# Python 3

## Compendium

José Fernando Mendes da Silva Costa

# Summary

## Figure summary

## Table summary





## Considerations

1)

This Compendium was created based on my experience as a Python programmer, notes taken during practice, the website Programiz (https://www.programiz.com/python-programming) and what is available in the official Python documents (available at: https://docs.python.org/3/).

2)

At the footer of each page there will be a ⌂ symbol. You can click this in any page and it will bring you back to this page.

3)

Every word/phrase/expression written in with `grey background` is considered a sort of excerpt copied directly from a programming editor (IDE), which means this is proper programming syntax instead of "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 programming syntax, some things should be clarified:

Syntax 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, a placeholder or an example pertaining the context where it's shown.

`string.format()` -> in this case only the `.format()` part is the syntax for this function; `string` is only written as `string` to show that a string will be in that place when using the `.format()` function. Take a look at some practical examples for `.format()` to see how this applies:

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

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

*Figure 1: Formatting practical example*

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

If a word inside the syntax is written in bold, like this **`module`**, 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 placeholder arguments.

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

# Glossary

This section serves as sorts of an introduction to the document, as it defines some useful terms and concepts regarding the Python programming language.

1. >>>

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

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).

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*

### 4. Attribute

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

### 5. BDFL

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

### 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. Class

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

### 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`.

## 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 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.

## 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.

## 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.

## 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.

## 13. Descriptor

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

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.

### 14. Dictionary

An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl programming language.

### 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)`.

### 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.

Read more about documentations strings in its own section, here.

### 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 programming.

## 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.

The technique contrasts with the LBYL style common to many other languages such as C.

## 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.

## 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.

## 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.

### 22. Function

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

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

### 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.

### 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.

### 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).

### 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*

## 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()`.

## 28. IDLE

An Integrated Development Environment for Python.

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

## 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.

## 30. Importing

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

## 31. Importer

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

## 32. Iterable

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

Examples of iterables include all sequence types (such as list, string, and tuple) 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.

## 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.

### 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 this section.

### 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 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.

### 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).

### 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 from 0 to 255, inclusive.

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

### 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.

### 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`).

### 40. Method resolution order

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

### 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.

### 42. Mutable

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

## 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.

## 44. Object

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

## 45. Package

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

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

## 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.

## 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”.

## 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:

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

*Figure 5: for-loop using a numerical counter*

As opposed to the cleaner, Pythonic method:

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

*Figure 6: "Pythonic" for-loop*

### 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*.

### 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.

### 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`.

### 52. Text encoding

A codec which encodes Unicode strings to bytes.

### 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`.

## 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.

## 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)`.

## Basic information

In this section is given some basic information regarding Python, for example, how a variable assignment looks like or how to write comments a program.

### 1. Variable assignment

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

*Figure 7: Variable assignment examples*

### 2. Indentation

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

*Figure 8: Indentation examples*

### 3. 1-line commentary

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

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

*Figure 9: 1-line commentary example*

### 4. Multiple-lines commentary

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

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

*Figure 10: Multiple-lines commentary*

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

## 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 examples*

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.

## 6. output-formatting

There are a couple of ways to format the outputs in Python

### Substitution using %

```
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: Substitution using '%' examples*

### .format()

`string.format()`

Substitutes the curly braces for the *`arguments`* taken by `.format()`

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

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

*Figure 13: format() 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()` and Format String Syntax, please read this section.

Output in the same line:

*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 14: Output in the same line using commas example*

*Using print() arguments*

`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 *'\n'* 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 15: Output in the same line using 'end'*



## Useful information

This section goes a bit beyond the "basic" information given in the Basic Information section. It details, for example, how to convert integers from different bases, or how to have access to the contents of a module.

### 1. How to use apostrophes inside strings

To use an apostrophe inside a string use a backlash before the apostrophe, like this: `\'`.

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

*Figure 16: Using apostrophes inside strings*

### 2. Boolean operators' priority

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

### 3. String slicing

```
s = "text"
print(s[0]) #outputs t
print(s[1:2]) #outputs ex
print(s[1:]) #outputs ext
print(s[:2]) #outputs te
```

*Figure 17: String slicing examples*

### 4. Output the contents of a module

```
import module
print(dir(module))
```

*Figure 18: Output the contents of a module*

### 5. Syntax for multiple-lines code

The `\` 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: How to extend code for more than one line using '\'*

## 6. Syntax to write base 2, base 8 or base 16 integers

### Base 2 integers

Write the integer with `0b` as a prefix.

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

*Figure 20: Binary numbers examples*

### Base 8 integers

Write the integer with `0o` as a prefix.

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

*Figure 21: Octal numbers examples*

### Base 16 integers

Write the integer with `0x` as a prefix.

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

*Figure 22: Hexadecimal numbers examples*

## 7. Arbitrary argument lists

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

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

*Figure 23: Arbitrary arguments lists syntax*

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.

```
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: Arbitrary arguments list examples*



⌂

## 8. 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:

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

*Figure 25:Unpack arguments using '*'*

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 arguments using '**'*

## 9. 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. (We can't use the first line since it is generally adjacent to the string's opening quotes so 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 multi-line docstring:

```
1   def my_function():
2       """Do nothing, but document it.
3
4       No, really, it doesn't do anything.
5       """
6       pass
7
8   print(my_function.__doc__)
9   #Do nothing, but document it.
10  #
11  #    No, really, it doesn't do anything.
```

*Figure 27: Documentation string example*



## 10. Print Python's built-in elements

Using the syntax `print(locals()['__builtins__'])` prints a dictionary containing built-in exceptions, functions and attributes in Python.

Here's 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: Output Python's built-in elements*

## 11. Format string syntax

As you've read in other sections, the `.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 13: format() examples*

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

### 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_names`* 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 29: field_name examples*

### 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 29: conversion examples*

### format_spec

Lastly, there's a third optional argument: *format_spec*. *format_spec* 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.

The "formatting mini-language" will be further explained in the next section.

## 12. Format specification mini-language

As seen in the previous section, “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 you had called `str()` 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]`*

### 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 *`str`*`.format()` method.

However, it is possible to insert a curly brace with a nested replacement field. This limitation doesn’t affect the `.format()` 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'.<br>This alignment option is only valid for numeric types.<br>It becomes the default when '0' immediately precedes the field width. |
| ^ | Forces the field to be centered within the available space. |

*Table 1: 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 that the *align*ment option has no meaning in this case.

### 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). |
| space ( ) | Indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers. |

*Table 2: sign options*

### Stylistic options (formatting syntax)

*'#'*

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.

',' 

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

'_'

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.

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 alignment is given, preceding the *`width`* field by a zero ('0') character enables sign-aware zero-padding for numeric types. This is equivalent to a fill character of '0' with an alignment type of '='.



### .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`' and '`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. The `.precision` is not allowed for integer values.

### type

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

For information on String presentation types, Integer presentation types and Floating point and Decimal presentation types check their tables further ahead (or click the names to go there directly).

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.

## 13. Test an object's memory consumption

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

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

```python
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 30: sys.getsizeof() examples*

For more information about this method, please check its own section, here.

## Notable functions and methods

Python as dozens and dozens of functions and methods. Unfortunately, I couldn't write about all of them in this compendium.

Here are presented only functions that I found out to be quite useful for the programs I've created so far or simply because they are notable, and quite useful for certain purposes and situations.

For functions and methods related to threads and concurrent execution read its own section, Concurrent execution (threads).

### 1. 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 means to use 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 31: print() examples*

## 2. len()

`len(argument)`

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

*`argument`* 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 32: len() examples*

## 3. .lower()

*`string`*`.lower()`

Returns all characters of *`string`* in lowercase.

*`string`* must be a string.



```
a = "Hello World"
a.lower() #"hello world"

b = "ALL CAPS"
b.lower() #"all caps"
```

*Figure 33: lower() examples*

4. .upper()

`string.upper()`

Returns all characters of `string` in uppercase.

`string` must be a string.

```
a = "Hello World"
a.upper() #"HELLO WORLD"

b = "no caps"
b.upper() #"NO CAPS"
```

*Figure 34: upper() examples*

5. str()

`str(argument)`

Returns a string version of `argument`.

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

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

b = 1.5
str(b) #"1.5"

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

*Figure 35: str() examples*



⌂

### 6. .capitalize()

`string.capitalize()`

Returns `string` 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 36: capitalize() examples*

### 7. .format()

`string.format(argument)`

At first use, `.format()` can simply be used to replaces the placeholder curly braces `{}` inside `string` 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 13: format() examples*

Note: in case you actually need to use either `{` or `}` in the string, just simply use double braces: `{{` or `}}`.

However, this method for used for much more than this if the String Formatting syntax is applied to it. Please read this section for more information on the subject.

### 8. 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 37: input() examples*

## 9. .isalpha()

*string*.isalpha()

Returns True if all the characters in *string* 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 38: isalpha() examples*

Note: alphabetic characters are those characters defined in the Unicode character database as “Letter”.

## 10. .isdigit()

`string.isdigit()`

Return `True` if all characters in *`string`* are digits and there is at least one character, `False` otherwise.

```
print('a'.isdigit()) #False
print('1'.isdigit()) #True
print(''.isdigit()) #False
```

*Figure 39: isdigit() examples*

## 11. max()

`max(iterable)`

Return the largest item in an *`iterable`* or the largest of two or more *`arguments`*.

```
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 40: max()*

## 12. min()

`min(iterable)`

Return the smallest item in an *`iterable`* or the smallest of two or more *`arguments`*.

```
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 41: min() examples*

## 13. 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.

