 28).

Lastly, the topic of security needs to be recovered since closure is not completely secure. All function objects have a `__closure__` attribute that returns a tuple of cell objects, if it is a closure function. Referring to the previous example (Figure 216), we know both `times3` and `times5` are closure functions. This way, we can extract which value was passed to the functions using the `cell_contents` attribute from the cell object:

```
1  def make_multiplier_of(n):
2      def multiplier(x):
3          return x * n
4      return multiplier
5
6  #Call 'make_multiplier_of()' twice and bound it to variables passing\
7  #different values
8  times3 = make_multiplier_of(3)
9  times5 = make_multiplier_of(5)
10
11 print(times3.__closure__) #(<cell at 0x02F8C7D0: int object at 0x63F868A0>,)
12 print(times5.__closure__) #(<cell at 0x053C66B0: int object at 0x63F868C0>,)
13 print(times3.__closure__[0].cell_contents) #3
14 print(times5.__closure__[0].cell_contents) #5
```

Figure 217: Extract values used in closure functions

24. Loops

In this chapter the thematic of loops in Python is brought to light, namely the `for` loops and `while` loops, and their intricacies.

24.1. for Loops

Rather than always iterating over an arithmetic progression of numbers or giving the user the ability to define both the iteration step and halting condition, Python's `for` statement iterates over the items of any sequence (e.g. a list or a string), in the order that they appear in the sequence.

Below is an example of the general syntax for a `for` loop and a couple of practical examples.

```
#general syntax
for variable in iterable:
    #write code here

words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))
#the code above outputs:
''' cat 3
    window 6
    defenestrate 12'''

list_1 = [1,2,3,4,5]
for item in list_1:
    print(item)
#the code above outputs
''' 1
    2
    3
    4
    5'''
```

Figure 218: for loop examples

While a `for` loop can be used to iterate through an entire list or a string, most times you'll want to repeat some code a certain amount of times (for example, if you want to ask the user for input five times). In this cases the `for` loop will go hand-in-hand with the `range()` type. What `range()` does in these cases is telling the Python interpreter how many times that `for` loop will be repeated.

Take a look at the example below to see `range()` in action, and in case you want to learn more about this Python built-in constructor read its own section (Chapter 10.24).

```
for x in range(5):
    year = input("What's the current year?")
    birth_year = input("What's birth year?")
    age = int(year) - int(birth_year)
    print("Your age is", age)

#In this example the for loop will be executed 5 times. What it does
#is ask the user for the input of 2 items: year and birth_year.
#After the input the program calculates the age of the user
#(not taking months into account of course). After this the progam
#outputs a single line that tells the user their age.
```

Figure 219: for loop using range() example

24.1.1. for/else Loops

Another element you can combine with a `for` loop is an `else` conditional clause.

The construction for this combination is similar to the one used in `if`/`else`, and by that, I mean in a `for`/`else` the `else` will only be executed if the `for` condition returns `False`. This means that for the `else` to be executed the `for` loop can't be terminated with a `break` statement, it needs to terminate normally (by exhausting the initial condition).

```
for i in range(5):
    print(i)
else:
    print('Finished.')
#outputs the numbers 0 to 4 and the word Finished. in the end

for i in range(5):
    if i == 3:
        break
    print(i)
else:
    print('Finished.')
#only outputs 0, 1 and 2
```

Figure 220: for/else loops examples

In the first example after outputting the numbers `0` to `4` the program will output `Finished.`, which means the `else` was executed. However, in the second example because `i == 3` during the fourth loop, the loop will be terminated with the `break`, meaning the `else` won't be executed (it outputs only `0`, `1` and `2`).

24.2. while Loops

The `while` statement, unlike with the `for` case, is used for repeated execution of code while the initial condition is `True`.

```
#general syntax
while 'condtion':
    #execute code

i = 0
while i < 10:
    print(i, end=' ')
    i += 1
#this loop outputs 0 1 2 3 4 5 6 7 8 9 ; when i == 10 the condition returns False
#and the code in the while loop won't be executed
```

Figure 221: while loop example

24.2.1. Infinite Loops

There's a very important type of loop to keep in mind when creating a `while` loop: *infinite loops*.

These are loops with conditions which will always return `True`, therefore the loop's code will keep on being executed. These can turn out to be a reason why a program or application keeps "crashing". Below are some examples of infinite loops:

(Continued in the next page)

```
while 1 < 2:
    print("Infinite Loop!")
#since 1<2 will always return True the loop will keep
#outputing the same string, eventually leading to a crash

 i = 0
 while i < 10:
     print(i)
#this is another example of an infinite loop; i is assigned the value 0,
#and because i never gets incremented inside the loop, it means
#i will always be 0, which means i<10 will always return True,
#hence another case of an infinite loop
```

Figure 222: Infinite loops examples

A way to deal with some infinite loops could be the usage of the `break` statement. Let's pick the first example, `while 1 <2`, and let's say that for some reason we actually want to output `Infinite Loop!` under that condition, but we only want to output this once. With the `break` statement this turns out to be simple: just add the keyword `break` below the `print()` and voila, no more infinite loops (at least for this case):

```
while 1 < 2:
    print("Infinite Loop!")
    break
#with the break statement the program will only output
#Infinite Loop! once and after the loop is terminated
```

Figure 223: Solving an infinite loop with the break statement (1)

In case for some reason you wanted to use this loop to output the `Infinite Loop!` string more than once without creating an infinite loop you could, for example, use a `counter` variable with an `if` nested inside the loop:

```
counter = 0 #start the counter at 0

while 1 < 2:
    print("Infinite Loop!")
    if counter  == 2: #let's say we only wanted to output three times
        break
    counter += 1 #here we add 1 to counter to make sure each time the loop is
    #executed the counter variable will be incremented
```

Figure 224: Solving an infinite loop with the break statement (2)

24.2.2. while/else Loops

Another element you can combine with a `while` loop is an `else` conditional statement.

The construction for this combination is similar to the one used in `for`/`else` loops, and by that, I mean in a `while`/`else` the `else` will only be executed if the `while` returns `False`.

This means that for the `else` to be executed one of two conditions must be met: (1) the `while` isn't executed, or (2) the `while` doesn't return `True` anymore and returns `False` instead.

```
from random import randint
count = 0

while count < 3:
    num = randint(1,6)
    print(num)
    if num == 5:
        print('You lose.')
        break
    count += 1
else:
    print('You win.')

#this is a simple program that outputs 3 random numbers between 1 and 5, one number at a time
#if one of the numbers is 5 then the programs outputs 'You lose.' and the loop is terminated
#if all the 3 random numbers aren't 5 then an else is executed, which outputs the message
#'You win.'
```

Figure 225: while/else loop example

In the example shown above, the `else` would only be executed if the none of the three random numbers was `5`. If at least one of them turned out to be `5` then the `while` loop would output a simple message to the user and terminate (because of the `break` statement). In this case,

the `else` wouldn't be executed: when a loop is terminated because of a `break` then the statement will automatically skip the `else`.

25. Imports

These chapter presents the different ways of importing code, more specifically in this case, importing modules.

In Python, there are 3 ways to import modules and its functions and variables: generic, function and universal imports.

25.1. Generic Import

Imports entire modules.

```
1  import math
2  print(math.sqrt(25)) #outputs 5.0
```

Figure 226: Generic import example

25.2. Function Import

Imports specific functions of a certain module.

```
1  from math import sqrt
2  print(sqrt(25)) #outputs 5.0
```

Figure 227: Function import example

25.3. Universal Import

Imports every function and variable in the module (this type of import isn't advised because it may result in function and class overwriting; it also uses much more memory resources).

```
1  from math import *
2  print(sqrt(25)) #outputs 5.0
```

Figure 228: Universal import example

26. Bitwise Operations

This chapter tackles bitwise operations applied to Python, from the theory to the practical cases, including the operations themselves and "tools" such as bit masks.

26.1. Theory

Bitwise operations are, just like the name implies, operations related with bits. These can go from simple base 10 to base 2 conversions to more complex operations such as bit shifts or creating bit masks.

For starters, let me show how to write binary/base 2 numbers in Python:

```
#general syntax to write a binary number
#0b1 (simply write the binary number, but add 0b as a prefix)

0b1 #binary number 1, which corresponds to 1 in base 10
0b10 #binary number 10, which corresponds to 2 in base 10
0b11 #binary number 11, which corresponds to 3 in base 10
0b100 #binary number 100, which corresponds to 4 in base 10
0b101 #binary number 101, which corresponds to 5 in base 10
```

Figure 229: How to write binary numbers in Python

However, when you try to output directly a binary number, for example, `print(0b11)`, you'd expect the program to output `0b11`. However, what happens is that number is converted to base 10, which in this case would output `3`.

In order to output the number in base 2, use the `bin()` function. The same applies for normal operations such as addition or subtraction when using binary numbers: if `bin()` isn't used then the program will just output the base 10 numbers. Look at the examples below to get a better idea of this:

```
print(0b1 + 0b11) #it is interpreted as 1 + 3, which is 4
print(bin(0b1 + 0b11)) #outputs 0b100, which is 4 written as a base 2 number

print(0b1-0b11) #it is interpreted as 1 - 3, which is -2
print(bin(0b1-0b11)) #outputs -0b10, which is -2 written as a base 2 number

print(0b11 * 0b11) #it is interpreted as 3 * 3, which is 9
print(bin(0b11 * 0b11)) #output 0b1001, which is 9 written as a base 2 number

print(0b100 // 0b10) #it is interpreted as 4 // 2, which is 2
print(bin(0b100 // 0b10)) #output 0b10, which is 2 written as a base 2 number;
# used floor division otherwise it would return a float (2.0);
# only integers can be used as arguments for bin()
```

Figure 230: Output base 2 integers with bin()

Another important thing to keep in mind about binary numbers is that, as you know, they are written with ones and zeros. What's worth noting is that a bit (digit) being `1` means that bit is `True` (or turned on) while a bit being `0` (zero) means that bit is `False` (or turned off).

Before going into detail about each possible operation, look at the following table, which contains all bitwise operations available in Python, sorted by ascending priority:

Operation	Result
`x | y`	Bitwise `or` of `x` and `y`.
`x ^ y`	Bitwise exclusive `or` of `x` and `y`.
`x & y`	Bitwise `and` of `x` and `y`.
`x << n`	`x` shifted left by `n` bits.
`x >> n`	`x` shifted right by `n` bits.
`~x`	The bits of `x` inverted (`-(x+1)`).

Table 5: Bitwise operations

26.2. Or (|)

The logical structure for this operator is quite similar to the "normal" `or` Boolean operator, except here it is used to compare two binary numbers, bit by bit.

The bit comparisons will be `True` (`1`) if one of the bits is `True` (`1`), else the comparison is `False` (`0`).

```
#general syntax
x | y

binary_1 = 0b101010 #42 in base 10
binary_2 = 0b001111 #15 in base 10
#what happens here is that the or (|) compares binary_1 and binary_2 bit by bit
#in each comparison, if one of the bits is True/1 then it returns 1, else it returns 0
#i wrote those comparions next and the results below each one
#(1,0), (0,0), (1,1), (0,1), (1,1), (0,1)
#  1      0      1      1      1      1
print(bin(binary_1 | binary_2)) #0b101111
print(binary_1 | binary_2) #47 in base 10

binary_3 = 0b110 #6 in base 10
binary_4 = 0b1010 #10 in base 10
#(0,1), (1,0), (1,1), (0,0)
#  1      1      1      0
print(bin(binary_3 | binary_4)) #0b1110
print(binary_3 | binary_4) #14 in base 10
```

Figure 231: Bitwise Or examples

Here's a summary of all the possible outcomes while using bitwise *or*:

- `0 | 0 == 0`
- `0 | 1 == 1`
- `1 | 1 == 1`
- `1 | 0 == 1`

26.3. Xor (^)

This operator is similar to the previous *or*, except XOR compares two binary numbers, bit by bit, while looking for different bits. Then, the bit comparisons will be `True` (`1`) if both bits are different, else the comparisons are `False` (`0`).

```
#general syntax
x ^ y

binary_1 = 0b101010 #42 in base 10
binary_2 = 0b001111 #15 in base 10
#what happens here is that the XOR (^) compares binary_1 and binary_2 bit by bit
#in each comparison, if both bits are True/1 or False/0 then it returns 0, else it returns 1
#i wrote those comparions next and the results below each one
#(1,0), (0,0), (1,1), (0,1), (1,1), (0,1)
#  1      0      0      1      0      1
print(bin(binary_1 ^ binary_2)) #0b100101
print(binary_1 ^ binary_2) #37 in base 10

binary_3 = 0b110 #6 in base 10
binary_4 = 0b1010 #10 in base 10
#(0,1), (1,0), (1,1), (0,0)
#  1      1      0      0
print(bin(binary_3 ^ binary_4)) #0b1100
print(binary_3 ^ binary_4) #12 in base 10
```

Figure 232: XOR examples

Here's a summary of all the possible outcomes while using XOR:

- `0 ^ 0 == 0`
- `0 ^ 1 == 1`
- `1 ^ 1 == 0`
- `1 ^ 0 == 1`

26.4. And (&)

Just like with the *or*, the *and* works quite similarly to its Boolean operator counterpart `and`. Bitwise, the *and* compares two binary numbers, bit by bit, and, if both bits are `True` (`1`) then the result is also `1`, else the result is `False` (`0`).

```
#general syntax
x & y


binary_1 = 0b101010 #42 in base 10
binary_2 = 0b001111 #15 in base 10
#what happens here is that the and (&) compares binary_1 and binary_2 bit by bit
#in each comparison, if both bits are True/1 it returns 1, else it returns 0
#i wrote those comparions next and the results below each one
#(1,0), (0,0), (1,1), (0,1), (1,1), (0,1)
#  0      0      1      0      1      0
print(bin(binary_1 & binary_2)) #0b001010 or just 0b1010
print(binary_1 & binary_2) #10 in base 10

binary_3 = 0b110 #6 in base 10
binary_4 = 0b1010 #10 in base 10
#(0,1), (1,0), (1,1), (0,0)
#  0      0      1      0
print(bin(binary_3 & binary_4)) #0b0010 or just 0b10
print(binary_3 & binary_4) #2 in base 10
```

Figure 233: Bitwise And examples

Here's a summary of all the possible outcomes using *and*:

- `0 & 0 == 0`
- `0 & 1 == 0`
- `1 & 1 == 1`
- `1 & 0 == 0`

26.5. Bit Shifts (<< and >>)

This is used to move all the bits (digits) in a binary number either to the right or to the left *n* numbers of times.

This operation can be used with both binary numbers and base 10 integers (the program will convert the bases on its own).

Take a look at the examples below:

```
#general syntax for left shift
binary_number or base 10 integer << number of units to be moved


print(0b001 << 2) #4; this means every digit in the binary number 001
#will be moved 2 units to the left (which is 00100 or just 100)
print(1 << 2) #4; exactly the same as the previous example, except
#this time the number is written in base 10
print(bin(0b001 << 2)) #0b100; if you want to output the base 2 number,
#you still need to use the bin() function

#general syntax for right shift
binary_number or base 10 integer >> number of units to be moved

print(0b100 >> 2) #1; this is the exact same thing that happens with the left shift,
#instead here the digits are moved 2 units to the right (so 100 becomes 001 or just 1)
print(4 >> 2) #1; exactly the same as the previous example, except
#this time the number is written in base 10
print(bin(0b100 >> 2)) #0b1; if you want to output the base 2 number,
#you still need to use the bin() function
```

Figure 234: Bit shifts examples

It's also worth noting three things about bit shifts:

- Negative shift counts are illegal and cause a `ValueError` to be raised
- A left shift by *n* bits is equivalent to multiplication by `pow(2, n)` without overflow check
- A right shift by *n* bits is equivalent to division by `pow(2, n)` without overflow check

26.6. Not (~)

This is also similar to the Boolean operator *not*, except here it's applied to bitwise operations.

The simplest way to describe how it works is that it flips all the bits in a number. Since this is much more complicated for the programs, put simply it's equivalent to adding 1 to the number and then making it negative (`-(x+1)`).

```python
#general syntax
~x

#simply put: ~x == -(x+1)
print(~1) #-2
print(bin(~1)) #-0b10 == -2 in base 10
print(~2) #-3
print(bin(~2)) #-0b11 == -3 in base 10
print(~3) #-4
print(~42) #-43
print(~123) #-124
```

Figure 235: Bitwise Not examples

26.7. Bit Mask

This is not considered a bitwise operation, rather, it's something to help you execute those operations.

A bit mask is just a variable that helps you manipulate bits and obtain the results you want.

These bit masks can be used, for example, to turn specific bits on, turn others off, or just collect data from an integer about which bits are on or off.

For example, you could use the code below to turn on a specific bit in a given position of a binary number:

```python
#Function to turn on the bit number 'n' (counting from the right) of 'number'

def flip_bit(number, n):
    #Create a binary number (the "mask") that only has one bit\
    #turned on: the one at the wanted position
    mask = (0b1 << (n-1))
    #Use XOR to turn on that bit in the input number; then save it to a new variable
    result = number ^ mask
    return bin(result)

print(flip_bit(0b10, 4)) #0b1010
print(flip_bit(0b1010, 3)) #0b1110
```

Figure 236: Bit mask example

27. Classes

In this chapter, the very important concept of *class* is tackled. It explores the theory behind classes and its related concepts, the likes of *namespace* and *attribute*, the practical uses of classes and more.

27.1. Theory

Python classes provide all the standard features of Object Oriented Programming: the *class inheritance* mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a *method* can call the method of a base class with the same name.

Objects can contain arbitrary amounts and kinds of data.

As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime and can be modified further after creation.

Now let me give you some definitions related to classes: namespace, attribute, scope and variable binding.

27.1.1. Namespace

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future.

Examples of namespaces are: the set of built-in names (containing functions such as `abs()`, and built-in exception names); the global names in a module; and the local names in a function invocation.

In a sense, the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces: for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name (if the function comes from a `maxmin` module, to call the function `maxmin.maximize` would be the syntax used).

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up and is never deleted.

The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace.

The local namespace for a function is created when the function is called and deleted when the function returns or raises an exception that is not handled within the function (forgetting would be a better way to describe what actually happens). Of course, recursive invocations each have their own local namespace.

27.1.2. Attribute

An attribute is any name following a dot. For example, in the expression `z.real`, *`real`* is an attribute of the object `z`.

Strictly speaking, references to names in modules are *attribute references*: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace.

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible.

Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by *`modname`*.

27.1.3. Scope

A *scope* is a textual region of a Python program where a namespace is directly accessible. "Directly accessible" here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are at least four nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first; contains the local names;

- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope; contains non-local, but also non-global names;
- the next-to-last scope contains the current module's global names;
- the outermost scope (searched last) is the namespace containing built-in names;

If a name is declared using the `global` statement, then all references and assignments go directly to the middle scope containing the module's global names.

To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared `nonlocal`, those variables are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at "compile" time, so don't rely on dynamic name resolution.

A special quirk of Python is that – if no `global` statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should be rebound there.

27.1.4. Variable Binding

Next, let me show you a practical example of how the `global` and `nonlocal` statements can affect variable binding:

```
1   def scope_test():
2       def do_local():
3           spam = "local spam"
4
5       def do_nonlocal():
6           nonlocal spam
7           spam = "nonlocal spam"
8
9       def do_global():
10          global spam
11          spam = "global spam"
12
13      spam = "test spam"
14      do_local()
15      print("After local assignment:", spam)
16      do_nonlocal()
17      print("After nonlocal assignment:", spam)
18      do_global()
19      print("After global assignment:", spam)
20
21  scope_test()
22  print("In global scope:", spam)
```

Figure 237: global and nonlocal statements example

```
24  #outputs
25  #After local assignment: test spam
26  #After nonlocal assignment: nonlocal spam
27  #After global assignment: nonlocal spam
28  #In global scope: global spam
```

Figure 238: Figure 237's code output

What happened in this example was that the local assignment (which is default) didn't change `scope_test`'s binding of `spam`. The `nonlocal` assignment changed `scope_test`'s binding of `spam`, and the `global` assignment changed the module-level binding. You can also see that there was no previous binding for `spam` before the `global` assignment.

27.2. Class Definition

And now let's pass to the practical stuff. Let's start by taking a look at the syntax needed to create the most basic class possible:

```
#general syntax
class ClassExample:
    statement 1
    ...
    statement n
```

Figure 239: Generic class definition

Now let's analyze each component:

- When starting a class you need to start with the `class` keyword, just like using the `def` keyword to start a function definition;
- Then there's the name of class, in this case it's `ClassExample`. Please remember that the classes you create should always start with a capitalized letter;
- The rest is simply the "content" of the class.

Class definitions, like function definitions (`def` statements) must be executed before they have any effect.

In practice, the statements inside a class definition will usually be function definitions, but other statements are also allowed. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for *methods*.

When a class definition is entered, a new namespace is created, and used as the local *scope* — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the end), a class object is created. This is basically a wrapper around the contents of the namespace created by the class definition. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (*`ClassExample`* in the example shown).

27.3. Class Objects

Class objects support two kinds of operations: *attribute references* and *instantiation*.

27.3.1. Attribute References

Attribute references use the standard syntax used for all attribute references in Python: `obj.name` (dot notation).

Valid attribute names are all the names that were in the class' namespace when the class object was created. So, if the class definition looked like this:

```
class MyClass:
    i = 12345
    def f(self):
        return 'hello world'
```

Figure 240: Basic class example

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment.

27.3.2. __init__() and Instantiation

The instantiation operation ("calling" a class object) creates an empty object.

Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

```
def __init__(self):
    self.example = []
```

Figure 241: __init__() example

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly-created class instance. So, in this example, a new initialized instance can be obtained by:

```
x = MyClass()
```

Figure 242: Class instantiation example

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example:

```
class Complex():
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart

x = Complex(3.0, -4.5)

print(x.r, x.i) #3.0 -4.5
```

Figure 243: __init__() with arguments

27.4. Instance Objects

The only operations understood by instance objects are attribute references. There are two kinds of valid attribute references: data attributes and methods.

27.4.1. Data Attributes

Data attributes don't need to be declared: like local variables, they spring into existence when they are first assigned to.

For example, if `x` is the instance of `MyClass` created above, the following piece of code will print the value `16`, without leaving a trace:

```python
class MyClass:
    i = 12345
    def f(self):
        return 'hello world'

x = MyClass()
x.counter = 1

while x.counter < 10:
    x.counter = x.counter * 2
    #while in the loop, x would have the values:
    #1, 2, 4, 8 and 16 (which terminates the loop)

print(x.counter) #16
del x.counter
```

Figure 244: Data attributes example

27.4.2. Methods

The other kind of instance attribute reference are *methods*.

A method is a function that "belongs to" an object. In Python, the term is not unique to class instances: other object types can have methods as well.

For example, list objects have methods called `.append()`, `.insert()`, `.remove()`, `.sort()` and so on. However, in the following discussion, the term method will be used exclusively to refer to methods of class instance objects, unless explicitly stated otherwise.

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances.

So, in the same example from Figure 244, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not a method (it is an integer). But `x.f` is not the same thing as `MyClass.f` — it is a method object, not a function object.

Often, the first argument of a method is called `self`. This is nothing more than a convention: the name `self` has absolutely no special meaning to Python. Note, however, that by not following the convention your code may be less readable to other Python programmers, and it

is also conceivable that a class browser program might be written in a way that relies upon such a convention.

27.5. Method Objects

Usually, a method is called right after it is bound:

```
x.f()
```

Figure 245: Method object example

Once again, in the `MyClass` example from Figure 244, this will return the string `hello world`.

However, it is not necessary to call a method right away: `x.f` is a method object and so can be stored away to be called later. For example:

```
class MyClass:
    i = 12345
    def f(self):
        return 'hello world'

x = MyClass()
x.counter = 1

while x.counter < 10:
    x.counter = x.counter * 2
    #while in the loop, x would have the values:
    #1, 2, 4, 8 and 16 (which terminates the loop)

print(x.counter) #16
del x.counter


xf = x.f #x.f. is now stored in the xf variable
while True:
    print(xf())
```

Figure 246: Store method objects to be used later

will continue to output `hello world` until the end of time.

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f()` specified an argument. What happened to the argument? The special thing about methods is that the instance object is passed as the first argument of the function.

In our example, the call `x.f()` is equivalent to *`MyClass.f(x)`*. In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's instance object before the first argument.

If you still don't understand how methods work, a look at the implementation can perhaps clarify matters. When an instance attribute is referenced and it isn't a data attribute, its class is searched. If the name denotes a valid class attribute that is a function object, a method object is created by packing the instance and function objects just found together in an abstract object: this is the method object.

When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, so the function object is called with this new argument list.

27.6. Class and Instance Variables

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

```
1   class Dog:
2       kind = 'canine' #class variable shared by all instances
3
4       def __init__(self, name): #instance variable unique to each instance
5           self.name = name
6
7   dog_1 = Dog('Fido')
8   dog_2 = Dog('Buddy')
9
10  print(dog_1.kind) #outputs 'canine'; this is shared by all Dogs
11  print(dog_1.name) #outputs 'Fido'; unique to dog_1
12  print(dog_2.kind) #outputs 'canine'; this is shared by all Dogs
13  print(dog_2.name) #outputs 'Buddy'; unique to dog_2
```

Figure 247: Class variables example

Shared data can have possibly surprising effects with involving mutable objects such as lists and dictionaries.

For example, the `tricks` list in the following code should not be used as a class variable because just a single list would be shared by all `Dog` instances:

```
class Dog:
    tricks = [] #mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

dog_1 = Dog('Fido')
dog_2 = Dog('Buddy')
dog_1.add_trick('roll over')
dog_2.add_trick('play dead')
print(dog_1.tricks) # ['roll over', 'play dead']
```

Figure 248: Incorrect usage of class variables

For correct usage, the following code should be used instead, which uses an instance variable for the list:

(Continued in the next page)

```python
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = [] #creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

dog_1 = Dog('Fido')
dog_2 = Dog('Buddy')
dog_1.add_trick('roll over')
dog_2.add_trick('play dead')
print(dog_1.tricks) #['roll over']
print(dog_2.tricks) #['play dead']
```

Figure 249: Correct usage of class variables

27.7. Random Remarks

Data attributes override method attributes with the same name; to avoid accidental name conflicts, which may cause hard-to-find bugs in large programs, it is wise to use some kind of convention that minimizes the chance of conflicts.