```
a = 1.5
b = -2

print(abs(a)) #1.5
print(abs(b)) #2
```

*Figure 42: abs() examples*

## 14. 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 43: type() examples*

## 15. int()

`int(argument)`

Returns an integer version of *`argument`*.

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

*Figure 44: int() basic examples*

## 16. 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 45: float() examples*

## 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 46: list() examples*

## 18. .append()

`list.append(x)`

Add `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 47: append() examples*

## 19. .index()

`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.

```
list_1 = ['ant', 'boar', 'cat']
print(list_1.index('cat')) #2
```

*Figure 48: index() examples*

## 20. .insert()

`list.insert(i, x)`



⌂

Insert an item 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`, and `list.insert(len(list), x)` is equivalent to `list.append(x)`.

```
list_1 = ['ant', 'boar', 'cat']
list_1.insert(1, 'dog')
print(list_1) # ['ant', 'dog', 'boar', 'cat']
```

*Figure 49: insert() examples*

## 21. range()

`range([start,] stop[, step])`[1]

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).

Pertaining the `start` and `stop` arguments, the list will start with the integer used as `start` and end in the integer right before `stop`.

[1] 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 original Python documents, `range()` is included in the "Useful functions and methods" section of this compendium.

```python
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 50: range() examples*

## 22. .sort()

`list.sort([reverse])`

Modifies `list` so that its items are sorted in a growing order.

If the items are numbers, `.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.

```python
a = [4,5,6,2,1,3]
a.sort()
print(a) #[1,2,3,4,5,6]

b = ['fish', 'eagle', 'sloth']
b.sort(reverse = True)
print(b) #['sloth', 'fish', 'eagle']

b = ['fish', 'eagle', 'sloth']
b.sort()
print(b) #['eagle', 'fish', 'sloth']
```

*Figure 51: sort() examples*

## 23. 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.

```
a = [3,1,2]
print(sorted(a)) #[1,2,3]

b = ['b', 'c', 'a']
print(sorted(b)) #['a', 'b', 'c']

c = {'c': 3, 'a': 1, 'b': 2}
print(sorted(c)) #['a', 'b', 'c']

d = {'c': 3, 'a': 1, 'b': 2}
print(sorted(d, reverse = True)) #['c', 'b', 'a']
```

*Figure 52: sorted() examples*

## 24. .pop()

*list*.pop(*[i]*)

Remove the item at index *i* in *list* and return it.

If no index *i* is specified, pop() removes and returns the last item in *list*.

```
a = [1,2,3,1,4]

print(a.pop(1)) #2
print(a) #[1,3,1,4]
```

*Figure 53: pop() examples*

## 25. .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 54: remove() examples*

## 26. .clear()

`list.clear()`

This function removes (clears) every item from `list`, returning only an empty `list`.

```
list_1 = [1,2,3]
list_1.clear()
print(list_1) #[]
```

*Figure 55: clear() examples*

## 27. .extend()

`list.extend(iterable)`

Extend `list` by appending all the items from `iterable`. Equivalent to `a[len(a):] = iterable`.

```python
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 56: Flat lists using the extend() method*

## 28. datetime.now()

`datetime.now(tz = None)`

From the **`datetime`** module.

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))`.

```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 57: datetime.now() examples*

## 29. random.rantint()

`random.randint(a, b)`

From the `random` module.

Generate an integer N such that $a \leq N \leq b$ (both $a$ and $b$ must be integers).

```python
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 58: random.randint() examples*

## 30. 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* (it doesn't 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 59: enumerate() examples*

## 31. 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).

```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 60: zip() examples*

## 32. 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 61: 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. See Floating Point Arithmetic: Issues and Limitations for more information.

### 33. .split()

`string.split([sep] [,maxsplit = integer])`

Return a list of the words in `string`, 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 62: split() examples*

## 34. .join()

`string.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 `string` 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 63: join() examples*

## 35. math.ceil()

`math.ceil(x)`

From the `math` module.

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.

```python
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 64: math.ceil() examples*

## 36. dict()

`dict(iterable)`

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

```python
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 65: dict() examples*

## 37. 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`.

```
my_list = range(16)
print(list(filter(lambda x: x % 3 == 0, my_list)))
#[0, 3, 6, 9, 12, 15]

languages = ['HTML', 'Python', 'Ruby']
print(list(filter(lambda x: x == 'Python', languages)))
#['Python']
```

*Figure 66: filter() examples*

## 38. Convert numbers from different bases (base 2, 8, 16):

### bin()

`bin(x)`

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

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

*Figure 67: bin() examples*

### 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 68: oct() examples*

### 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 69: hex() examples*

### int()

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

Converts *x* to a base 10 integer.

If *base = y* 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 10.

```
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 70: int() advanced examples*

## 39. isinstance()

`isinstance(object, classinfo)`

Returns `True` if the *object* argument is an instance of the *classinfo* argument, or 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.

### 40. 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.

### 41. 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):

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

*Figure 71: super() syntax*

```
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 72: 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*.

## 42. iter()

`iter(object[, sentinel])`

Return 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) #<str_iterator object at 0x7f569c9b5f60>
4
5   print(next(it)) #a
6
7   print(next(it)) #b
8
9   print(next(it)) #c
10
11  print(next(it)) #raises the following error
12  # Traceback (most recent call last):
13  #   File "python", line 11, in <module>
14  # StopIteration
```

*Figure 73: iter() example (1)*

One useful application of 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:

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

*Figure 74: iter() example (2)*

## 43. 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 0x7f569c9b5f60>

print(next(it)) #a

print(next(it)) #b

print(next(it)) #c

print(next(it)) #raises the following error
# Traceback (most recent call last):
#   File "python", line 11, in <module>
# StopIteration
```

*Figure 75: next() example*

## 44. __next__()

*`iterator`*`.__next__()`

Return the next item from the container.

Python defines several iterator *objects* to support iteration over general and specific sequence types, dictionaries, and other more specialized forms. The specific types are not important beyond their implementation of the iterator protocol.

Once an *iterator*'s `__next__()` *method* raises `StopIteration`, it must continue to do so on subsequent calls. Implementations that do not obey this property are deemed broken.

### 45. __repr__()

*object*.`__repr__(self)`

Called by the `repr()` built-in function to compute the "official" string representation of an *object*.

If possible, this should look like a valid Python expression that could be used to recreate an *object* with the same value (given an appropriate environment). If this is not possible, a string of the form `<...some useful description...>` should be returned.

The return value must be a string *object*.

If a *class* defines `__repr__()` but not `__str__()`, then `__repr__()` is also used when an "informal" string representation of instances of that *class* is required.

```
1   class Point3D(object):
2       def __init__(self, x, y, z):
3           self.x = x
4           self.y = y
5           self.z = z
6       def __repr__(self):
7           return "(%d, %d, %d)" %(self.x, self.y, self.z)
8
9   my_point = Point3D(1, 2, 3)
10
11  print(my_point) #(1, 2, 3)
12
13  example_2 = 'my_point'
14  print(repr(example_2)) #'my_point'
15  print(str(example_2)) #my_point
```

*Figure 76: repr() example*



⌂

## 46. json.dump()[2]

**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.

[2] Visit this section to learn more about the JSON conversion table

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.

### 47. json.dumps()[2]

**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 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*.

### 48. json.load()[2]

**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.

## 49. json.loads()[2]

**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.

## 50. 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 77: collections.Counter() examples (1)*

`Counter` *objects* have a *dictionary* interface except that they return 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 78: collections.Counter() examples (2)*

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 79: Deleting a Counter*

Counter objects support three methods beyond those available for all dictionaries:

### elements()

`CounterObject.elements()`

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.

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

*Figure 80: elements() example*

### most_common()

`most_common([n])`

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 81: most_common() example*

### subtract()

`subtract([iterable-or-mapping])`

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 82: subtract() example*

The usual dictionary *methods* are available for `Counter` *objects* except for two which work differently for counters:

### fromkeys()

`fromkeys(iterable)`

This class method is not implemented for `Counter` *objects*.

### update()

`update([iterable-or-mapping])`

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.

### Counter 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 83: collections.Counter() examples (3)*

Several mathematical operations are provided for combining `Counter` *objects* to produce *multisets* (counters 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 84: collections.Counter examples (4)*

## 51. bool()

`bool([x])`

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

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

If *x* is `False` or omitted, this 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 85: bool() examples*

## 52. 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.

```python
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 30: sys.getsizeof() examples*

## Notable statements

As with the Notable Functions and Methods section, this Notable statements section 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.

### 1. 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.

```
1  class Example1:
2      pass
3      #this class does nothing
4
5  def example1():
6      pass
7      #this function does nothing
```

*Figure 86: pass examples*

### 2. del

`del expression`

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

```
5   list_1 = [1,2,3,4,5]
6   dict_1 = {'a':1, 'b':2, 'c':3}
7
8   del list_1[2]
9   del dict_1['a']
10
11  print(list_1) #[1,2,4,5]
12  print(dict_1) #{'b':2, 'c':3}
```

*Figure 87: del examples (1)*

```
4   a = [-1, 1, 66.25, 333, 333, 1234.5]
5
6   del a[0]
7   print(a) #[1, 66.25, 333, 333, 1234.5]
8
9   del a[2:4]
10  print(a) #[-1, 1, 333, 1234.5]
11
12  del a[:]
13  print(a) #[]
```

*Figure 88: del examples (2)*

### 3. 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 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.