Possible conventions include capitalizing method names, prefixing data attribute names with a small unique string (perhaps just an underscore) or using verbs for methods and nouns for data attributes.

There is no shortage of means for referencing data attributes (or other methods) from within methods. Thus, this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also acceptable.

For example:

```
#function defined outside of a class
def f1(self, x, y):
    return min(x, x+y)

class C1:
    f = f1
    def g(self):
        return 'hello world'
    h = g
```

Figure 250: Function object defined outside of a class

Now `f`, `g` and `h` are all attributes of the class `C1` that refer to function objects, and consequently they are all methods of instances of `C1` — with `h` being exactly equivalent to `g`. Note that this practice usually only serves to confuse the reader of a program.

Methods may call other methods by using method attributes of the `self` argument:

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Figure 251: Methods calling other methods

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing its definition (due note a class is never used as a global scope).

While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in the global scope.

Each value is an object, and therefore has a class (also called its *type*). It is stored as `object.__class__`.

27.8. Inheritance

Of course, a language feature would not be worthy of the name class without supporting *inheritance*. The syntax for a derived class definition looks like this:

```
#general syntax
class DerivedClass(BaseClass):
    statement 1
    ...
    statement n
```

Figure 252: Class inheritance syntax

Now let's analyze each component:

- `DerivedClass` is the name of a new class which will inherit from an existing class (the class(es) wrapped in parenthesis);
- `BaseClass` is the class (or classes) from which `DerivedClass` will inherit;
- The rest is the content of `DerivedClass` (this does not include the data inherited from `BaseClass`).

The name `BaseClass` must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```
class DerivedClass(modulename.BaseClass):
```

Figure 253: Class inheritance from another module

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from another class.

There's nothing special about instantiation of derived classes: `DerivedClass()` creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it.

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call `BaseClass.method(self, arguments)`. This is occasionally useful to clients as well (note that this only works if the base class is accessible as `BaseClass` in the global scope).

Python has two built-in functions that work with inheritance:

1. Use `isinstance()` (Chapter 10.13) to check an instance's type: `isinstance(obj, int)` will be `True` only if `obj.__class__` is `int` or some class derived from `int`;
2. Use `issubclass()` (Chapter 10.14) to check class inheritance: `issubclass(bool, int)` is `True` since `bool` is a subclass of `int`. However, `issubclass(float, int)` is `False` since `float` is not a subclass of `int`.

27.8.1. Multiple Inheritance

Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
#general syntax
class DerivedClass(BaseClass1, BaseClass2, BaseClass3):
    statement 1
    ...
    statement n
```

Figure 254: Multiple inheritance syntax

Please note that inheriting from three base classes was an arbitrary choice, you can choose inherit from more or less base classes depending on the case.

For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy.

Thus, if an attribute is not found in `DerivedClass`, it is searched for in `BaseClass1`, then (recursively) in the base classes of `BaseClass1`, and if it was not found there, it searches in `BaseClass2`, and so on.

In fact, it is slightly more complex than that: the method resolution order changes dynamically to support cooperative calls to `super()`. This approach is known in some other multiple-inheritance programming languages as *call-next-method* and is more powerful than the *super* call found in single-inheritance languages.

Dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class).

For example, all classes inherit from `object`, so any case of multiple inheritance provides more than one path to reach `object`.

To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance.

27.8.2. super() Application Example

```
class Employee(object):
    def __init__(self, employee_name):
        self.employee_name = employee_name

    def calculate_wage(self, hours):
        self.hours = hours
        return hours * 20.00

class PartTimeEmployee(Employee):
    #PartTimeEmployee inherits from Employee
    def part_time_wage(self, hours):
        #this accesses the superclass of PartTimeEmployee, Employee,
        #and executes the calculate_wage method
        return super(PartTimeEmployee, self).calculate_wage(hours)

milton = PartTimeEmployee("Milton")
print(milton.part_time_wage(10))
```

Figure 255: super() example

For more information about this built-in function, read its own section (Chapter 10.28).

27.9. Private Variables

"Private" instance variables that cannot be accessed except from inside an object don't exist in Python.

However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. `_spam`) should be treated as a non-public part of the API whether it is a function, a method or a data member. This should be considered an implementation detail and subject to change without notice.

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called *name mangling*. Any identifier of the form `__spam` (at least two leading underscores, at most one) is textually replaced with `_classname__spam`, where `classname` is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. For example:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update #private copy of original update() method

class MappingSubclass(Mapping):
    #MappingSubclass inherits from Mapping

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

Figure 256: Name mangling example

Note that while the mangling rules are designed mostly to avoid accidents, it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

Notice that code passed to `exec()` or `eval()` does not consider the `classname` of the invoking class to be the current class; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to `getattr()`, `setattr()` and `delattr()`, as well as when referencing `__dict__` directly.

27.10. Odds and Ends

Sometimes it is useful to have a data type similar to the Pascal “record” or C “struct”, bundling together a few named data items.

An empty class definition will do nicely:

```
class Employee:
    pass

john = Employee()  # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000

print(john.name, john.dept, john.salary)
#John Doe computer lab 1000
```

Figure 257: Empty class to simulate C's struct data type

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that get the data from a string buffer instead, and pass it as an argument.

Instance method objects have attributes too: `m.__self__` is the instance object with the method `m()`, and `m.__func__` is the function object corresponding to the method.

27.11. property() and @property

Once you have created a class, you will probably want to set attributes to a certain value and maybe later you will want to get that value or even delete it. Well, according to what we've learned so far, we can set an attribute using *`object.attribute = value`*, get an attribute using *`object.attribute`* and delete an attribute using `del` *`object.attribute`*.

The thing is, we could go one step further and create detailed methods for these three operations: *setter*, *getter* and *deleter* methods, respectively. In a very basic example, what was just said could be put into practice like this:

```
class C:
    def __init__(self):
        self.x = None

    #Getter method
    def getx(self):
        return self.x

    #Setter method
    def setx(self, value):
        self.x = value

    #Deleter method
    def delx(self):
        del self.x

c = C() #Create an instance of C
c.x = 30 #Set the attribute 'x' to 30
print(c.x) #30; Get the value of 'x'
del c.x #Delete the 'x' attribute
```

Figure 258: Class with getter, setter and deleter methods

The thing is, this way we aren't using any of those methods for these operations. What if you wanted to print a message to the user before returning `self.x`? You could specifically call the method you want, but that would defeat the purpose of simplifying things. Then you'd use either the `property()` function or a decorator, the `@property` decorator to be more precise.

27.11.1. property()

Starting with `property()`, this function returns a property attribute when called. However, when you set up the function, you feed it the different methods we were talking about: a getter, a setter, a deleter and even a docstring if you want to.

This function was already described in an earlier section, **property()** (Chapter 10.23), but as a refresher, the syntax for this function is `property(fget=None, fset=None, fdel=None, doc=None)`, where `fget` corresponds to the getter method, `fset` to the setter, `fdel` to the deleter and `doc` to the docstring. All of these parameters are optional.

So if we now refactor the class using `property()` and add some `print()` statements in each method to "prove" they are actually being invoked by `property()`, we will obtain this:

```
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        print('GET')
        return self._x

    def setx(self, value):
        print('SET')
        self._x = value

    def delx(self):
        print('DEL')
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")


c = C() #Create an instance of C
c.x = 30 #Set the attribute '_x' to 30; Prints 'SET'
print(c.x) #Get the value of '_x'; Prints 'GET' and 30
del c.x #Delete the '_x' attribute; Prints 'DEL'
```

Figure 259: property() example

We successfully made it so that when we operate with the `_x` attribute, the respective methods will be invoked. Notice that while we are calling the `x` attribute, what we are doing with that is calling `property()`, which then invokes the corresponding method to execute the required operation on `_x`.

27.11.2. @property

Now you know you can create getters, setters and deleters using `property()`, but as it says in the beginning, that returns a property attribute. Then if we simply create a property object using `x = property()`, we could assign new methods to it, like this:

```
#Create a property object
x = property()
#Then assign new methods to it
x = x.getter(get_x)
x = x.setter(set_x)

#Which is equivalent to:
x = property(get_x, set_x)
```

Figure 260: Method assignment to a property object

After you read the **Decorators** chapter (Chapter 28), you'll notice this construct could also be implemented as a *decorator*. And that's what we are going to do, we'll use the `@property` decorator to implement the getter, setter and deleter methods. In this section we won't delve into the intricacies of what is a decorator, we'll leave that for the other section. Here we'll focus on the practical side of it.

(Continued in the next page)

```python
class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        print('GET')
        return self._x

    @x.setter
    def x(self, value):
        print('SET')
        self._x = value

    @x.deleter
    def x(self):
        print('DEL')
        del self._x

c = C() #Create an instance of C
c.x = 30 #Set the attribute '_x' to 30; Prints 'SET'
print(c.x) #Get the value of '_x'; Prints 'GET' and 30
del c.x #Delete the '_x' attribute; Prints 'DEL'
```

Figure 261:@property example

What we have here is the exact same thing from the `property()` example, except this time we are using decorators, which means we just need to give the same name to each of the relevant methods and use the correct notation for the decorators.

Just a clarification: the `@property` decorator turns a method into a getter method, `@x.setter` turns a method into a setter method and `@x.deleter` turns a method into a delete method. Of course, instead of `x` you could call it something else, as long you use the same name used for the original property.

28. Decorators

Python has an interesting feature called decorators to add functionality to an existing piece of code. This can also be called metaprogramming since part of the program tries to modify another part at run time.

28.1. Introduction

For learning about decorators, we must first have the concept of higher order functions in mind. These functions are nothing more than functions that take other functions as arguments.

This is a topic that was already tackled in the **Closure Functions** section (Chapter 23.5), but as a practical reminder, when a higher order function receives another function as an argument, it can use the argument in two ways: simply call that second function or return the function (making it a closure function).

```
1   def print_msg(msg):
2       def printer():
3           print(msg)
4       printer()
5
6   def print_msg(msg):
7       def printer():
8           print(msg)
9       return printer
```

Figure 262: Nested functions examples

Both functions and methods are called *callable* since they can be called. In fact, any object that implements the special method `__call__()` is a callable. So, we arrive at the most basic definition of a decorator: a decorator is a callable that returns another callable. Put in other words, a decorator takes in a function, adds some functionality and returns it.

```
def make_pretty(func):
    def inner():
        #Prints a string and calls the argument\
        #(a function) from the outer function
        print("I got decorated")
        func()
    return inner

def ordinary():
    print("I am ordinary")
```

Figure 263: Closure function example

If you bound the result of calling `make_pretty(ordinary)` to a variable and then call that variable, you'll notice different results from just calling `ordinary()`:

```
ordinary() #I am ordinary
alternative = make_pretty(ordinary)
alternative() #prints two strings
#I got decorated
#I am ordinary
```

Figure 264: Calling a closure function

And here you have a decorator. In the assignment of `make_pretty(ordinary)`, the function `ordinary()` got decorated, that is, received additional functionality, and the returned function was given the name `alternative`.

28.2. Decorate Functions

Generally, a function is decorated using the syntax used in the example: *`variable = decorator_function(target_function)`*. Since this is a common construct in Python, there is dedicated syntax for the effect. We can use the at symbol, `@`, combined with the name of the decorator function and simply write that above the definition of the function to be decorated. All in all, please keep in mind this is simply syntactic sugar to implement decorators.

Using the same example but with the "proper" syntax this time, we'd have:

```
@make_pretty
def ordinary():
    print("I am ordinary")
```

Figure 265: Decorator for functions with no parameters

Which is equivalent to the previous example (Figure 264).

This first decorator example is simple and only works with functions that do not have any parameters. But what if our target functions have at least one parameter? Well, you can create dedicated decorators (e.g. a target function has two parameters, so you create a decorator that works with functions that receive two parameters) or you can create more general decorators that work with any function no matter how many parameters, be it positional or keyword arguments.

(Continued in the next page)

```python
def works_for_all(func):
    def inner(*args, **kwargs):
        print("I can decorate any function")
        return func(*args, **kwargs)
    return inner

@works_for_all
def ordinary():
    print("I am ordinary")
ordinary()
#I can decorate any function
#I am ordinary

@works_for_all
def mult_two(a,b):
    print(a*b)
mult_two(3,4)
#I can decorate any function
#12

@works_for_all
def sum_args(*args):
    print(sum(args))
sum_args(1,2,3)
#I can decorate any function
#6
```

Figure 266: Decorator for functions with multiple parameters

Note that decorators can also be used in classes, I simply preferred to show functions to explain the concept. For more information on this part, refer to the **@property** section (Chapter 27.11.2).

28.3. Using Multiple Decorators

You also have the option to chain multiple decorators, which means you can decorate the same function multiple times. For that, simply "stack" the decorators, each decorator on a separate line above the target function's definition:

```
@decoratora
@decoratorb
...
@decoratorn
def target_function
```

Figure 267: Decorator chaining syntax

And here's a concrete example of a function using two different decorators:

```
def star(func):
    def inner(*args, **kwargs):
        print("*" * 30)
        func(*args, **kwargs)
        print("*" * 30)
    return inner

def percent(func):
    def inner(*args, **kwargs):
        print("%" * 30)
        func(*args, **kwargs)
        print("%" * 30)
    return inner

@star
@percent
def printer(msg):
    print(msg)
printer("Hello")
```

Figure 268: Decorator chaining example

Which results in the output:

```
'''
**********
%%%%%%%%%%
Hello
%%%%%%%%%%
**********
'''
```

Figure 269: Figure 268's code output

It is also interesting to compare the decorator syntax to the “manual syntax”:

```
@star
@percent
def printer(msg):
    print(msg)
printer("Hello")

#Equivalent to
alternative = star(percent(printer))
```

Figure 270: Decorator and closure function comparison

Of course, the order in which the decorators are chained matters. If we reverse the order in this case, the output would result in this instead:

```
'''
%%%%%%%%%%
**********
Hello
**********
%%%%%%%%%%
'''
```

Figure 271: Importante of decorator chaining order

If you’re a bit confused about the order of events when calling the function after being decorated, it goes like this:

`printer("Hello")`, `print("*" * 30)`, `func(*args, **kwargs)`, `print("%" * 30)`, `func(*args, **kwargs)`, `print(msg)`, `print("%" * 30)`, `print("*" * 30)`.

29. Iterators

By now you have probably noticed that most container objects can be looped over using a `for` statement:

```
for element in [1, 2, 3]:
    print(element) #1'\n'2'\n'3
for element in (1, 2, 3):
    print(element) #1'\n'2'\n'3
for key in {'one':1, 'two':2}:
    print(key) #one'\n'two
for char in "123":
    print(char) #1'\n'2'\n'3
for line in open("myfile.txt"):
    print(line, end='')
    #this one actually raises an error
    #because there's no 'myfile.txt' file for this program
```

Figure 272: Loop through objects with for loops

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python.

Behind the scenes, the `for` statement calls `iter()` on the container object.

The function returns an iterator object that defines the method `__next__()` which accesses elements in the container one at a time. When there are no more elements, `__next__()` raises a `StopIteration` exception which tells the `for` loop to terminate.

You can call the `__next__()` method using the `next()` built-in function.

This example shows how it all works:

```
s = 'abc'
it = iter(s)
print(it)
#<str_iterator object at 0x04E5C750>
print(next(it)) #a
print(next(it)) #b
print(next(it)) #c
print(next(it))
#Traceback (most recent call last):
#    print(next(it))
#StopIteration
```

Figure 273: iter() + next() example

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an `__iter__()` method which returns an object with a `__next__()` method. If the class defines `__next__()`, then `__iter__()` can just return `self`:

(Continued in the next page)

```python
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

abc  = Reverse('abc')
iter(abc)
for char in abc:
    print(char) #c'\n'b'\n'a

rev = Reverse('spam')
iter(rev)
for char in rev:
    print(char) #m'\n'a'\n'p'\n's
```

Figure 274: Implementation of an __iter__() and __next__() methods

30. Generators

A *generator* is a function which returns a *generator iterator*.

It looks like a normal function except that it contains `yield` expressions for producing a series of values usable in a `for`-loop or that can be retrieved one at a time with the `next()` function.

It usually refers to a generator function but may refer to a generator iterator in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

An example shows that generators can be trivially easy to create:

```
1  def reverse(data):
2      for index in range(len(data)-1, -1, -1):
3          yield data[index]
4
5  for char in reverse('golf'):
6      print(char) #f'\n'l'\n'o'\n'g
```

Figure 275: Generator example

Anything that can be done with *generators* can also be done with *class*-based *iterators* as described in the **Iterators** chapter (Chapter 29). What makes generators so compact is that the `__iter__()` and `__next__()` methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls. This makes the function easier to write and much more clear than an approach using instance variables like `self.index` and `self.data`.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise `StopIteration`.

In combination, these features make it easy to create *iterators* with no more effort than writing a regular function.

30.1. Generator Iterator

An object created by a generator function.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending `try` statements).

When the generator iterator resumes, it picks-up where it left-off (in contrast to functions which start fresh on every invocation).

30.2. Generator Expressions

Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of brackets. These expressions are designed for situations where the generator is used right away by an enclosing function. Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

```
example1 = sum(i*i for i in range(10)) #sum of squares
print(example1) #285

xvec = [10, 20, 30]
yvec = [7, 5, 3]
example2 = sum(x*y for x,y in zip(xvec, yvec)) #dot product
print(example2) #260

from math import pi, sin
sine_table = {x: sin(x*pi/180) for x in range(0, 91)}
print(sine_table)
#creates a dictionary containing key-value pairs of angle-sin(angle) pairs
#{0: 0.0, 1:1: 0.01745240643728351, ... 90: 1.0}


data = 'golf'
example3 = list(data[i] for i in range(len(data)-1, -1, -1))
#list containg all characters of data, in reversed order
print(example3) #['f', 'l', 'o', 'g']
```

Figure 276: Generator expressions examples

31. File I/O

File I/O or File Input/Output is a term used for just that: file inputs and file outputs. This is used, for example, to read files from your computer and/or write information to another file.

31.1. File Input

Starting with the basics, to open (receive input from) a file in a Python program the `open()` function should be used.

`open()` returns a *file object*, and is most commonly used with these two parameters: `open(file, mode)`.

```
example1 = open('filename', 'w')
#this open the file filename, in write mode
```

Figure 277: open() common syntax

The first parameter must be the name of the file or the file's path, written as a string. It's worth pointing out that if a file path is being used in Windows, the backslash (used to separate the different components in a path) becomes a problem because it is also a special character used in Python strings to escape characters. Thus, there are two solutions for the problem: you can either double every backslash inside the string, so that it is interpreted normally, or you can simply prefix the string with an `r` to convert it into a raw string, that is, a string that doesn't allow characters to be escaped.

The second argument is a string containing one or two letters that specify how the file will used.

In this example, the file *`filename`* is going to be opened in `'w'`rite mode, which means the program would only write information in the file.

Here's a summary of all the available modes and the corresponding syntax:

Mode	Explanation
`'r'`	Open for reading.
`'w'`	Open for writing, truncating the file first.

`'x'`	Open for exclusive creation, failing if the file already exists.
`'a'`	Open for writing, appending to the end of the file if it exists.
`'b'`	Open in binary mode.
`'t'`	Open in text mode.
`'+'`	Open a disk file for updating (reading and writing).
`'U'`	Universal newlines mode (deprecated).

Table 6: Available modes for open()

Normally, files are opened in text mode, that means, you read and write strings from and to the file, which are encoded in a specific encoding. If *`encoding`* is not specified, the default is platform dependent.

If `'b'` is the chosen mode, then the file is opened in binary mode: the data is read and written in the form of bytes objects. This mode should be used for all files that don't contain text.

In text mode, the default when reading is to convert platform-specific line endings (`\n` on Unix, `\r\n` on Windows) to just `\n`. When writing in text mode, the default is to convert occurrences of `\n` back to platform-specific line endings. This behind-the-scenes modification to file data is fine for text files but will corrupt binary data like that in .jpg or .exe files. Be very careful of using binary mode when reading and writing such files.

Note that the full parameters list for `open()` is:

`open(`*`file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None`*`)`

However, the other parameters won't be tackled in this document[22].

[22] The `open()` section in the official Python documents is available at:
https://docs.python.org/3/library/functions.html#open

It is good practice to use the `with` keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point. Using `with` is also much shorter than writing equivalent `try`/`finally` blocks:

```
with open('workfile') as f:
    read_data = f.read()

print(f.closed) #True
```

Figure 278: Open files using the with statement

If you're not using the `with` keyword, then you should call `f.close()` to close the file and immediately free up any system resources used by it. If you don't explicitly close a file, Python's garbage collector will eventually destroy the object and close the open file for you, but the file may stay open for a while. Another risk is that different Python implementations will do this clean-up at different times.

After a file object is closed, either by a `with` statement or by calling `close()`, attempts to use the file object will automatically fail.

```
f.close()
f.read()
# raises the following error
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# ValueError: I/O operation on closed file
```

Figure 279: Using a file after calling close() on it

31.2. Methods of File Objects

For the following examples please assume that a file object `f` has been called beforehand.

31.2.1. fileobject.read()

To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode).

`.read()` reads and returns at most *`size`* characters from the stream as a single string. If *`size`* is negative or `None` (omitted), reads until `EOF` (End Of File exception).

If the end of the file has been reached, `f.read()` will return an empty string (`''`).

```
f.read()
#'file contents'\n'
f.read() #''; because the end of the file has been reached
```

Figure 280: Calling read() at the end of a file

31.2.2. fileobject.readline()

`f.readline(`*`size = -1`*`)` reads a single line from the file.

Read until newline or `EOF` and return a single string. If the stream is already at `EOF`, an empty string is returned.

If *`size`* is specified, at most *`size`* characters will be read.

A newline character (`\n`) is left at the end of the string and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous: if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by `'\n'`, a string containing only a single newline.

```
f.readline()
#'This is the first line of the file.\n'
f.readline()
#'Second line of the file\n'
f.readline()
#''; reached the end of the file
```

Figure 281: fileobject.readline() examples

If you want to read all the lines of a file in a list you can also use `list(f)` or `f.readlines()`.

31.2.3. fileobject.write()

`f.write(string)` writes the contents of *`string`* to the file, returning the number of characters written.

```
1   f.write('This is a test\n')
2   #15
```

Figure 282: fileobject.write() example

Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) – before writing them:

```
1   value = ('the answer', 42)
2   #convert the tuple to string
3   s = str(value)
4   f.write(s)
5   #18
```

Figure 283: Converting data for fileobject.write()

31.2.4. fileobject.tell()

`f.tell()` returns an integer with the file object's current position in the file, represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.

31.2.5. fileobject.seek()

To change the file object's position, use `f.seek(`*`offset, from_what`*`)`.

The position is computed from adding *`offset`* to a reference point; the reference point is selected by the *`from_what`* argument.

A *`from_what`* value of `0` measures from the beginning of the file; `1` uses the current file position; `2` uses the end of the file as the reference point.

`from_what` can be omitted and defaults to `0`, using the beginning of the file as the reference point.

```
f = open('workfile', 'rb+')
f.write(b'0123456789abcdef')
#16; 16 characters were written to the file
f.seek(5) #Go to the 6th byte in the file
f.read(1)
#b'5'; Read the character at the position we're at
f.seek(-3,2) #Go to 3rd last byte in the file (13th byte)
f.read(1)
#b'd'; Read the character at the position we're at
```

Figure 284: fileobject.seek() examples

In text files (those that do not include a `'b'` in `open()`'s mode), only seeks relative to the beginning of the file are allowed (the exception being seeking to the very file end with `f.seek(0, 2)`) and the only valid *`offset`* values are those returned from the `f.tell()`, or zero.

Any other *`offset`* value produces undefined behavior.

31.3. Saving Structured Data with JSON

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the `read()` method only returns strings, which will have to be passed to a function like `int()`, which takes a string like `'123'` and returns its numeric value, `123`. When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.

Rather than having users constantly writing and debugging code to save complicated data types to files, Python allows you to use the popular data interchange format called JSON[23] (JavaScript Object Notation).

The standard module **`json`** can take Python data hierarchies and convert them to string representations; this process is called *serializing*. Reconstructing the data from the string representation is called *deserializing*.

[23] The JSON format is commonly used by modern applications to allow for data exchange. Many programmers are already familiar with it, which makes it a good choice for interoperability.

Between serializing and deserializing, the string representing the object may have been stored in a file or sent over a network connection to some distant machine.

Given an object, you can view its JSON representation with simple code:

```
1   import json
2   json.dumps([1, 'simple', 'list'])
3   #'[1, "simple", "list"]'
```

Figure 285: Serialize a Python object using json.dumps()

There's a variant of the `dumps()` function, called `dump()`, that is used for the purpose, with the addition of serializing data directly to a text file. So, given a text file `f` opened for writing and an object `x`, we can serialize `x` and write it to `f` with:

```
1   json.dump(x, f)
```

Figure 286: Serialize an object and write it to a file

To decode the object again, if `f` has been opened for reading, we can call the `load()` function from the **`json`** module to deserialize the object:

```
1   x = json.load(f)
```

Figure 287: Deserialize an object from a file

32. Concurrent Execution (Threads)

Threads allow programs to run operations concurrently in the same process.

Normally, a program written with Python would only be executed line by line, but with threads you'll be able to execute different operations at the same time, which in turn reduces the time needed to complete your program.

32.1. Practical Example

Let's take a look at the program below:

```
1   import threading
2   import time
3
4   def product(a,b):
5       #first thing to be executed
6       print('{} \n'.format(a**b))
7       #while it sleeps the code outside the thread is executed
8       time.sleep(3)
9       #after sleep finishes the function prints this statement
10      print('Finished')
11
12  #create a thread called 't1', that targets the 'product' function,\
13  #and takes in two arguments
14  t1 = threading.Thread(target = product, name = 't1', args = (2,3))
15
16  #start the thread
17  t1.start()
18
19  #this statement will be executed while the thread is "sleep"ing
20  print('Concurrent execution!!!')
```

Figure 288: Concurrent execution using the threading module

Now that you've seen an example of concurrent execution let me explain what happened in this example. We begin by creating a function called `product`, which takes two arguments: `a` and `b`. By itself, `product` simply prints a string with the result of `a` to the power of `b`, sleeps (suspends the execution of the calling thread or the whole program if only one thread is being used) for three seconds and outputs `'Finished'` at the end. In the last line of the program there's a last `print` statement for another string.