```
def example1(x):
    y = 2
    return bool(x > y)
#if x is bigger than y then example1 will return True, \
#else it returns False

print(example1(1)) #False; 1 > 2 is False
print(example1(3)) #True; 3 > 2 is True
print(example1(5)) #True; 5 > 2 is True
```

*Figure 89: return examples*

### 4. yield

`yield expression`

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

Citing PEP 255: "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 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, and then you can store them as you need (in this case we output them right away, and store them in a list).

```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 90: yield examples*

### 5. 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 `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 91: raise example*

### from clause

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 92: raise...from 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 93: try/except/raise example*

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 94:try/except/raise from None example*

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.

## 6. 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.

```
string = 'string'
for char in string:
    print(char)
    break
#s


while 1 < 2:
    print("Infinite Loop!")
    break
#Infinite Loop!
```

*Figure 95: break examples*

## 7. 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 really 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.

```python
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 96: continue examples*

## 8. import

`import modulename`

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

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:

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

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

*Figure 97: import 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.

## 9. 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.

## 10. 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.

You can read more about the global statement in the Variable binding section.

## 11. try

`try`

### try…except

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

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[3]:

```
1   # import sys to get the type of exception
2   import sys
3
4   randomList = ['a', 0, 2]
5
6   for entry in randomList:
7       try:
8           print('The entry is:', entry)
9           r = 1/int(entry)
10          break
11      except:
12          print('Oops!',sys.exc_info()[0],'occured.')
13          print('Next entry.')
14          print()
15  print('The reciprocal of', entry,'is', r)
```

*Figure 98: try...except examples (1)*

[3] A note about the mathematical term 'reciprocal' used in the example. 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'.

```
#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 99: try...except examples (2)*

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 `ex_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 clause 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:

```python
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 100: try...except examples (3)*

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 previously.

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

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

*Figure 101: raise using arguments example*

### 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:
    f = open("test.txt")
    #opens a test.txt file in text mode
finally:
    f.close()
    #closes the file
```

*Figure 102: try...finally example*

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

## 12. 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:

- The context expression (the expression given in the `with_item`) is evaluated to obtain a context manager;
- The context manager's `__exit__()` is loaded for later use;
- The context manager's `__enter__()` method is invoked;
- If a target was included in the with statement, the return value from __enter__() is assigned to it;
- The suite is executed;

- 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;
- 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;
- 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.

With more than one item, the context managers are processed as if multiple `with` statements were nested:

```
1  with A() as a, B() as b:
2      suite
3
4  #is equivalent to
5
6  with A() as a:
7      with B() as b:
8          suite
```

*Figure 103: with examples*



⌂

## Notable modules

Once again, this Notable modules section contains only the modules I've come across during my time programming in Python or that are related to some parts of this compendium.

### 1. 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.

```
1  import datetime
2  print(dir(datetime))
3
4  '''
5  ['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__', '__doc__',
6  '__file__', '__loader__', '__name__', '__package__', '__spec__',
7  '_divide_and_round', 'date', 'datetime', 'datetime_CAPI', 'time',
8  'timedelta', 'timezone', 'tzinfo']
9  '''
```

*Figure 104: datetime module contents*

### 2. 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 105: math module contents*

### 3. 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.

```
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 106: cmath module contents*

## 4. random

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

```
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 107: random module contents*

## 5. collections

The **collections** module implements specialized container data types providing alternatives to Python's general purpose built-in containers, dictionary, list, set, and tuple.

```
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 108: collections module contents*

## 6. json

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

```
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 109: json module contents*



⌂

## 7. 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 110: sys module contents*

## 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 111: itertools module contents*

## 9. os.path

The **`os.path`** module implements some useful functions on pathnames.

```
import os.path
print(dir(os.path))

'''
['__all__', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__',
'__spec__', '_get_sep', '_joinrealpath', '_varprog',
'_varprogb', 'abspath', 'altsep', 'basename', 'commonpath',
'commonprefix', 'curdir', 'defpath', 'devnull', 'dirname',
'exists', 'expanduser', 'expandvars', 'extsep', 'genericpath',
'getatime', 'getctime', 'getmtime', 'getsize', 'isabs', 'isdir',
'isfile', 'islink', 'ismount', 'join', 'lexists', 'normcase',
'normpath', 'os', 'pardir', 'pathsep', 'realpath', 'relpath',
'samefile', 'sameopenfile', 'samestat', 'sep', 'split',
'splitdrive', 'splitext', 'stat',
'supports_unicode_filenames', 'sys']
'''
```

*Figure 112: os.path module contents*

## 10. io

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

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.

```
1   import io
2   print(dir(io))
3
4   '''
5   ['BlockingIOError', 'BufferedIOBase', 'BufferedRWPair',
6   'BufferedRandom', 'BufferedReader', 'BufferedWriter',
7   'BytesIO', 'DEFAULT_BUFFER_SIZE', 'FileIO', 'IOBase',
8   'IncrementalNewlineDecoder', 'OpenWrapper', 'RawIOBase',
9   'SEEK_CUR', 'SEEK_END', 'SEEK_SET', 'StringIO',
10  'TextIOBase', 'TextIOWrapper', 'UnsupportedOperation',
11  '__all__', '__author__', '__builtins__', '__cached__',
12  '__doc__', '__file__', '__loader__', '__name__',
13  '__package__', '__spec__', '_io', 'abc', 'open']
14  '''
```

*Figure 113: io module contents*

## 11. 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 114: threading module contents*

## 12. 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 115: dummy_threading module contents*

The suggested usage here is:

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

*Figure 116: dummy_threading suggested usage*

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.

## 13. 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.

## 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.

```
import urllib.request
print(dir(urllib.request))

'''
['AbstractBasicAuthHandler', 'AbstractDigestAuthHandler',
'AbstractHTTPHandler', 'BaseHandler', 'CacheFTPHandler',
'ContentTooShortError', 'DataHandler', 'FTPHandler',
'FancyURLopener', 'FileHandler', 'HTTPBasicAuthHandler',
'HTTPCookieProcessor', 'HTTPDefaultErrorHandler',
'HTTPDigestAuthHandler', 'HTTPError',
'HTTPErrorProcessor', 'HTTPHandler', 'HTTPPasswordMgr',
'HTTPPasswordMgrWithDefaultRealm',
'HTTPPasswordMgrWithPriorAuth', 'HTTPRedirectHandler',
'HTTPSHandler', 'MAXFTPCACHE', 'OpenerDirector',
'ProxyBasicAuthHandler', 'ProxyDigestAuthHandler',
'ProxyHandler', 'Request', 'URLError', 'URLopener',
'UnknownHandler', '__all__', '__builtins__', '__cached__',
'__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', '__version__', '_cut_port_re',
'_ftperrors', '_have_ssl', '_localhost', '_noheaders',
'_opener', '_parse_proxy', '_proxy_bypass_macosx_sysconf',
'_randombytes', '_safe_gethostbyname', '_thishost',
'_url_tempfiles', 'addclosehook', 'addinfourl', 'base64',
```

*Figure 117: urllib.request module contents (1)*

```
'bisect', 'build_opener', 'collections', 'contextlib',
'email', 'ftpcache', 'ftperrors', 'ftpwrapper',
'getproxies', 'getproxies_environment', 'hashlib', 'http',
'install_opener', 'io', 'localhost', 'noheaders', 'os',
'parse_http_list', 'parse_keqv_list', 'pathname2url',
'posixpath', 'proxy_bypass', 'proxy_bypass_environment',
'quote', 're', 'request_host', 'socket', 'splitattr',
'splithost', 'splitpasswd', 'splitport', 'splitquery',
'splittag', 'splittype', 'splituser', 'splitvalue', 'ssl',
'string', 'sys', 'tempfile', 'thishost', 'time', 'to_bytes',
'unquote', 'unquote_to_bytes', 'unwrap', 'url2pathname',
'urlcleanup', 'urljoin', 'urlopen', 'urlparse', 'urlretrieve',
'urlsplit', 'urlunparse', 'warnings']
'''
```

*Figure 118: urllib.request module contents (2)*

### 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 119: urllib.error module contents*

### 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.

```
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 120: urllib.parse module contents*



### 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.

```
1   import urllib.robotparser
2   print(dir(urllib.robotparser))
3
4   '''
5   ['Entry', 'RobotFileParser', 'RuleLine',
6   '__all__', '__builtins__', '__cached__',
7   '__doc__', '__file__', '__loader__',
8   '__name__', '__package__', '__spec__',
9   'collections', 'urllib']
10  '''
```

*Figure 121: urlib.robotparser module contents*

### 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 122: urllib.response module contents*

## 14. 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.

## 15. socket

The **`socket`** module provides access to the BSD socket interface.

It is available on all modern Unix systems, Windows, MacOS, and probably additional platforms.

Note: some behavior may be platform dependent, since calls are made to the operating system socket APIs.

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

## Numeric Operations

This section 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` [4] | Floored quotient of `x` and `y`. |
| `x % y` [5] | 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)` [6,6] | `x` converted to floating point. |
| `complex(re,im)` [7] | 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)` [8] | `x` to the power of `y`. |

[4] 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`

[5] Not for complex numbers; instead convert to floats using `abs()` if appropriate

[6] float also accepts the strings "`nan`" and "`inf`" with an optional prefix "`+`" or "`-`" for Not a Number (NaN) and positive or negative infinity

[7] The numeric literals accepted include the digits 0 to 9 or any Unicode equivalent

[8] Python defines `pow(0, 0)` and `0 ** 0` to be 1, as is common for programming languages

| | |
|---|---|
| `x ** y` [7] | `x` to the power of `y`. |

*Table 3: Numeric operations*

## Data types

This section presents the most commonly used  data types in Python.

### 1. String

```
"This is a string"
"Hello"
"World"
```

*Figure 123: string examples*

Since strings are interpreted as a *list* it is possible to access a specific character from a string through its index (taking into account that index starts at zero and starting the count at the left-most character)

```
a = "Amazing"
print(a[2]) #outputs 'a'
print(a[5]) #outputs 'n'
```

*Figure 124: string slicing for one character*

### 2. Integer

As used normally, integers in Python are numbers that aren't a fraction, these are whole numbers.