However, you don't see any call for `product`, right? But you do see `t1`, which is a variable that has a thread assigned to it. This is where the threads come into play. We create a `Thread` object called `t1` by calling the `Thread()` function from the **threading** module. This object targets `product` and passes to it the arguments `(2,3)` to be used as input for the created function.

Though, what actually calls our function isn't `Thread()`, it's the `t1.start()` line that calls the function (more information ahead about this method in Chapter 32.3.4). Now that the thread has been started and `product()` called, the `print('{} \n'.format(a**b))` and `time.sleep(3)` lines are executed.

Now that the thread is sleeping, the rest of the code outside the thread is executed, which in this case is only the `print('Concurrent execution!!!')` line. After three seconds of being asleep, `product()` executes its last line: `print('Finished')`.

32.2. Theory

Now that you are familiar with a thread's behavior let's get a bit more into it.

Effectively, what happened in the example I've shown is we stored a function in a thread, started the thread to call the function, put the thread to sleep at one point and meanwhile executed the code outside of the thread while the first thread is was suspended. After resuming, the original thread executed the rest of the code in the function.

For this to happen, some steps were needed: importing the **`threading`** module, creating a `Thread` object (targeting a function in the example) which is then assigned to a variable, and lastly calling the `.start()` method on the thread.

32.3. threading.Thread()

We started by using the **`threading`**`.Thread()` constructor to create a `Thread` object. It can be called using the following arguments:

`threading``.Thread(`*`group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None`*`)`

Let's now explain each argument:

group should be None (it is reserved for future extension when a ThreadGroup class is implemented).

target is the callable object to be invoked by the *thread*.run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form Thread-N where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If not None, *daemon* explicitly sets whether the thread is daemonic. If None (the default), the daemonic property is inherited from the current thread.

The Thread object has handful of methods we can take advantage of, such as .start() and .run().

32.3.1. thread.is_alive()

thread.is_alive()

Return whether the *thread* is alive.

This method returns True just before the .run() method starts until just after the .run() method terminates. The module's function enumerate() returns a list of all alive threads.

32.3.2. thread.join()

threading.*thread*.join(*timeout=None*)

This method makes the program wait until *thread* terminates. This blocks the calling thread until the *thread* whose .join() method is called terminates – either normally, through an unhandled exception or until the optional *timeout* occurs.

When the *timeout* argument is present and not None, it should be a floating-point number specifying a timeout for the operation in seconds (or fractions thereof). As .join() always returns None, you must call .is_alive() after .join() to check if a timeout happened – if the thread

is still alive, then the `.join()` call timed out. When the *`timeout`* argument is not present or `None`, the operation will block until the thread terminates.

A thread can be `join`ed multiple times.

Note that `.join()` raises a `RuntimeError` if an attempt is made to join the current *`thread`* as that would cause a deadlock. It is also an error to `.join()` a thread before it has been started and attempts to do so raise the same exception.

32.3.3. thread.run()

`threading.`*`thread`*`.run()`

A method representing the *`thread`*'s activity.

You may override this method in a subclass. The standard `.run()` method invokes the callable object passed to the object's constructor as the *`target`* argument, if any, with sequential and keyword arguments taken from the *`args`* and *`kwargs`* arguments, respectively.

32.3.4. thread.start()

`threading.`*`thread`*`.start()`

Start the *`thread`*'s activity.

It must be called at most once per thread object. It arranges for the object's `.run()` method to be invoked in a separate thread of control.

This method will raise a `RuntimeError` if called more than once on the same thread object.

32.4. threading.enumerate()

`threading.``enumerate()`

Return a list of all `Thread` objects currently alive. The list includes daemonic threads, dummy thread objects created by `current_thread()`, and the main thread. It excludes terminated threads and threads that have not yet been started.

32.5. threading.current_thread()

`threading.``current_thread()`

Return the current `Thread` object, corresponding to the caller's thread of control. If the caller's thread of control was not created through the **`threading`** module, a dummy thread object with limited functionality is returned.

32.6. threading.main_thread()

`threading``.main_thread()`

Return the main `Thread` object.

In normal conditions, the main thread is the thread from which the Python interpreter was started.

32.7. Daemon Threads

Daemon threads are abruptly stopped at shutdown. Their resources (such as open files, database transactions, etc.) may not be released properly. If you want your threads to stop gracefully, make them non-daemonic and use a suitable signaling mechanism.

A thread can be made daemonic if when creating a thread using **`threading`**`.Thread()` you don't leave the argument *`daemon`* as `None`, that is, assign *`daemon`* `= True`.

32.7.1. thread.isdaemon()

`threading``.`*`thread`*`.isdaemon()`

You can use the `.isdaemon()` method to test if *`thread`* is a daemon thread or not. This returns a Boolean value of `True` or `False`.

33. Fetch Internet Resources with urllib

Before going further, for this chapter I based its content around Michael Foord's article "HOWTO Fetch Internet Resources Using The urllib Package" which is available at https://docs.python.org/3.1/howto/urllib2.html. It is a very well written article that gives you a "tour" of what you can do with the **`urllib`** package and even explains some of the underlying concepts regarding HTTP requests.

33.1. Introduction

Throughout this chapter of the Manual we are going to focus mainly on the **`urllib.request`** module because it is the one we need the most to make just that, requests. However, the other modules are still going to be tackled because they are also important and include helpful tools in creating these requests.

So, what is **`urllib.request`**? It is one of the modules included in the **`urllib`** package, and it is used for fetching URLs (Uniform Resource Locators). It offers a simple way of doing this: the `urlopen()` function. Using this, you can fetch most of the URLs you need, while also offering some other tools to deal with common situations on the Internet, such as basic authentication or cookies. These are provided by objects called handlers and openers.

In short, for simple and objective situations `urlopen()` will be simple to use and sufficient to do the work. However, as soon as you encounter errors or non-trivial cases when opening HTTP (HyperText Transfer Protocol) URLs, you will need some understanding of the HTTP protocol. As you can expect, this is not my objective here, so I will leave you a source where you can find some of your answers regarding HTTP: https://tools.ietf.org/html/rfc2616.html.

33.2. Fetching URLs

The simplest way to fetch a URL with **`urllib.request`** is using `urlopen()`. Here is an example of the syntax for this:

```
1  import urllib.request
2  response = urllib.request.urlopen('http://python.org/')
3  html = response.read()
```

Figure 289: urlopen() example

HTTP is based on requests and responses, that is, the client makes requests and servers send responses. `urllib.request` mirrors this behavior with the `Request` object which represents the HTTP request you're making. In its simplest form, you create a `Request` object that specifies the URL you're trying to fetch. Calling `urlopen()` with this object returns a response object for the URL requested. This response is a file-like object, which means you can call methods on it, such as `.read()`:

```
1  import urllib.request
2
3  req = urllib.request.Request('http://python.org/')
4  response = urllib.request.urlopen(req)
5  page_file = response.read()
```

Figure 290: Calling file object methods on Request objects

Fortunately, `urllib.request` makes use of the same `Request` interface to handle all URL schemes. This means you can easily request URLs that use other protocols, such as FTP.

```
1  import urllib.request
2  req = urllib.request.Request('ftp://example.com/')
```

Figure 291: Requesting non-HTTP URL schemes

For the HTTP scheme, there are two extra things that `Request` objects allow: (1) you can pass data to be sent to the server and (2) you can pass extra information (metadata) about the data or about the request itself to the server (this information is sent as HTTP headers).

33.3. Data

In some cases, you may want to send data to a URL (often the URL will refer to a Common Gateway Interface, CGI, or other web application). With HTTP, this is often done with a POST request.

If you're wondering what's a POST request, this is often what your browser does when you submit a HTML form filled online. Not all POST requests have to come from forms though, you can use a POST to transmit data to your own application. In the common case of HTML forms, the data needs to be encoded in a standard way and then passed to the `Request` object as the *`data`*

argument. The encoding is done using `urlencode()`, a function from another module included in the **`urllib`** package: **`urllib.parse`**.

```
1  import urllib.parse
2  import urllib.request
3
4  url = 'http://www.someserver.com/cgi-bin/register.cgi'
5  values = {'name':'José Costa', 'location':'Portugal', 'language':'Python'}
6
7  data = urllib.parse.urlencode(values)
8  req = urllib.request.Request(url, data)
9  response = urllib.request.urlopen(req)
10 page_file = response.read()
```

Figure 292: urlencode() example

Note that other encodings may be required (e.g. for file upload from HTML forms visit the link: https://www.w3.org/TR/REC-html40/interact/forms.html#h-17.13).

If you do not pass the ***`data`*** argument, **`urllib`** uses a GET request. One way in which GET and POST requests differ is that the latter often have "side-effects", that is, they change the state of the system in some way. Though the HTTP standard makes it clear that POSTs are intended to always cause side-effects, GET requests are never intended to cause side-effects. However, nothing prevents a GET request from having side-effects, nor a POST request from having no side-effects. Data can also be passed in an HTTP GET request by encoding it in the URL itself. This is done as such:

(Continued in the next page)

```
import urllib.parse
import urllib.request

data = {}
data['name'] = 'José Costa'
data['location'] = 'Portugal'
data['language'] = 'Python'
url_values = urllib.parse.urlencode(data)
print(url_values)
#name=Jos%C3%A9+Costa&location=Portugal&language=Python
url = 'http://www.example.com/example.cgi'
full_url = url + '?' + url_values
data = urllib.request.open(full_url)
```

Figure 293: Passing data to a GET request

Notice that the full URL is created by adding a `?` to the URL, followed by the encoded values.

33.4. Headers

Some websites do not like to be browsed by programs or send different versions to different browsers. By default, **`urllib`** identifies itself as `Python-urllib/x.y` (where `x` and `y` are the major and minor version numbers of the Python release, e.g. `Python-urllib/3.6`), which may confuse the site, or just don't work at all. The way a browser identifies itself is through the `User-Agent` header. When you create a `Request` object, you can pass a dictionary of headers in.

Take a look at the following example, which is the same as the one from the previous section but identifies itself as a version of Internet Explorer.

(Continued in the next page)

```
import urllib.request
import urllib.parse

url = 'http://www.someserver.com/cgi-bin/register.cgi'
user_agent = 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'
values = {'name' : 'José Costa',
          'location' : 'Portugal',
          'language' : 'Python' }
headers = {'User-Agent' : user_agent}

data = urllib.parse.urlencode(values)
req = urllib.request.Request(url, data, headers)
response = urllib.request.urlopen(req)
the_page = response.read()
```

Figure 294: Passing headers to a Request object

The response also has two useful methods: `.geturl()` and `.info()`, but those will be tackled further ahead (Chapter 33.10).

33.5. Handling Exceptions

`urlopen()` raises an `URLError` when it cannot handle a response (though as usual with Python APIs, built-in exceptions such as `ValueError`, `TypeError`, etc. can also be raised).

`HTTPError` is the subclass of `URLError` raised in the specific case of HTTP URLs. These exception classes are exported from the **`urllib.error`** module.

33.6. URLError

In a large number of cases, `URLError` is raised because there is no network connection (there is no way to connect to the specified server) or the specified server doesn't exist. However, the exception raised will have a `reason` attribute, which returns an error code and a text error message.

Here's an example:

```
import urllib.request, urllib.error
req = urllib.request.Request('http://www.pretend_server.org')
try:
    urllib.request.urlopen(req)
except urllib.error.URLError as e:
    print(e.reason)
    #[Errno 11001] getaddrinfo failed
```

Figure 295: URLError.reason example

33.7. HTTPError

Every HTTP response from the server contains a numeric status code. In some cases, this status code will indicate the server is unable to fulfil the request. Though, in these cases, the default handlers will handle some of these responses (e.g. if the response is a "redirection" that requests the client to fetch a document from a different URL, **`urllib`** will handle that for you). For the rest of the cases, `urlopen()` will raise an `HTTPError`. Typical errors include 404 (Page Not Found), 403 (Request Forbidden) and 401 (Authentication Required).

`HTTPError` instances when raised will have an integer `code` attribute which corresponds to the error sent by the server.

To read the references for all HTTP error codes read section 10[24] of the already mentioned RFC 2616.

33.8. Error Codes

Because the default handlers handle redirects, that is, codes in the 300s range, and codes in the 100-299 range indicate success, you will usually only error codes in the 400-599 range.

`http.server``.BaseHTTPRequestHandler.responses` is a useful dictionary that contains all the response codes used by RFC 2016. For each code, the dictionary uses the following format: `{`*`code`*`: (`*`shortMessage`*`, `*`longMessage`*`)}`, with *`code`* being the error code itself, *`shortMessage`* one or two words to represent the error and *`longMessage`* a small explanation of the error.

[24] Available at: https://tools.ietf.org/html/rfc2616.html#section-10

For the sake of space management, these error codes contained in the dictionary will be available in the **Notes and Specific Information** chapter (Chapter 35.10).

When an error is raised, the server responds by returning an HTTP error code and an error page. You can use the `HTTPError` instance as a response on the page returned. This means that beyond the `code` attribute, it also has a `.read()`, `.geturl()` and `.info()` methods, as returned by the **`urllib.response`** module.

```
1  import urllib.request, urllib.error
2  req = urllib.request.Request('http://www.python.org/fish.html')
3  try:
4      urllib.request.urlopen(req)
5  except urllib.error.HTTPError as e:
6      print(e.code) #404
7      print(e.read())
8      #b'<!doctype html>\n<!--[if lt IE 7]>   <html class="no-js\
9      #ie6 lt-ie7 lt-ie8 lt-ie9">   <![endif]-->\n<!--[if IE 7]>
10     #(...)
```

Figure 296: urllib.error.HTTPError example

33.9. Preparations for HTTPError and URLError

There are two basic approaches to prepare for `HTTPError` and/or `URLError` errors.

In the first approach, we use a `try`/`except`/`else` clause, with separate `except`s for each error type, and a final `else` statement that is only executed if both `except` clauses aren't. Due note that the `except HTTPError` statement must come first otherwise `except URLError` will also catch an `HTTPError`.

In the second approach, we use the same `try`/`except`/`else`, with the difference that this time we only except `URLError`, and then we use a conditional statement to test if it is an `URLError` or an `HTTPError`. Still, we keep the `else` statement that is executed if the `except` statement is not.

```
from urllib.request import Request, urlopen
from urllib.error import URLError, HTTPError
req = Request('http://www.example.com')
try:
    response = urlopen(req)
#except clause for HTTPError
except HTTPError as e:
    print('The server couldn\'t fulfill the request.')
    print('Error code: ', e.code)
#except clause for URLError
except URLError as e:
    print('We failed to reach a server.')
    print('Reason: ', e.reason)
#Optional else clause to be executed only when the \
#except clauses do not execute
else:
    pass
```

Figure 297: First approach for HTTPError and URLError

```
from urllib.request import Request, urlopen
from urllib.error import  URLError
req = Request('http://www.example.com')
try:
    response = urlopen(req)
#Create a single except clause for URLError
except URLError as e:
    #If the error has a 'reason' attribute then it is an \
    #URLError
    if hasattr(e, 'reason'):
        print('We failed to reach a server.')
        print('Reason: ', e.reason)
    #Else if the error has a 'code' attribute then it is \
    #an HTTPError
    elif hasattr(e, 'code'):
        print('The server couldn\'t fulfill the request.')
        print('Error code: ', e.code)
#This else clause will only execute if the except clause \
#does not execute
else:
    pass
```

Figure 298: Second approach for HTTPError and URLError

33.10. .info() and .geturl()

As said in the beginning, the response returned by `urlopen()` (or the `HTTPError` instance) has two useful methods: `.info()` and `.geturl()`, both defined in the **`urllib.response`** module.

Typical headers include `Content-length`, `Content-type` and so on. For a more detailed listing of HTTP headers with brief explanations of their meaning and use you read the document Quick Reference to HTTP Headers[25].

33.10.1. .info()

`.info()` returns a dictionary-like object that describes the page fetched, particularly the headers sent by the server. It is currently an **`http.client.`**`HTTPMessage` instance.

33.10.2. .geturl()

`.geturl()` returns the real URL of the page fetched. This is useful in cases that `urlopen()` (or the opener object used) may have followed a redirect. Thus, the URL of the page fetched may not be the same as the URL requested.

33.11. Openers and Handlers

When you fetch a URL you're using an opener, that is, an instance of **`urllib.request.`**`OpenerDirector`. So far, I've shown examples using only the default opener, `urlopen()`, but you can create custom openers. These custom openers use handlers that know how to open URLs for a particular URL scheme (http, ftp, etc.) or how to handle an aspect of URL opening, such as HTTP redirections or HTTP cookies. As an example, one of the motivations to create custom openers can be to fetch URLs with specific handlers installed.

To create an opener, we start by instantiating an `OpenerDirector` object, to then call `OpenerDirector.add_handler(`*`handler_instance`*`)` method repeatedly. Alternatively, you can use **`urllib.request.`**`build_opener()`, which is a convenience function for creating opener objects with a single function call. This second option adds several handlers by default while also providing a quick way to add more and/or override the default handlers.

[25] Available at: http://jkorpela.fi/http.html

Other sorts of handlers you might want to create can handle proxies, authentication, and other common but slightly specialized situations.

`urllib.request.install_opener(opener_instance)` can be used to make a custom opener object the (global) default opener. This means that calls to `urlopen()` will use the opener you have installed.

Lastly, `opener` objects haven an `.open()` method, which can be called directly to fetch URLs in the same way as the `urlopen()` function: there's no need to call `install_opener()`, only as a convenience.

33.12. Basic Authentication

When authentication is required the server sends a header, as well as a 401 error code, requesting authentication. This specifies the authentication scheme and a *realm*. The header looks like this: `Www-authenticate: SCHEME realm="REALM"`. Here is a practical example: `Www-authenticate: Basic realm="cPanel Users"`.

After this, the client should then retry the request with the appropriate name and password for the realm included as a header in the request. This is what is called "basic authentication". In order to simplify this process we can create an instance of `urllib.request.HTTPBasicAuthHandler` and an opener to use this handler.

`HTTPBasicAuthHandler` uses an object called a password manager to handle the mapping of realm and URLs to usernames and passwords, respectively. If you know what the realm is (from the authentication header sent by the server) then you can use a `urllib.request.HTTPPasswordMgr` object. Normally, one doesn't care what the realm, so it is convenient to use `urllib.request.HTTPPasswordMgrWithDefaultRealm`. This allows you to specify a default username and password for a URL. This will be supplied in the absence of you providing an alternative combination for a specific realm. We indicate this by providing `None` as the realm argument to the `.add_password()` method.

The top-level URL is the first URL that requires authentication. URLs "deeper" than the URL you pass to `.add_password()` will also match.

```python
import urllib.request as libreq

#Create a password manager
passwrd_mgr = libreq.HTTPPasswordMgrWithDefaultRealm()

#Add the username and password
#If the realm was known, we could use it instead of None
top_level_url = "http://example.com./foo/"
#.add_password(realm, uri, user, password)
passwrd_mgr.add_password(None, top_level_url, username, password)
handler = libreq.HTTPBasicAuthHandler(passwrd_mgr)

#Create the opener
opener = libreq.build_opener(handler)

#Use the opener to fetch a URL
opener.open(example_url)

#Install the opener
#From here on all urllib.request calls use this opener
libreq.install_opener(opener)
```

Figure 299: HTTPPasswordMgr example

Note that in the above example we only supplied our `HTTPBasicAuthHandler` to `.build_opener()`. By default, openers have the handlers for normal situations: `ProxyHandler`, `UnknownHandler`, `HTTPHandler`, `HTTPDefaultErrorHandler`, `HTTPRedirectHandler`, `FTPHandler`, `FileHandler`, `HTTPErrorProcessor`.

Still regarding the previous example, `top_level_url` is actually either a full URL, including the `http:` scheme component and the hostname and optionally even the port number. Examples of these full URLs can be `http://example.com` or an *authority*, such as the hostname, optionally including the port number (e.g. `example.com` or `example.com:8080` (the second one includes a port number)). The authority, if present, must not contain the "userinfo" component (e.g. `joe@password:example.com` is not correct).

33.13. Proxies

`urllib` will detect automatically your proxy settings and use those. This is through the **`urllib.request`**`.ProxyHandler` which is part of the normal handler chain. Normally that's a

good thing, but there are occasions when it may not be helpful. One way to do this is to setup our own `ProxyHandler`, with no proxies defined. This is done using similar steps to the ones involved in a basic authentication handler setup:

```
import urllib.request as libreq
#Create a ProxyHandler object using a dictionary mapping \
#protocol names to URLs of proxies
proxy_support = libreq.ProxyHandler({})
#Create an OpenerDirector instance
opener = libreq.build_opener(proxy_support)
#Lastly, install the opener, that is, the proxy
libreq.install_opener(opener)
```

Figure 300: ProxyHandler example

34. Web Scraping

Maybe after reading the **File I/O** chapter (Chapter 31) a thought occurred to you "What if I could scrape the latest news from site X and then write them to a .txt file so I can read them at the end of the day?". Well, I also had that thought once and so I set myself to learn how to scrape the web using Python and a couple of Python external libraries (modules that I had to install separately, but we'll get to that in a moment).

Well, thankfully I found this video tutorial created by Corey Schafer called "Python Tutorial: Web Scraping with BeautifulSoup and Requests", which you can watch at https://www.youtube.com/watch?v=ng2o98k983k&ab_channel=CoreySchafer.

34.1. Introduction

For starters, we are going to need three things: basic knowledge of HTML (here I am going to assume you're good to go), and two Python external libraries: `BeautifulSoup4` and `Requests`. When I say external libraries, I mean these don't come with Python when you install it, you have to install them separately using *pip*, the preferred installer program. I am not going into details regarding *pip* either, but you can learn more about it in this link: https://docs.python.org/3/installing/index.html.

Moving on to the aforementioned external libraries: `BeautifulSoup4` and `Requests`. The first library, **BeautifulSoup4**[26], think of it as a formatting plate. You "grab" the source code from your target website and then parse into a `BeautifulSoup` object, so that it can "read" your HTML.

On the other hand, **Requests**[27], is what mediates your Python script and the web so that you can fetch the needed source code for Beautiful Soup.

Before going further, maybe you're thinking it's strange for me to not use `urllib` to fetch internet resources. Well, the reason is that `Requests` is much simpler and more straightforward to use for HTTP requests. However, do not think it was pointless for me to include the previous chapter (Chapter 33) in this document, because it includes vital information regarding this area,

[26] To read more about BeautifulSoup4 access the link https://pypi.python.org/pypi/beautifulsoup4
[27] To read more about Requests access the link http://docs.python-requests.org/en/master/

and, after all, **`urllib`** is a built-in module while **`Requests`** isn't, which means there may be a time when you don't have the latter available. But if I do have it available I will use it instead of the built-in option.

34.2. Fetch Websites' Source Code with Requests

Let's start, obviously from the beginning: fetching the source code for the website we want. For this example we are going to target Science Magazine's website (http://www.sciencemag.org/) and scrape the titles and URLs of its featured articles (the ones that appear on the banner).

First off, we import **`Requests`** in our script. But wait, do we really need the entire module? No. We are only going to make one web request, a GET request to be precise. So, we'll start our script by importing the function and calling it with our target website:

```
1  from requests import get
2  website = "http://www.sciencemag.org/"
3  source = get(website)
```

Figure 301: requests.get()

However, we only want a text version of the source code, so we call the `text` attribute of what we just fetched and assign it to the `source` variable:

```
1  from requests import get
2  website = "http://www.sciencemag.org/"
3  source = get(website).text
```

Figure 302: Obtain the website's source code text version

And for the **Requests** side of things that's everything! We have a text version of the website's source code so now we're ready to just work with it and scrape whatever information we want.

34.3. Digging through the HTML

Now we import **`BeautifulSoup4`**, particularly, the `BeautifulSoup` class.

The first thing we do with `BeautifulSoup` is to create an instance of the class, parsing to it the `source` variable we just created. However, we can't just parse the source code like that, we need to indicate an HTML parser for the class. We are going to use `lxml`.

```
1  from requests import get
2  from bs4 import BeautifulSoup
3
4  website = "http://www.sciencemag.org/"
5  source = get(website).text
6
7  soup = BeautifulSoup(source, 'lxml')
```

Figure 303: Parsing the HTML to a BeautifulSoup object

Right now, we have the source code of the website and we have it completely parsed into a `BeautifulSoup` object, perfect. This means we have almost finished what we aimed to accomplish. Our next step is to dig through the HTML and find where our information, that is, the article's titles and URLs, is. Of course, for this part we are going to use the browser's built-in Inspect tool.

After right-clicking the first article's title to Inspect it, the tool guides opens almost directly in our point of interets. At the moment, we are located in a nested `a` tag, inside an `h2` tag inside a `div` tag of the class `"hero__content"`. Luckily, this is exactly where we want to be.

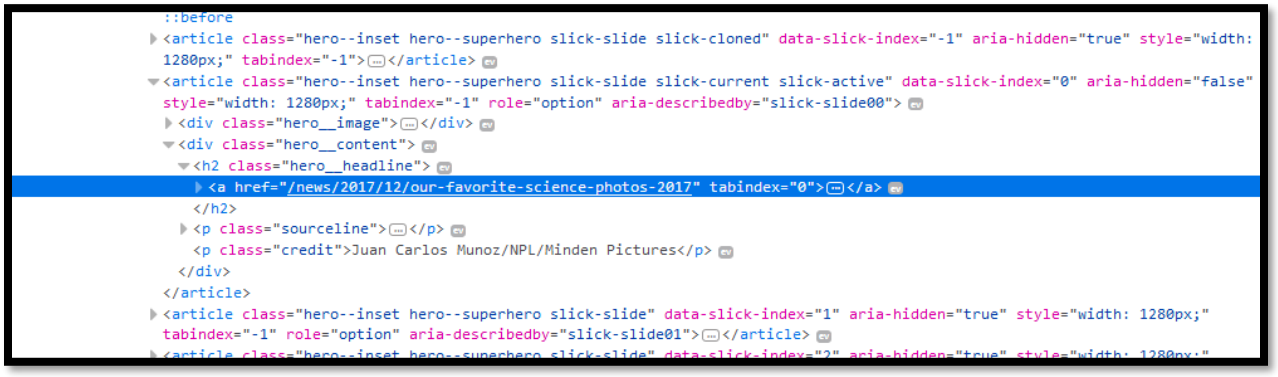


Figure 304: Snippet of the HTML source code using the Inspect tool

If you look closely, the highlighted `a` tag is condensed. If you expand it, you'll find we already have both things we need: the article's title and its URL.