```
1
2
3
4
```

*Figure 125: integer examples*

### 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 126: float examples*

### 4. Boolean Operators

Boolean operators can only be one of two values: `True` or `False`.

### 5. Lists

```
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 127: lists examples*

Read the section Lists for more information on this data type.

### 6. Dictionaries

```
dict_1 = {'a':1, 'b':2, 'c':3}
dict_2 = {
    'd':4,
    'e':5,
    'f':6
}
dict_3 = {} #empty dictionary
```

*Figure 128: dictionaries examples*

Read the section Dictionaries for more information on this data type.

## 7. Tuples

```
#example of a tuple
t = 12345, 54321, 'hello!'
print(t[0]) #12345
print(t) #(12345, 54321, 'hello!')
```

*Figure 129: tuple example*

Read the sections Tuples for more information on this data type.

## 8. Sets

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
#show that duplicates have been removed
print(basket) #{'apple', 'orange', 'pear', 'banana'}
```

*Figure 130: set example*



## Lists

Lists are a type of data types to begin with. These are 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 117: lists examples*

### How to create lists

Lists can be created through several ways (note the functions and *methods* shown won't be explained in this section, but in separate sections of this compendium):

- Using a pair of square brackets to denote the empty list

```
4   list_2 = [] #empty list
```

*Figure 131: Create an empty list using square brackets*

- Using square brackets, separating items with commas

```
1   list_1 = ['a', 'b', 'c']
2   list_2 = [1, 2,'a', 3, 'b', 'c']
```

*Figure 132: Examples of lists with content*

- Using a *list comprehension*

```
1   list_1 = [x for x in range(5)] #[0,1,2,3,4]
```

*Figure 133: Creating a list using list comprehension*

- Using the type constructor

```
1   list_4 = list(range(5)) #[0,1,2,3,4]
```

*Figure 134: Creating a list using list()*

## Lists' operations

### 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.

```
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 135: Item callout in lists*

### Item overwrite

To alter 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 136: Item overwriting in lists*

### Add items to a list

To add items to a list use the `list.append(x)` method:



```
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 137: Adding items to a list using append()*

Do note that with using the `list.append(x)` method will append `x` at the end of `list`.

In order to add an item to a specific index of a list the `list.insert(i, x)` method should be used. Regarding index overwriting, if you were to `insert()` a new item to the index `i` of a list with an already existing item at that index, the original item wouldn't be overwritten, instead all items starting at index `i` would move one index forward):

```
list_1 = ['ant', 'boar', 'cat']
list_1.insert(1, 'dog')
print(list_1) # ['ant', 'dog', 'boar', 'cat']
```

*Figure 138: Adding items to a list using insert()*

### 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 and then obtain a new list that contains all items from `list_a` and `list_b`:

```
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 139: Groupping lists example*

### Check an item's index

To check what is the index of a certain item, `list.index(x)` should be used:



```
list_1 = ['ant', 'boar', 'cat']
print(list_1.index('cat')) #2
```

*Figure 140: Find an item's index in a list using index()*

### Sort a list

To sort a list, simply use the `list.sort()` method:

```
a = [4,5,6,2,1,3]
a.sort()
print(a) #[1,2,3,4,5,6]

b = ['fish', 'eagle', 'sloth']
b.sort(reverse = True)
print(b) #['sloth', 'fish', 'eagle']

b = ['fish', 'eagle', 'sloth']
b.sort()
print(b) #['eagle', 'fish', 'sloth']
```

*Figure 141: Sort a list using sort()*

### Delete/Remove items from a list

To delete/remove an item from a list there are four ways of doing it:

`list.pop([i])` -> remove the item at index `i` in `list` and return it; if no index is specified, `pop()` removes and returns the last item in `list`.

```
#general syntax
list_name.pop(item_index)

a = [1,2,3,1,4]

print(a.pop(1)) #2
print(a) #[1,3,1,4]
```

*Figure 142: Remove an item from a list using pop()*



`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 143: Remove an item from a list using remove()*

`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`).

```
#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 144: Delete (part of) a list using del*

`list.clear()` -> this method actually removes every item from `list`, returning only an empty `list`.

```
#general syntax
list_name.clear()

list_1 = [1,2,3]
list_1.clear()
print(list_1) #[]
```

*Figure 145: Delete all the content in a list using clear()*

### Flat 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 (flat the list) you have two ways of doing it (actually, there's a third one, but we'll get to that in a bit).

#### *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 (flat) 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 1D 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:

```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 146: Flat lists using nested for loops*

### *Using extend()*

The second method still involves using a `for` loop, but this time we add the `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 `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 example below using the same list from the previous example:

```python
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 56: Flat lists using the extend() method*



⌂

For more information about the `extend()` method please read its own separate section here.

### *Using recursion*

Finally, we arrive at the third method: recursion. I've shown you how to flat lists using both nested `for` loops and using a combination of a `for` loop and the `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). 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 `extend()`).

Then let's stick with the `extend()` method, but use it in a recursive function context. To raise the bar even more, let's have two new list examples: one with depth 3 and the other with depth 4. Using the same function for both we'll be able to flat both nonetheless, which proves the automatization of the process.

```
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 147: Flat lists using recursion*



⌂

## 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[9].

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result will be a new list created from evaluating the expression in the context of the `for` and `if` clauses.

[9] 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 148: List comprehension syntax and examples*



⌂

## 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 118: dictionaries examples*

### 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 through 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 above).

### Dictionaries' operations

The main operations on a dictionary are storing a value with some key and extracting the value given the key. If you store 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:

## 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 149: Output the contents of a dictionary*

## 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 150: Output the value of a specific key*



## Add a new key 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 151: How to add new keys to a dictionary*

## Delete specific key-value pairs

```
#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 152: Delete specific key-value pairs in a dictionary using del*

## 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 153: How to overwrite a key's value in a dictionary*

## Example of a dictionary with different data types as values

```
dict_1 = {
    'fish': ['a', 'b', 'c', 'd'],
    'cash': -4483,
    'luck':'good'
}

print(dict_1)
#{'cash': -4483, 'fish': ['a', 'b', 'c', 'd'], 'luck': 'good'}
```

*Figure 154: Example of a dictionary containing different data types*

## Output a dictionary's values

```
dict_1 = {
    'fish': ['a', 'b', 'c', 'd'],
    'cash': -4483,
    'luck':'good'
}

for item in dict_1:
    print(dict_1[item])
#-4483
#['a', 'b', 'c', 'd']
#good
```

*Figure 155: Iterate through a dictionary to output all its values one at a time*



### Create a dictionary with dict()

```python
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 156: Create a dictionary using dict()*

### 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 157: dictionary comprehension syntax and examples*



# Tuples

Just like lists and strings, tuples are a sequence data type in Python.

## Theory

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).

Now that you a have a general idea of what a tuple is, take a look at the following examples of tuples:

```
#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 158: 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 159: Empty tuples vs. tuples with one item*

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 160: tuple sequence unpacking*

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*.

### Advantages of tuples over lists

To finish off the tuples section, 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.

## Sets

Python also includes a data type for sets. A set is an unordered collection with no duplicate elements.

### 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.

### Examples

Here are some practical examples of a set and its operations:

```
1  basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
2  #show that duplicates have been removed
3  print(basket) #{'apple', 'orange', 'pear', 'banana'}
```

*Figure 120: set example*

### Set 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:

```python
#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 161: sets' operations examples*

## Set comprehension

Similarly to list and dictionary comprehensions, set comprehensions are also supported:

```python
a = {x for x in 'abracadabra' if x not in 'abc'}
print(a) #{'r', 'd'}
```

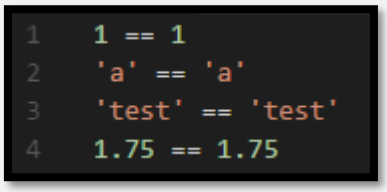
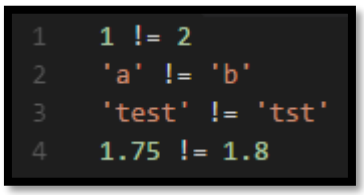
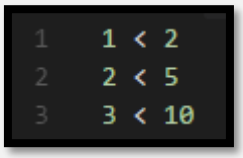
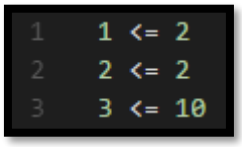
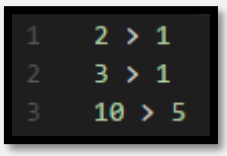
*Figure 162: set comprehension example*



⌂

## Comparison operators

This section presents and explains the comparison operators available in Python.

| Operator | Name | Example |
|---|---|---|
| `==` | Equal to (different from `=` which is used to assign a variable). |  <br>*Figure 163: '==' operator examples* |
| `!=` | Different from/ not equal to. |  <br>*Figure 164: '!=' operator examples* |
| `<` | Lesser than. |  <br>*Figure 165: '<' operator examples* |
| `<=` | Lesser or equal to. |  <br>*Figure 166: '<=' operator examples* |
| `>` | Bigger than. |  <br>*Figure 167: '>' operator example* |

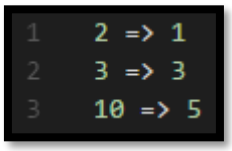
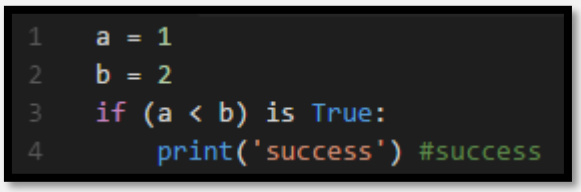
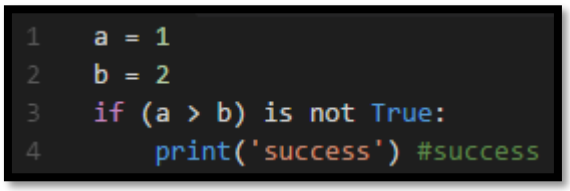
| | | |
|---|---|---|
| `>=` | Bigger or equal to. |  <br>*Figure 168: '>=' operator examples* |
| `is` | Object identity. |  <br>*Figure 169: 'is' operator example* |
| `is not` | Negated object identity. |  <br>*Figure 170: 'is not' operator example* |

*Table 4: Comparison operators*



⌂

## Boolean Operators

This section presents and explains the three boolean operators of Python: `or`, `and` and `not`.

### 1. or

`or`[10]

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 171: 'or' boolean operator examples*

### 2. and

`and`[11]

Tests two arguments: returns `True` if both arguments are `True`, else returns `False`.

[10] It only evaluates the second argument if the first one is `False`
[11] It only evaluates the second argument if the first one is `True`

```python
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 172: 'and' boolean operator examples*

### 3. not

`not`[12]

Tests one argument: if the argument is `True` returns `False`, else returns `True`.

```python
a = 1 > 2
print(not(a)) #True

print(not(1+1 == 2)) #False
```

*Figure 173: 'not' boolean operator examples*

[12] `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



⌂

## Conditional clauses

This section tackles the conditional clauses found in Python: namely `if`, `elif` and `else`.

### 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 an `else`.

```
a = 1
b = 2

if a < b:
    print(a) #outputs 1 because a < b is True

if a > b:
    print(a) #outputs nothing because a > b is False
```

*Figure 174: if clause examples*

Note: 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.

Note: in case the `if` returns `True` then the following `elif` and `else` clauses will be completely ignored.

### 2. elif

`elif expression:`

Short for *else if*.

Always written after the `if` 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:
    print(a) #outputs 1 because a < b is True
elif a == 1:
    print(b) #even though the elif returns True it is ignored because the if returns True

if a > b:
    print(a) #outputs nothing because a > b is False
elif a == b:
    print(a) #returns False so the code isn't executed
elif a == 1:
    print(a) #outputs 1; this elif returns True so the code is executed
```

*Figure 175: if/elif examples*

### 3. else

`else:`

The last clause of a conditional chain.

`else` is only executed if the `if` condition, and the `elif` conditions, in case they exist, returned `False`.

This means the `else` isn't tested at all in case an `if` or an `elif` condition is `True`.

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:
    print(a) #outputs 1 because the if returns True
else:
    print(b) #the else is ignored because the if returns True

if a > b:
    print(a) #this if returns False
else:
    print(b) #outputs 2 because the if returned False

if a > b:
    print(a + b) #this returns False
elif a == b:
    print(a + b) #this returns False
elif b == 1:
    print(a + b) #this returns False
else:
    print(a + b) #outputs 3; because all the previous conditions returned False the else is executed
```

*Figure 176: if/elif/else examples*

# Functions

This section 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.

## 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 adapt it to its different context. Using functions usually enhances the comprehensibility and quality of the program. It also lowers the cost for development and maintenance of the software.

Functions are known under various names in programming languages: subroutines, routines, procedures, methods, or subprograms.

### Function definition

Example of a generic function definition in Python:

```
1   def function_name(parameters):
2       statements a.k.a body of the function
```

*Figure 177: function definition syntax*

Examples of functions:

```
1   def hello_world():
2       print("Hello World!") #outputs Hello World!
3
4   hello_world()
```

*Figure 178: function 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 179: function example (2)*

From the examples shown above we conclude the following:

- A function definition always starts with the `def` keyword;
- 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 is given one argument);
    - A function may be called as many times as necessary.

### Recursion

Recursion is the process of defining something in terms of itself. Applying this concept in Python concepts, we obtain recursive functions: function that call themselves.

Take a look at a simple example[13] of a recursive function:

[13] Factorial of a number x is 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 180: example of a recursive function*

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 sometimes recursion is sometimes hard to follow;
- Recursive calls are inefficient: they consume a lot of memory and time;

- Recursive functions are hard to debug.

### 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:

```
def lambda_function(arguments):
    return expression
```

*Figure 181: "normal" function equivalent of a lambda function*

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:

```
#general syntax
lambda arguments: expression
```

*Figure 182: lambda function syntax*

Below you can see some practical uses and comparisons between using `lambda` functions and "normal" functions:

```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 183: "normal" function vs. lambda function 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 184: "normal" function vs. lambda function example (2)*



## Loops

In this section the thematic of loops in Python is brought to light, namely the `for` loops and `while` loops, and their intrincacies.

### 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 (for example a list or a string), in the order that they appear in the sequence.

Below are some examples of the general syntax for a `for` loop and a couple of examples of how to use it.

```
#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 185: for-loop syntax and 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 5 times). In this cases the `for` loop will go hand-in-hand with the `range()` type.

What the `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 the `range()` in action, and in case you want to learn more about this Python built-in type check out the `range()` section of this compendium.

```
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 186: for-loop/range() example*

### For/Else loops

Another element you can combine with a `for` loop is an `else` conditional statement.

The construction for this combination is similar to the one used in the `if/else` conditions, 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 187: 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`).



⌂

### While loops

The `while` statement, unlike in 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 188: while-loop examples*

### 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*:

```python
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 189: infinite while-loop 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):

```python
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 190: infinite while-loop with break*

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 191: infinite while-loop with a counter variable*

### 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 192: 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`.

## Imports

These sections presents the different ways of importing some code, more specifically in this case, importing modules.

In Python, there are 3 ways to import modules and its functions and variables:

### Generic import

Imports entire modules.

```
1   import math
2   print(math.sqrt(25)) #outputs 5.0
```

*Figure 193: generic import example*

### Function import

Imports specific functions of a certain module.

```
1   from math import sqrt
2   print(sqrt(25)) #outputs 5.0
```

*Figure 194: function import example*

### 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 195: universal import example*

## Bitwise operations

This section tackles bitwise operations applied to Python, from the theory to the practical cases, including the operations themselves and "tools" such as bit masks.

### 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 196: examples of binaries 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 the program will instead output the conversion of that binary number to base 10, which in this case would be 3.

In order to output the binary number, use the `bin()` function. The same applies for normal operations such as addition or subtraction while 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 197: bin() applications and examples*

Another important thing to keep in mind about binary numbers is that, as you can see, they are written with ones and zeros. What's worth noting is that a bit (digit) being `1` means that bit is `True`, while a bit being `0` (zero) means that bit is `False`.

Before going into detail about each possible operation, look at the following table, which contains all bitwise operations, sorted in 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*

### 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 198: bitwise '|' syntax and examples*

Here's a summary of all the possible outcomes while using *or*:

- `0 | 0 == 0`
- `0 | 1 == 1`
- `1 | 1 == 1`
- `1 | 0 == 1`

### 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`).

```python
#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 199: bitwise '^' syntax and 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`

### And (&)

Just like with the *or*, the *and* works quite similarly as the boolean operator *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 200: bitwise '&' syntax 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`

### Bit shifts (<<; >>)

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 201: left and right bit shifts syntax and examples*

It's also worth noting three things about bit shifts:

1. Negative shift counts are illegal and cause a `ValueError` to be raised
2. A left shift by *n* bits is equivalent to multiplication by `pow(2, n)` without overflow check
3. A right shift by *n* bits is equivalent to division by `pow(2, n)` without overflow check

### Not (~)

This is also similar to the boolean operator *not*, except here it's applied to bitwise operations.

The simple 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)`).

```
#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 202: bitwise '~' syntax and examples*

### 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.

```python
#in this example the third bit, counting from the right, of num1 is turned on
#this is accomplished by using mask1, which has the third bit on
def flip_bit1(num1):
    mask1 = 0b1111
    result1 = num1 | mask1 #use the or operator so the third bit is turned on
#no matter if it was on or off in the beginning
    return result1
print(bin(flip_bit1(0b1011))) #0b1111


#in this example, again counting from the right, the nth bit of num2 is turned on
#this is accomplished by using mask2, which has its only bit on
#if instead of wanting to turn the nth bit on you wanted to turn it off,
#then instead mask2 should have its bit turned off (0b0)
def flip_bit_2(num2, n):
    mask2 = (0b1 << (n-1))
    result2 = num2 ^ mask2
    return bin(result2)
print(flip_bit_2(0b1010, 3)) #0b1110
```

*Figure 203: bitmask examples*



## Class

In this section 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.

### 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

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.

*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*.

### Attribute

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`.

## 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.

### Variable binding

Next, let me show you a practical example of how the `global` and `nonlocal` statements can affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

*Figure 204: global and nonlocal statements examples (1)*



⌂

```
#outputs
#After local assignment: test spam
#After nonlocal assignment: nonlocal spam
#After global assignment: nonlocal spam
#In global scope: global spam
```

*Figure 205: global and nonlocal statements examples (2)*

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.

### 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 206: class definition syntax*

Now let's analyze each component:

- When starting a *class* you need to start with the `class` keyword, it's 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).

## Class objects

*Class objects* support two kinds of operations: *attribute references* and instantiation.