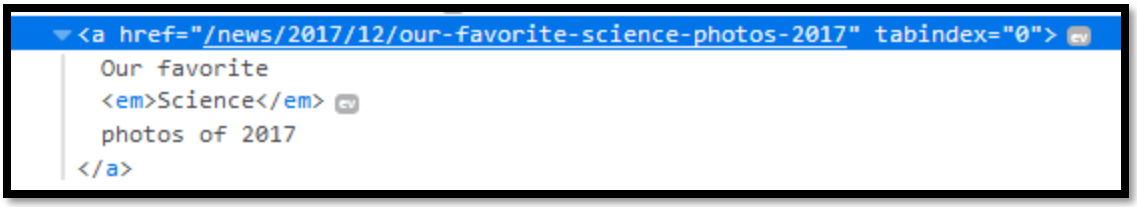


Figure 305: Relevant snippet of the source HTML (condensed)

The article's title was condensed inside the `a` tag, while the URL, albeit not complete, it's also present in the tag, as the `href`. Fortunately, this is one of those cases where to obtain the full URL you just need to prefix the website's URL to what you found. This means the full URL for this particular article is http://www.sciencemag.org/news/2017/12/our-favorite-science-photos-2017.

34.4. BeautifulSoup_object.find()

We've done the hard part, locate the information in the HTML. Now we just need to write the code to do it for us. Before switching back to the script let me just show you the full HTML code relevant to scrape this article:

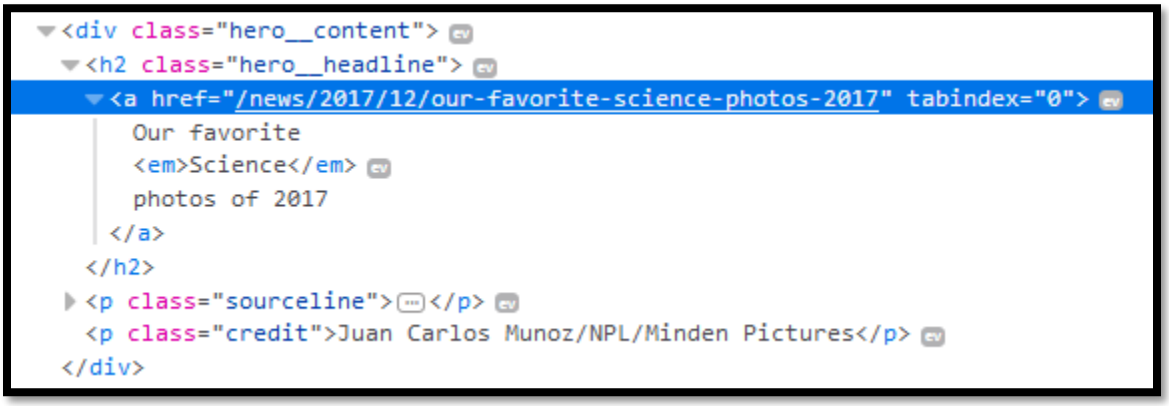


Figure 306: Relevant snippet of the source HTML (expanded)

The first thing we need to do in the script is to call the `.find()` method of the `soup` variable to find the `div` we are looking for. We need to use the `div` instead of the `a` because we are also going to parse the `div`'s class, so the method knows what to look for (rather, the first instance of this `div`).

```
from requests import get
from bs4 import BeautifulSoup

website = "http://www.sciencemag.org/"
source = get(website).text

soup = BeautifulSoup(source, 'lxml')

#.div(tagToLookFor, class_ = classOfTheTag)
article = soup.find('div', class_ = "hero__content")
```

Figure 307: BeautifulSoup_object.find()

With this, `article` will be the first instance of `div` with the class `"hero__content"` of the source code we scraped. Though, as we've seen before we actually want the code under the `a` tag. What we'll do is navigate the nested tags inside the `div` using dot notation, with each dot denoting a nested line.

```
from requests import get
from bs4 import BeautifulSoup

website = "http://www.sciencemag.org/"
source = get(website).text

soup = BeautifulSoup(source, 'lxml')

#.div(tagToLookFor, class_ = classOfTheTag)
article = soup.find('div', class_ = "hero__content")
#from the div we pass to the h2 then pass to the a
link = article.h2.a
print(link)
'''
<a href="/news/2017/12/our-favorite-science-photos-2017">
        Our favorite <em>Science</em> photos of 2017                </a>
'''
```

Figure 308: Finding the relevant HTML code using find()

We did get the `a` tag, but we got the whole tag. To get the `link`, which is the `href` property of the tag, we'll have to access it using dictionary notation, like this:

```
from requests import get
from bs4 import BeautifulSoup

website = "http://www.sciencemag.org/"
source = get(website).text

soup = BeautifulSoup(source, 'lxml')

#.div(tagToLookFor, class_ = classOfTheTag)
article = soup.find('div', class_ = "hero__content")
#from the div we pass to the h2 then pass to the a
link = article.h2.a["href"]
print(link)
#/news/2017/12/our-favorite-science-photos-2017
```

Figure 309: Extracting just the link from the HTML

Much better, but remember, this isn't the whole URL, we still need to prefix it with the website's URL. To do that, we'll simply add two strings: the `website` and `link` variable.

```
from requests import get
from bs4 import BeautifulSoup

website = "http://www.sciencemag.org/"
source = get(website).text

soup = BeautifulSoup(source, 'lxml')

#.div(tagToLookFor, class_ = classOfTheTag)
article = soup.find('div', class_ = "hero__content")
#from the div we pass to the h2 then pass to the a
link = website + article.h2.a["href"]
print(link)
#http://www.sciencemag.org//news/2017/12/our-favorite-science-photos-2017
```

Figure 310: Obtaining the article's full URL

And there we have it. We finally got the URL in our hands! To scrape the actual title, we'll simply call the `text` attribute on the tag. However, did you notice that in Figure 308 when it printed the whole tag there was some whitespace in the title? I guess it's better we also call the `str.strip()` string method on it so we make sure we only get a clean string for the title.

```
from requests import get
from bs4 import BeautifulSoup

website = "http://www.sciencemag.org/"
source = get(website).text

soup = BeautifulSoup(source, 'lxml')

#.div(tagToLookFor, class_ = classOfTheTag)
article = soup.find('div', class_ = "hero__content")
#from the div we pass to the h2 then pass to the a
link = website + article.h2.a["href"]
print(link)
#http://www.sciencemag.org//news/2017/12/our-favorite-science-photos-2017
title = article.h2.a.text.strip()
print(title)
#Our favorite Science photos of 2017
```

Figure 311: Extracting the article's title

There we go! Now we have successfully scraped the title and URL of an article from Science Magazine's website using Python. Why don't we wrap it up with a single nice `print` statement?

(Continued in the next page)

```
from requests import get
from bs4 import BeautifulSoup

website = "http://www.sciencemag.org/"
source = get(website).text

soup = BeautifulSoup(source, 'lxml')

#.div(tagToLookFor, class_ = classOfTheTag)
article = soup.find('div', class_ = "hero__content")
#from the div we pass to the h2 then pass to the a
link = website + article.h2.a["href"]
title = article.h2.a.text.strip()
print('The first featured article on Science Magazine is "{}" which you \
can read at: {}'.format(title, link))
#The first featured article on Science Magazine is "Our favorite Science \
#photos of 2017" which you can read at: \
#http://www.sciencemag.org//news/2017/12/our-favorite-science-photos-2017
```

Figure 312: Creating the final string with both the title and URL of the article

Perfect, our objective is complete. Fortunately, it was easy to find the relevant HTML code but, anyway, we did something pretty cool in less than 20 lines of Python code.

34.5. BeautifulSoup_object.find_all()

Now, let's say you read a bit of the code after the `div` we used and noticed for each article in the website's banner the `div`s are all from the same class? Does it mean there's an easy way to scrape this information from each of these articles? Yes, yes there is. The `BeautifulSoup` object we created also has a `.find_all()` method, which instead of finding just the first instance of the tag you parse, it returns every instance of the tag. This means we could use this and very easily scrape the information for all the articles! Let's give it a shot by simply switching the methods and nesting all the code inside a `for` loop:

```
website = "http://www.sciencemag.org/"
source = get(website).text

soup = BeautifulSoup(source, 'lxml')

#we'll loop through each div this method found
articles = soup.find_all('div', class_ = "hero__content")
for article in articles:
    link = website + article.h2.a["href"]
    title = article.h2.a.text.strip()
    print('The first featured article on Science Magazine is "{}" which you \
can read at: {}'.format(title, link))
    #a line for space
    print()
```

Figure 313: Extracting all the article's titles and URLs

```
#The first featured article on Science Magazine is "Our favorite Science photos of 2017" \
#which you can read at: http://www.sciencemag.org//news/2017/12/our-favorite-science-photos-2017

#The first featured article on Science Magazine is "A detailed map of North and South \
#America's plant diversity" which you can read at: \
#http://www.sciencemag.org/http://science.sciencemag.org/content/358/6370/1614

#The first featured article on Science Magazine is "What can machine learning do? \
#Workforce implications" which you can read at: \
#http://www.sciencemag.org/http://science.sciencemag.org/content/358/6370/1530

#The first featured article on Science Magazine is "Science's 2017 Breakthrough of \
#the Year: Colliding neutron stars" which you can read at: \
#http://www.sciencemag.org//news/2017/12/science-s-2017-breakthrough-year-colliding-neutron-stars

#The first featured article on Science Magazine is "The lower your social class, \
#the 'wiser' you are, suggests new study" which you can read at: \
#http://www.sciencemag.org//news/2017/12/lower-your-social-class-wiser-you-are-suggests-new-study

#The first featured article on Science Magazine is "Why Americans support refusing \
#service" which you can read at: \
#http://www.sciencemag.org/http://advances.sciencemag.org/content/3/12/eaao5834

#The first featured article on Science Magazine is "Can a multibillion-dollar \
#biotech prove its RNA drugs are safe for a rare disease?" which you can read at: \
#http://www.sciencemag.org//news/2017/12/can-multibillion-dollar-biotech-prove-its-rna-drugs-are-safe-rare-disease
```

Figure 314: Figure 313's code output

We did it! Just like that we scraped all the articles. What have we learned from this? Generally, we can take advantage of this structure in websites to loop through it and extract the contents for each item with little effort.

Of course, as I've acknowledged, this was a very simple example, just by Inspecting the page we got directly the relevant HTML code, but still, you can generalize this process for other websites. Most of the time it will be a matter of adjusting your Inspecting to search for the code you need.

All in all, now you have a basic grasp of how to scrape information from a website using Python.

35. Notes and Specific Information

This final chapter aggregates mostly tables for conversions, such as the JSON conversion table, and specifications regarding certain subjects, such as the built-in exceptions in Python.

35.1. %-formatting Format Specifiers

The following table contains the different format specifiers to format strings using the percentage symbol substitution.

Conversion	Meaning
`%d`	Integer decimal.
`%i`	Integer decimal.
`%o`	Octal value.
`%u`	Obsolete; Identical to `%d`.
`%x`	Hexadecimal (lowercase).
`%X`	Hexadecimal (uppercase).
`%e`	Floating point exponential format (lowercase).
`%E`	Floating point exponential format (uppercase).
`%f`	Floating point decimal format.
`%F`	Floating point decimal format.
`%g`	Floating-point format; Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.
`%G`	Floating-point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise[28].

[28] The alternate form causes the result to always contain a decimal point, and trailing zeroes are not removed as they would otherwise be. The precision determines the number of significant digits before and after the decimal point and defaults to 6.

`%c`	Single character (accepts integer or single character string).
`%r`	String (converts any Python object using `repr()`).
`%s`	String (converts any Python object using `str()`).
`%a`	String (converts any Python object using `ascii()`).
`%`	No argument is converted, results in a `%` character in the result.

Table 7: %-formatting format specifiers

35.2. Division Differences (Python 2 vs. Python 3)

Between Python 2 and Python 3 there is a big difference in the way divisions are used and how the programs execute them:

```
Python 2                                Python 3

a = 5 / 2                               a = 5 / 2
print a '''outputs 2'''                 print(a) '''outputs 2.5'''

a = 5.0 / 2                             a = 5.0 / 2.0
print a '''outputs 2.5'''               print(a) '''outputs 2.5'''
```

Figure 315: Python 2 vs. Python 3 division

In Python 2 when both numbers are integers the program will execute the division and output the highest integer lower than the quotient (floor division); this means either or both the dividend and the divisor need to be written as a float to obtain a floating-point number.

In Python 3 it doesn't matter if integers or floats are used in the division, the program will output the correct result without rounding it up or down.

To force the floor division in Python 3, `//` should be used instead of `/`, which will output the result rounded down as expected in floor division:

```
a = 5 // 2
print(a) #outputs 2
```

Figure 316: Floor division in Python 3

35.3. JSON Conversion Table

The following table contains the corresponding values for Python-JSON and JSON-Python objects conversion.

Python	JSON
`dict`	`object`
`list`, `tuple`	`array`
`str`	`string`
`int`, `float`, `int`- & `float`-derived `Enum` classes	`number`
`True`	`true`
`False`	`false`
`None`	`null`

Table 8: JSON conversion table

35.4. Python Built-In Exceptions

The following table contains the built-in exceptions available in Python.