### Attribute references

*Attribute references* use the standard syntax used for all *attribute references* in Python: `obj.name`.

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 207: 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.

*Class* instantiation uses function notation. Just pretend that the *class object* is a function with no parameters that returns a new instance of the *class*. For example (assuming the above class):

```
x = MyClass()
```

*Figure 208: class instantiation example*

creates a new instance of the *class* and assigns this *object* to the local variable `x`.

### __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 209: __init__() structure*

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 197: 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 210: class containing __init__() example*

### Instance objects

The only operations understood by *instance objects* are *attribute references*. There are two kinds of valid *attribute references*, *data attributes* and *methods*.

#### 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 211: data attributes example*

### methods

The other kind of instance *attribute reference* is a *method*.

A *method* is a function that "belongs to" an *object*. In Python, the term *method* 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 exclusively used to mean *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 data attributes, `x.f` is a valid *method* reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not (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.

### Method objects

Usually, a *method* is called right after it is bound:

```
x.f()
```

*Figure 212: general method example*

Once again, in the `MyClass` example, 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 can be stored away and 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 213: calling a stored method object example*

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 object* and the function *object* 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, and the function *object* is called with this new argument list.

### 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*:

```python
class Dog:
    kind = 'canine' #class variable shared by all instances

    def __init__(self, name): #instance variable unique to each instance
        self.name = name

dog_1 = Dog('Fido')
dog_2 = Dog('Buddy')

print(dog_1.kind) #outputs 'canine'; this is shared by all Dogs
print(dog_1.name) #outputs 'Fido'; unique to dog_1
print(dog_2.kind) #outputs 'canine'; this is shared by all Dogs
print(dog_2.name) #outputs 'Buddy'; unique to dog_2
```

*Figure 214: instance variables examples*

*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:

```python
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 215: mistaken use of a class variable*

For correct usage, the following code should be used instead, which uses an instance variable for the *list*:

```
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 216: mistaken use of a class variable corrected*

## 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 217: different ways of creating a class' attributes*

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 218: methods calling other methods example*

*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__`.

### 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 219: class inheritance syntax*

Now let's analyze each component:

- `DerivedClass` is the name of a new *class* which will inherit from an existing *class* (this *class* will be written between parenthesis);
- `BaseClass` is the *class* from which `DerivedClass` will inherit;
- The rest is the content of `DerivedClass` (this does not include the content 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 220: class inheriting from a class in 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()` 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()` 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`.

### 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 221: class multiple inheritance*

Please due note that in the example above I've only chosen to *inherit* from three base/parent *classes*, but of course you can choose more or less than three.

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 was searched for 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*.

### super() application example[14]

```
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 71: super() example*

### 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,

[14] For detailed information about `super()`, read its own section here.

called *name mangling*. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) 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 222: private variables/name mangling example*

Note that 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.

### 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 223: empty class definition application*

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*.

### Iterators

By now you have probably noticed that most container *objects* can be looped over using a `for` statement:

```python
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 224: iterators examples*

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 0x7f569c9b5f60>

print(next(it)) #a

print(next(it)) #b

print(next(it)) #c

print(next(it)) #raises the following error
# Traceback (most recent call last):
#   File "python", line 11, in <module>
# StopIteration
```

*Figure 225: "mechanic" behind an iterator*

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`:



⌂

```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 226: creating a manual iterator inside a class*

## 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 227: generator example*

Anything that can be done with *generators* can also be done with *class*-based *iterators* as described in the Iterators section. 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.

### 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).

#### 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.

Examples:

```
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 228: generator expressions examples*

## 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.

### File input

Starting with the basics, to receive input (open) 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 arguments: `open(file, mode)`.

```
example1 = open('filename', 'w')
#this open the file filename, in write mode
```

*Figure 229: open() syntax*

The first argument must be the name of the file, written as a string. The second argument is a string containing one or two letters that specify how the file will used.

For example, in this case, the file `'filename'` is going to be opened in `'w'`rite mode, which would mean 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'` | Binary mode. |
| `'t'` | Text mode. |

| | |
|---|---|
| `'+'` | Open a disk file for updating (reading and writing). |
| `'U'` | Universal newlines mode (deprecated). |

*Table 6: open() modes*

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.

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:

```
1  with open('workfile') as f:
2      read_data = f.read()
3
4  print(f.closed) #True
```

*Figure 230: with/open() example*

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 `f.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 231: close() behavior after using with/open() with a file*

## Methods of file objects

For the following examples please assume that a *file object* `f` has been called beforehand.

### .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`.

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 232: read() examples*

### .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 233: 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()`.

### .write()

`f.write(string)` writes the contents of `string` to the file, returning the number of characters written.

```
f.write('This is a test\n')
#15
```

*Figure 234: 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:

```
value = ('the answer', 42)
#convert the tuple to string
s = str(value)
f.write(s)
#18
```

*Figure 235: converting data to use it with write()*

### .tell()

`f.tell()` returns an integer giving 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.

### .seek()

To change the *file object*'s position, use `f.seek(offset[,whence])`.

The position is computed from adding `offset` to a reference point; the reference point is selected by the `from_what` argument.

A `from_whence` 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_whence` 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
f.seek(5) #Go to the 6th byte in the file
#5
f.read(1)
#b'5'
f.seek(-3, 2) #Go to the 3rd byte before the end
#13
f.read(1)
#b'd'
```

*Figure 236: seek() examples*

In text files (those opened without a `b` in the mode string), only seeks relative to the beginning of the file are allowed (the exception being seeking to the very file end with `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.

## 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 [15](JavaScript Object Notation).

The standard module called **`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*.

Between *serializing* and *deserializing*, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

If you have an *object* x, you can view its JSON string representation with a simple line of code:

```
1  import json
2  json.dumps([1, 'simple', 'list'])
3  #'[1, "simple", "list"]'
```

*Figure 237: view an object's representation in JSON using dumps()*

Another variant of the `dumps()` function, called `dump()`, simply serializes the object to a text file. So, if $f$ is a text file object opened for writing, we can do this:

[15] 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.

```
1    json.dump(x, f)
```

*Figure 238: json.dump() syntax*

To decode the object again, if `f` is a text file *object* which has been opened for reading:

```
1    x = json.load(f)
```

*Figure 239: json.load() syntax*

This simple *serialization* technique can handle lists and dictionaries, but serializing arbitrary *class* instances in JSON requires a bit of extra effort. The reference for the `json` module contains an explanation of this.

## Concurrent execution (threads)

Threads allow programs to run operations concurrently in the same process. Normally a program written with Python would only execute line by line, but if you use threads you'll be able to execute different operations at the same time, which in turn reduces the time needed to complete your program.

### 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 240: Concurrent execution example*

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`. What `product` does by itself is print a string with the product of `a` and `b`, sleep (pause the execution of the program) for three seconds and then print a final string `'Finished'`. In the last line of the program there's a last `print` statement used for a string.

However, you don't see any call for the `product` function, 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` using `.Thread()` from the **`threading`** module targeting the `product` function, passing to it the arguments `(2,3)` to be used as `a` and `b` in the function.

Though, this isn't what calls the function. It's the `t1.start()` line that calls the function (more information ahead about `.start()`). 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 `print('Concurrent execution!!!')`. After three seconds of being asleep, product executes its last line: `print('Finished')`.

## Theory

Now that you are familiar with a thread 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 execute part of the function, put it to sleep and then execute the code outside of the thread while the function is inactive.

For this to happen some basic things were needed: import the **`threading`** module, create a `Thread` *object* (containing a function in the example) which is then assigned to a variable, and lastly call the `.start()` method on the thread.

## Important thread-related functions and methods

Now let's examine some of the functions and methods from this module.

### threading.Thread()

We started by using the threading.Thread constructor we 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 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.

### .start()

*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*.

### .run()

*thread*.run()

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.

### .join()

*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 or 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 decide whether a timeout happened – if the thread is still alive, 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 many 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 raises the same exception.

### .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 function `enumerate()` returns a list of all alive threads.

### 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.

### 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.

### 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.

## 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 signalling mechanism.

A thread can be made daemonic if when creating a thread using **`threading.`**`Thread()` you don't leave the argument *`daemon`* as *`None`*, so assign *`daemon = True`*.

### .isdaemon()

*`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`.

# How to fetch internet resources

## Introduction

In order to fetch internet resources, the **`urllib`** package is very handy, specifically the **`urllib.request`** module.

**`urllib.request`** is a Python module for fetching URLs (Uniform Resource Locators). It offers a very simple interface, in the form of the `urlopen()` function. This is capable of fetching URLs using a variety of different protocols. It also offers a slightly more complex interface for handling common situations - like basic authentication, cookies, proxies and so on. These are provided by *objects* called handlers and openers.

**`urllib.request`** supports fetching URLs for many "URL schemes" (identified by the string before the ":" in URL - for example `"ftp"` is the URL scheme of "ftp://python.org/") using their associated network protocols (e.g. FTP, HTTP). We will be focusing on the most common case, HTTP.

For straightforward situations, `urlopen()` is very easy to use. But as soon as you encounter errors or non-trivial cases when opening HTTP URLs, you will need some understanding of the HyperText Transfer Protocol (HTTP). The most comprehensive and authoritative reference to HTTP is RFC 2616. This is a technical document and not intended to be easy to read. This section of the Compendium aims to illustrate how to use **`urllib`**, with enough detail about HTTP to help you through.

As everything in this document, it is not intended to replace the urllib.request docs, but to help you enough that you understand what you're dealing with.

To write this section, I based myself heavily on this document.

## Fetching URLs

The simplest way to use **`urllib.request`** is:

```
import urllib.request
with urllib.request.urlopen('http://python.org/') as response:
    html = response.read()
```

*Figure 241: urllib.request basic example*

If you wish to retrieve a resource via URL and store it in a temporary location, you can do so via the `urlretrieve()` function:

```
import urllib.request
local_filename, headers = urllib.request.urlretrieve('http://python.org/')
html = open(local_filename)
```

*Figure 242: urlretrieve() example*

Many uses of **`urllib`** will be that simple (note that instead of an `'http:'` URL we could have used a URL starting with `'ftp:'`, `'file:'`, etc.). However, here we're focusing on HTTP.

HTTP is based on requests and responses - the client makes requests and servers send responses. **`urllib.request`** mirrors this with a Request *object* which represents the HTTP request you are making. In its simplest form, you create a Request *object* that specifies the URL you want to fetch. Calling `urlopen()` with this Request *object* returns a response *object* for the URL requested. This response is a *file-like object*, which means you can for example call `read()` on the response:

```
import urllib.request

req = urllib.request.Request('http://python.org/')
with urllib.request.urlopen(req) as response:
    the_page = response.read()
```

*Figure 243: Request object usage example*

As said above, **`urllib.request`** makes use of the same Request interface to handle all URL schemes. So, for example, you can make an FTP request like this:

```
req = urllib.request.Request('ftp://example.com/')
```

*Figure 244: urllib.request with an FTP URL scheme example*

In the case of HTTP, there are two extra things that Request *objects* allow you to do: (1) you can pass data to be sent to the server and (2), you can pass extra information ("metadata") about the data or the about request itself, to the server - this information is sent as HTTP "headers".

### Data

Sometimes you want to send data to a URL (often the URL will refer to a CGI (Common Gateway Interface) script or other web application).

With HTTP, this is often done using what's known as a POST request. This is often what your browser does when you submit a HTML form that you filled in on the web. Not all POSTs have to come from forms: you can use a POST to transmit arbitrary 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 a function from the **`urllib.parse`** module.

```
import urllib.parse
import urllib.request

url = 'https://docs.python.org'
values = {'name' : 'Jose Costa',
          'language' : 'Python' }

data = urllib.parse.urlencode(values)
data = data.encode('ascii') # data should be bytes
req = urllib.request.Request(url, data)
with urllib.request.urlopen(req) as response:
   the_page = response.read()
```

*Figure 245: encode data for the Request object*



⌂

Note that other encodings are sometimes required (see HTML Specification, Form Submission for more details).

If you do not pass the *data* argument, **urllib** uses a GET request.

One way in which GET and POST requests differ is that POST requests often have "side-effects": they change the state of the system in some way (for example, by placing an order for a product).

Though the HTTP standard makes it clear that POSTs are intended to always cause side-effects, and GET requests never to cause side-effects, nothing prevents a GET request from having side-effects, nor a POST requests from having no side-effects.

Data can also be passed in an HTTP GET request by encoding it in the URL itself:

```
1   import urllib.request
2   import urllib.parse
3   data = {}
4   data['name'] = 'Jose Costa'
5   data['language'] = 'Python'
6   url_values = urllib.parse.urlencode(data)
7   print(url_values)
8   #name=Jose+Costa&language=Python
9   url = 'https://docs.python.org'
10  full_url = url + '?' + url_values
11  data = urllib.request.urlopen(full_url)
```

*Figure 246: encoding data in the URL for an HTTP GET request*

Notice that the full URL is created by adding a ? to the URL, followed by the encoded values.

### Headers

Some websites dislike being 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, for example, `Python-urllib/2.5`), which may confuse the site, or just not 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. The following example makes the same request as above, but identifies itself as a version of Internet Explorer.

```
1   import urllib.parse
2   import urllib.request
3
4   url = 'https://docs.python.org'
5   user_agent = 'Mozilla/5.0 (Windows NT 6.1; Win64; x64)'
6   values = {'name': 'Jose Costa',
7             'language': 'Python'}
8   headers = {'User-Agent': user_agent}
9
10  data = urllib.parse.urlencode(values)
11  data = data.encode('ascii')
12  req = urllib.request.Request(url, data, headers)
13  with urllib.request.urlopen(req) as response:
14      the_page = response.read()
```

*Figure 247: headers application example*

### Handling exceptions

`urlopen()` raises `URLError` when it cannot handle a response (though as usual with Python APIs, built-in exceptions such as `ValueError`, `TypeError` etc. may also be raised).

`HTTPError` is the *subclass* of `URLError` raised in the specific case of HTTP URLs.

The exception *classes* are exported from the **`urllib.error`** module.

### URLError

Often, `URLError` is raised because there is no network connection (no route to the specified server), or the specified server doesn't exist. In this case, the exception raised will have a 'reason' attribute, which is a tuple containing an error code and a text error message.

Take a look at the example below:

```
req = urllib.request.Request('http://www.pretend_server.org')
try:
    urllib.request.urlopen(req)
except urllib.error.URLError as e:
    print(e.reason)
#(4, 'getaddrinfo failed')
```

*Figure 248: URLError example*

### HTTPError

Every HTTP response from the server contains a numeric "status code". Sometimes the status code indicates that the server is unable to fulfil the request. The default handlers will handle some of these responses for you (for example, if the response is a "redirection" that requests the client fetch the document from a different URL, **`urllib`** will handle that for you). For those it can't handle, `urlopen()` will raise an `HTTPError`. Typical errors include '404' (page not found), '403' (request forbidden), and '401' (authentication required).

The `HTTPError` instance raised will have an integer 'code' attribute, which corresponds to the error sent by the server.

### Error codes

Because the default handlers handle redirects (codes in the 300s range), and codes in the 100–299 range indicate success, you will usually only see error codes in the 400–599 range.

`http.server.BaseHTTPRequestHandler.responses` is a useful dictionary of response codes in that shows all the response codes used by RFC 2616. You can read it in full extent in this section.

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 as well as the code attribute, it also has `read()`, `geturl()`, and `info()`, *methods* as returned by the **`urllib.response`** module:

```
req = urllib.request.Request('http://www.python.org/fish.html')
try:
    urllib.request.urlopen(req)
except urllib.error.HTTPError as e:
    print(e.code)
    print(e.read())

'''
404
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">\n\n\n<html
  ...
  <title>Page Not Found</title>\n
  ...
'''
```

*Figure 249: urllib.response methods application*

### Preparing for HTTPError or URLError

If you want to be prepared for `HTTPError` or `URLError` there are two basic approaches.

#### First approach

```
from urllib.request import Request, urlopen
from urllib.error import URLError, HTTPError
req = Request(someurl)
try:
    response = urlopen(req)
except HTTPError as e:
    print('The server couldn\'t fulfill the request.')
    print('Error code: ', e.code)
except URLError as e:
    print('We failed to reach a server.')
    print('Reason: ', e.reason)
else:
    # everything is fine
```

*Figure 250: First approach to preparing for HTTP or URL errors*

The `except HTTPError` must come first, otherwise `except URLError` will also catch an `HTTPError`.

### Second approach

```
from urllib.request import Request, urlopen
from urllib.error import URLError
req = Request(someurl)
try:
    response = urlopen(req)
except URLError as e:
    if hasattr(e, 'reason'):
        print('We failed to reach a server.')
        print('Reason: ', e.reason)
    elif hasattr(e, 'code'):
        print('The server couldn\'t fulfill the request.')
        print('Error code: ', e.code)
else:
    # everything is fine
```

*Figure 251: Second approach to preparing for HTTP or URL errors*

## info() and geturl()

The response returned by `urlopen()` (or the `HTTPError` instance) has two useful methods: `info()` and `geturl()` and which are defined in the module **`urllib.response`**.

### geturl()

`page.geturl()`

This returns the real URL of the *page* fetched. This is useful because `urlopen()` (or the opener *object* used) may have followed a redirect. The URL of the *page* fetched may not be the same as the URL requested.

### info()

`page.info()`

This 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.

Typical headers include ‘Content-length’, ‘Content-type’, and so on. See the Quick Reference to HTTP Headers for a useful listing of HTTP headers with brief explanations of their meaning and use.

### Openers and handlers

When you fetch a URL you use an opener (an instance of the perhaps confusingly-named `urllib.request.OpenerDirector`).

Normally we have been using the default opener - via `urlopen()` - but you can create custom openers. Openers use handlers. All the “heavy lifting” is done by the handlers. Each handler knows how to open URLs for a particular URL scheme (http, ftp, etc.), or how to handle an aspect of URL opening, for example HTTP redirections or HTTP cookies.

You will want to create openers if you want to fetch URLs with specific handlers installed, for example to get an opener that handles cookies, or to get an opener that does not handle redirections.

To create an opener, instantiate an `OpenerDirector`, and then call `.add_handler(`*`some_handler_instance`*`)` repeatedly.

Alternatively, you can use `build_opener()`, which is a convenience function for creating opener *objects* with a single function call. `build_opener()` adds several handlers by default, but provides a quick way to add more and/or override the default handlers.

Other sorts of handlers you might want to use can handle proxies, authentication, and other common but slightly specialized situations.

`install_opener()` can be used to make an opener *object* the (`global`) default opener. This means that calls to `urlopen()` will use the opener you have installed.

Opener *objects* have 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()`, except as a convenience.

### Basic authentication

When authentication is required, the server sends a header (as well as the 401 error code) requesting authentication. This specifies the authentication scheme and a 'realm'. The header looks like: `WWW-Authenticate: SCHEME realm="REALM"`. For example:

```
1   WWW-Authenticate: Basic realm="cPanel Users"
```

*Figure 252: Basic authentication server header*

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 'basic authentication'.