Exception	Cause of error
`AssertionError`	Raised when an `assert` statement fails.
`AttributeError`	Raised when attribute assignment or reference fails.
`EOFError`	Raised when the `input()` functions hits end-of-file condition.
`FloatingPointError`	Raised when a floating point operation fails.
`GeneratorExit`	Raise when a generator's `close()` method is called.
`ImportError`	Raised when the imported module is not found.
`IndexError`	Raised when the index of a sequence is out of range.
`KeyError`	Raised when a key is not found in a dictionary.

`KeyboardInterrupt`	Raised when the user hits interrupt key (Ctrl+C or delete).
`MemoryError`	Raised when an operation runs out of memory.
`NameError`	Raised when a variable is not found in the local or the global scope.
`NotImplementedError`	Raised by abstract methods.
`OSError`	Raised when a system operation causes a system related error.
`OverflowError`	Raised when the result of an arithmetic operation is too large to be represented.
`ReferenceError`	Raised when a weak reference proxy is used to access a garbage collected referent.
`RuntimeError`	Raised when an error does not fall under any other category.
`StopIteration`	Raised by the `next()` function to indicate that there is no more items to be returned by the iterator.
`SyntaxError`	Raised by the parser when a syntax error is encountered.
`IndentationError`	Raised when there is incorrect indentation.
`TabError`	Raised when the indentation consists of inconsistent tabs and spaces.
`SystemError`	Raised when the interpreter detects an internal error.
`SystemExit`	Raised by the `sys.exit()` function.
`TypeError`	Raised when a function or operation is applied to an object of an incorrect type.
`UnboundLocalError`	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
`UnicodeError`	Raised when a Unicode-related encoding or decoding error occurs.
`UnicodeEncodeError`	Raised when a Unicode-related error occurs during encoding.
`UnicodeDecodeError`	Raised when a Unicode-related error occurs during decoding.
`UnicodeTranslateError`	Raised when a Unicode-related error occurs during translating.

`ValueError`	Raised when a function gets an argument with the correct type but improper value.
`ZeroDivisionError`	Raised when the second operand of a division or the modulo operation is zero.

Table 9: Built-in exceptions available in Python

35.5. String Presentation Types

String presentation types for the *`type`* option of the **Format Specification Minilanguage** (Chapter 9.11.6).

Type	Meaning
`s`	String format. This is the default type for strings and may be omitted.
`None`	Same as `s`.

Table 10: String presentation types

35.6. Integer Presentation Types

Integer presentation types for the *`type`* option of the **Format Specification Minilanguage** (Chapter 9.11.6).

Type	Meaning
`b`	Binary format; Outputs the number in base 2.
`c`	Character; Converts the integer to the corresponding Unicode character before printing.
`d`	Decimal Integer; Outputs the number in base 10.
`o`	Octal format; Outputs the number in base 8.
`x`	Hex format; Outputs the number in base 16, using lower-case letters for the digits above 9.

`X`	Hex format; Outputs the number in base 16, using upper-case letters for the digits above 9.
`n`	Number; This is the same as `d`, except that it uses the current locale setting to insert the appropriate number separator characters.
`None`	The same as `d`.

Table 11: Integer presentation types

35.7. Floating-Point and Decimal Presentation Types

Floating point and decimal presentation types for the *`type`* option of the **Format Specification Minilanguage** (Chapter 9.11.6).

Type	Meaning
`e`	Exponent notation; Prints the number in scientific notation using the letter '`e`' to indicate the exponent; The default precision is 6.
`E`	Exponent notation; Same as `e` except it uses an upper case `E` as the separator character.
`f`	Fixed point; Displays the number as a fixed-point number; The default precision is 6.
`F`	Fixed point; Same as `f`, but converts `nan` to `NAN` and `inf` to `INF`.
`g`	General format. For a given precision `p >= 1`, this rounds the number to `p` significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude; Positive and negative infinity, positive and negative zero, and nans, are formatted as `inf`, `-inf`, `0`, `-0` and `nan` respectively, regardless of the precision; A precision of 0 is treated as equivalent to a precision of 1. The default precision is 6.
`G`	General format; Same as `g` except switches to `E` if the number gets too large; The representations of infinity and NaN are uppercased, too.

`n`	Number; This is the same as `g`, except that it uses the current locale setting to insert the appropriate number separator characters.
`%`	Percentage; Multiplies the number by 100 and displays in fixed (`f`) format, followed by a percent sign.
`None`	Similar to `g`, except that fixed-point notation, when used, has at least one digit past the decimal point; The default precision is as high as needed to represent the particular value; The overall effect is to match the output of `str()` as altered by the other format modifiers.

Table 12: Floating-point and decimal presentation types

35.8. Regular Expressions' Special Characters

The following table presents most of the special characters used in regular expressions' syntax. Due note this is not the full list[29].

When making literal searches of `.`, `^`, `$`, `*`, `+`, `?`, `{`, `}`, `[`, `]`, `\`, `|`, `(`, `)`, keep in mind that these have to be escaped (regular expression-wise, not string-wise: you can use a backslash in a raw string and it will be interpreted as escaping the character in the expression's compilation, not in the string).

Special character	Description
`.`	Matches any character except a newline
`\d`	Matches digits (0-9)
`\D`	Matches non-digits: everything except digits (0-9)
`\w`	Matches word characters (a-z, A-Z, 0-9, _)
`\W`	Matches non-word characters: everything except word characters (a-z, A-Z, 0-9, _)
`\s`	Matches whitespace (space, tab, newline)
`\S`	Matches non-whitespace: everything except whitespace (space, tab, newline)

[29] For the full list, read the re module's documentation dedicated section: https://docs.python.org/3/library/re.html#regular-expression-syntax

`\b`	Matches word boundaries[30]
`\B`	Matches non-word boundaries: everything except word boundaries
`^`	Matches the beginning of a string
`$`	Matches the end of a string
`[ ]`	Matches the characters inside the brackets; called a Character Set
`[^ ]`	Matches what is not in the Character Set
`|`	OR bitwise operator (e.g. `a|b` is read as "a or b")
`( )`	Group different sets of characters; called a Group
`m-n`	Creates a set of numbers or letters between *`m`* and *`n`*; case sensitive
`*`	Quantifier; looks for zero or more of a pattern (e.g. `\d*\w` looks for patterns that may start with one or more digits, followed by a word character)
`+`	Quantifier; looks for one or more of a pattern (e.g. `\d+\w` looks for patterns that start with at least one digit, followed by a word character)
`?`	Quantifier; looks for zero or one of a pattern (e.g. `\d?\w` looks for patterns that either start with a word character or that start with one digit followed by a word character)

[30] A word boundary can occur in one of three positions: before the first character in the string, if the first character is a word character; after the last character in the string, if the last character is a word character; between two characters in the string, where one is a word character and the other is not a word character.

`{n}`	Quantifier; looks for a pattern repeated *n* times, where *n* is an integer (e.g. `\d{3}` looks for groups of 3 digits)
`{m,n}`	Quantifier; looks for a pattern repeated *m* to *n* times, where *m* and *n* are integers (e.g. `\d{2,4}` looks for groups of 2 to 4 digits); *m* defaults to zero if not present and *n* to an infinite upper bound; the comma is obligatory in all cases

Table 13: Regular expression's special characters

35.9. re Module's Flags

The **`re`** module's compilation flags let you modify some aspects of how regular expressions work. Flags are available in this module under two names: a long and an abbreviated one. This means when passing a flag as an argument for a function you can use either form.

Multiple flags can be specified by using the bitwise OR operation (the `|` operator).

Flag	Description
`ASCII`, `A`	Makes several escapes such as `\w`, `\b`, `\s` and `\d` match only ASCII characters with the respective property
`DOTALL`, `S`	Make the period special character (`.`) match any character, including newlines.
`IGNORECASE`, `I`	Makes case-insensitive matches, that is, lowercase and uppercase letters are treated as the same
`LOCALE`, `L`	Make `\w`, `\W`, `\b`, `\B` and case-insensitive matching dependent on the current locale instead of the Unicode database

`MULTILINE`, `M`	Multi-line matching: `^` matches the beginning of the string and each newline; `$` matches the end of the string and the end of each line
`VERBOSE`, `X` (for 'extended')	

Table 14: re module's flags

35.10. HTTPError Codes

`http.server.BaseHTTPRequestHandler.responses` is a useful dictionary of response codes in that shows all the response codes used by RFC 2616.

As an extension to what's explained in the **Fetch Internet Resources with urllib** chapter (Chapter 33), this includes every HTTP error code contained in the aforementioned dictionary.

Due note, because of its extent, I separated the codes by range (the first figure includes the codes in the 100-299 range, the second includes the codes in the 300s range, the fourth includes the codes in the 400s range and the last one includes the codes in the 500s range).

(Continued in the next page)

35.10.1. 100-299 Range

```
# Table mapping response codes to messages; entries have the
# form {code: (shortmessage, longmessage)}.
responses = {
    100: ('Continue', 'Request received, please continue'),
    101: ('Switching Protocols',
          'Switching to new protocol; obey Upgrade header'),

    200: ('OK', 'Request fulfilled, document follows'),
    201: ('Created', 'Document created, URL follows'),
    202: ('Accepted',
          'Request accepted, processing continues off-line'),
    203: ('Non-Authoritative Information', 'Request fulfilled from cache'),
    204: ('No Content', 'Request fulfilled, nothing follows'),
    205: ('Reset Content', 'Clear input form for further input.'),
    206: ('Partial Content', 'Partial content follows.'),
```

Figure 317: HTTPError codes (100-299 range)

35.10.2. 300s Range

```
    300: ('Multiple Choices',
          'Object has several resources -- see URI list'),
    301: ('Moved Permanently', 'Object moved permanently -- see URI list'),
    302: ('Found', 'Object moved temporarily -- see URI list'),
    303: ('See Other', 'Object moved -- see Method and URL list'),
    304: ('Not Modified',
          'Document has not changed since given time'),
    305: ('Use Proxy',
          'You must use proxy specified in Location to access this '
          'resource.'),
    307: ('Temporary Redirect',
          'Object moved temporarily -- see URI list'),
```

Figure 318: HTTPError codes (300s range)

35.10.3. 400s Range

```
    400: ('Bad Request',
          'Bad request syntax or unsupported method'),
    401: ('Unauthorized',
          'No permission -- see authorization schemes'),
    402: ('Payment Required',
          'No payment -- see charging schemes'),
    403: ('Forbidden',
          'Request forbidden -- authorization will not help'),
    404: ('Not Found', 'Nothing matches the given URI'),
    405: ('Method Not Allowed',
          'Specified method is invalid for this server.'),
    406: ('Not Acceptable', 'URI not available in preferred format.'),
    407: ('Proxy Authentication Required', 'You must authenticate with '
          'this proxy before proceeding.'),
    408: ('Request Timeout', 'Request timed out; try again later.'),
    409: ('Conflict', 'Request conflict.'),
    410: ('Gone',
          'URI no longer exists and has been permanently removed.'),
    411: ('Length Required', 'Client must specify Content-Length.'),
    412: ('Precondition Failed', 'Precondition in headers is false.'),
    413: ('Request Entity Too Large', 'Entity is too large.'),
    414: ('Request-URI Too Long', 'URI is too long.'),
    415: ('Unsupported Media Type', 'Entity body in unsupported format.'),
    416: ('Requested Range Not Satisfiable',
          'Cannot satisfy request range.'),
    417: ('Expectation Failed',
          'Expect condition could not be satisfied.'),
```

Figure 319: HTTPError codes (400s range)

35.10.4. 500s Range

```
    500: ('Internal Server Error', 'Server got itself in trouble'),
    501: ('Not Implemented',
          'Server does not support this operation'),
    502: ('Bad Gateway', 'Invalid responses from another server/proxy.'),
    503: ('Service Unavailable',
          'The server cannot process the request due to a high load'),
    504: ('Gateway Timeout',
          'The gateway server did not receive a timely response'),
    505: ('HTTP Version Not Supported', 'Cannot fulfill request.'),
    }
```

Figure 320: HTTPError codes (500s range)

36. Finishing Notes

Lastly, I couldn't end this document without any finishing notes.

Writing this document was a brand-new experience for me: on one hand because it originated from notes I'd taken while completing Codeacademy's Python course, but on the other hand because it was the first time I wrote a technical document.

Beyond that, creating this Manual while learning Python was and still is an incredible experience because it served as a way for me to strengthen my knowledge of the language's fundamentals. Furthermore, this document helped me to establish objectives regarding what I should learn next or what I should improve, but nowadays it is also a great way for me to get a sense of my progression since the 8-9 months that have passed since I first printed "Hello World" in Python. Another important skill I've developed throughout the writing of this document is the skill to read technical documents, namely programming documentation. At first it may not seem that big of a deal, but now that I have a much better understanding of how to navigate through a technical document of a programming language, an API, etc. it helps me a great deal when it comes to write new code that involves platforms or concepts I haven't used before.

Now, looking into the future and what I already have in the present, even if it's a ton of work, I understand how valuable it is for me to write a document like this for other programming-related topics, since it will help me write down what I learn so that I can review when needed but also to make me research new topics and consequently, learn new things.