In order to simplify this process, we can create an instance of `HTTPBasicAuthHandler` and an opener to use this handler.

The `HTTPBasicAuthHandler()` uses an *object* called a password manager to handle the mapping of URLs and realms to passwords and usernames. If you know what the realm is (from the authentication header sent by the server), then you can use a `HTTPPasswordMgr()`.

Frequently one doesn't care what the realm is. In that case, it is convenient to use `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.

```
#create a password manager
password_mgr = urllib.request.HTTPPasswordMgrWithDefaultRealm()

#Add the username and password.
#If we knew the realm, we could use it instead of None.
top_level_url = "http://example.com/foo/"
password_mgr.add_password(None, top_level_url, username, password)

handler = urllib.request.HTTPBasicAuthHandler(password_mgr)

#create "opener" (OpenerDirector instance)
opener = urllib.request.build_opener(handler)

#use the opener to fetch a URL
opener.open(a_url)

#Install the opener.
#Now all calls to urllib.request.urlopen use our opener.
urllib.request.install_opener(opener)
```

*Figure 253: Basic authentication example*

In the above example, we only supplied our `HTTPBasicAuthHandler()` to `build_opener()`.

By default openers have the handlers for normal situations – `ProxyHandler` (if a proxy setting such as an `http_proxy` environment variable is set), `UnknownHandler`, `HTTPHandler`, `HTTPDefaultErrorHandler`, `HTTPRedirectHandler`, `FTPHandler`, `FileHandler`, `DataHandler`, `HTTPErrorProcessor`.

*`top_level_url`* is in fact either a full URL including the `http:` scheme component and the hostname and optionally the port number (for example, *`http://example.com/`*) or an "authority" such as the hostname, optionally including the port number (for example, *`example.com`* or *`example.com:8080`* which includes a port number).

The authority, if present, must not contain the *`userinfo`* component - for example *`joe:password@example.com`* is not correct.

### Proxies

**`urllib`** will auto-detect your proxy settings and use those. This is through the `ProxyHandler`, which is part of the normal handler chain when a proxy setting is detected.

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 setting up a Basic Authentication handler:

```
proxy_support = urllib.request.ProxyHandler({})
opener = urllib.request.build_opener(proxy_support)
urllib.request.install_opener(opener)
```

*Figure 254: ProxyHandler set up*

Currently **`urllib.request`** does not support fetching of `https` locations through a proxy. However, this can be enabled by extending **`urllib.request`** as shown in the recipe.

Note: `HTTP_PROXY` will be ignored if a variable `REQUEST_METHOD` is set.

### .getproxies()

**`urllib.request`**`.getproxies()`

This helper function returns a dictionary of scheme to proxy server URL mappings. It scans the environment for variables named `<scheme>_proxy`, in a case insensitive approach, for all operating systems first, and when it cannot find it, looks for proxy information from Mac OSX System Configuration for Mac OS X and Windows Systems Registry for Windows. If both lowercase and uppercase environment variables exist (and disagree), lowercase is preferred.

If the environment variable `REQUEST_METHOD` is set, which usually indicates your script is running in a CGI environment, the environment variable `HTTP_PROXY` (uppercase `_PROXY`) will be ignored. This is because that variable can be injected by a client using the "Proxy:" HTTP header. If you need to use an HTTP proxy in a CGI environment, either use `ProxyHandler` explicitly, or make sure the variable name is in lowercase (or at least the `_proxy` suffix).

## Sockets and layers

The Python support for fetching resources from the web is layered. **`urllib`** uses the **`http.client`** module, which in turn uses the socket library.

As of Python 2.3 you can specify how long a socket should wait for a response before timing out. This can be useful in applications which have to fetch web pages.

By default, the socket module has no timeout and can hang. Currently, the socket timeout is not exposed at the **`http.client`** or **`urllib.request`** levels. However, you can set the default timeout globally for all sockets using the following code:

```
import socket
import urllib.request

#timeout in seconds
timeout = 10
socket.setdefaulttimeout(timeout)

#this call to urllib.request.urlopen now uses the default timeout \
#we have set in the socket module
req = urllib.request.Request('https://docs.python.org')
response = urllib.request.urlopen(req)
```

*Figure 255: Set default global timeout for sockets*

## Notes and specific information

This final section aggregates mostly tables for conversions, such as the JSON conversion table, and specifications regarding certain subjects, such as the built-in exceptions in Python.

### 1. Types of % substitution

| **Conversion** | **Meaning** |
|---|---|
| `%d` | Integer decimal. |
| `%i` | Integer decimal. |
| `%o` | Octal value. |
| `%u` | Obsolete.<br>Identical to `%d`. |
| `%x` | Hexadecimal (lowercase). |
| `%X` | Hexadecimal (uppercase). |
| `%e` | Floating point exponential format (lowercase)[16]. |
| `%E` | Floating point exponential format (uppercase)[17]. |
| `%f` | Floating point decimal format[18]. |
| `%F` | Floating point decimal format[19]. |
| `%g` | Floating point format.<br>Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise[20]. |
| `%G` | Floating point format. |

[16] The alternate form causes the result to always contain a decimal point, even if no digits follow it.
[17] The alternate form causes the result to always contain a decimal point, even if no digits follow it.
[18] The alternate form causes the result to always contain a decimal point, even if no digits follow it.
[19] The alternate form causes the result to always contain a decimal point, even if no digits follow it.
[20] 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.

| | |
|---|---|
| | Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise[21]. |
| `%c` | Single character (accepts integer or single character string). |
| `%r` | String (converts any Python object using `repr()` )[22]. |
| `%s` | String (converts any Python object using `str()` )[23]. |
| `%a` | String (converts any Python object using `ascii()` )[24]. |
| `%` | No argument is converted, results in a '%' character in the result. |

*Table 7: Types of % substitution*

## 2. Division differences (Python 2 vs Python 3)

Between Python 2 and Python 3 there is a big difference in the way divisions are written 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 256: Division differences between Python 2 and Python 3*

[21] 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.
[22] If precision is N, the output is truncated to N characters.
[23] If precision is N, the output is truncated to N characters.
[24] If precision is N, the output is truncated to N characters.

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:

```
1   a = 5 // 2
2   print(a) #outputs 2
```

*Figure 257: floor division in Python 3*

## 3. JSON conversion table

| **Python** | **JSON** |
|---|---|
| dict | object |
| list, tuple | array |
| str | string |
| int, float, int- & flaot-derived Enums | number |
| True | true |
| False | false |
| None | null |

*Table 8: JSON conversion table*

## 4. Python built-in exceptions

| **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: Python built-in exceptions*

## 5. String presentation types

| Type | Meaning |
|---|---|
| `s` | String format.<br>This is the default type for strings and may be omitted. |
| `None` | Same as `s`. |

*Table 10: string presentation types*

## 6. Integer presentation types

| Type | Meaning |
|---|---|
| `b` | Binary format.<br>Outputs the number in base 2. |
| `c` | Character.<br>Converts the integer to the corresponding unicode character before printing. |
| `d` | Decimal Integer.<br>Outputs the number in base 10. |

| | |
|---|---|
| o | Octal format.<br><br>Outputs the number in base 8. |
| x | Hex format.<br><br>Outputs the number in base 16, using lower-case letters for the digits above 9. |
| X | Hex format.<br><br>Outputs the number in base 16, using upper-case letters for the digits above 9. |
| n | Number.<br><br>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*

## 7. Floating point and Decimal presentation types

| Type | Meaning |
|---|---|
| e | Exponent notation.<br><br>Prints the number in scientific notation using the letter ‘e’ to indicate the exponent.<br><br>The default precision is 6. |
| E | Exponent notation.<br><br>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.<br><br>The default precision is 6. |
| `F` | Fixed point.<br><br>Same as '`f`', but converts `nan` to `NAN` and `inf` to `INF`. |
| `g`[25] | General format.<br><br>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.<br><br>Positive and negative infinity, positive and negative zero, and nans, are formatted as `inf`, `-inf`, `0`, `-0` and `nan` respectively, regardless of the precision.<br><br>A precision of 0 is treated as equivalent to a precision of 1. The default precision is 6. |
| `G` | General format.<br>Same as '`g`' except switches to '`E`' if the number gets too large.<br>The representations of infinity and NaN are uppercased, too. |
| `n` | Number.<br>This is the same as '`g`', except that it uses the current locale setting to insert the appropriate number separator characters. |
| `%` | Percentage. |

[25] The precise rules are as follows: suppose that the result formatted with presentation type '`e`' and precision *p*-1 would have exponent *exp*. Then if -4 <= *exp* < *p*, the number is formatted with presentation type '`f`' and precision *p*-1-*exp*. Otherwise, the number is formatted with presentation type '`e`' and precision *p*-1. In both cases insignificant trailing zeros are removed from the significand, and the decimal point is also removed if there are no remaining digits following it.

| | |
|---|---|
| | 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.<br>The default precision is as high as needed to represent the particular value.<br>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*

## 8. HTTP error codes

`http.server.BaseHTTPRequestHandler.responses` is a useful dictionary of response codes in that shows all the response codes used by RFC 2616.

The dictionary is reproduced here for convenience.

Due note, because of its extent, I separated the codes by range (the first picture shows the codes in the 100-299 range, the second shows the codes in the 300s range, the fourth show the codes in the 400s range and the last one shows the codes in the 500s 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 258: HTTP error codes in the 100-299 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 259: HTTP error codes in the 300s 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 260: HTTP error codes in the 400s 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 261: HTTP error codes in the 500s range*