Python®

Notes for Professionals

700+ pages

of professional hints and tricks

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/PythonBook

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with Python Language

Python 3.x

Version	Release Date
3.7	2017-05-08
3.6	2016-12-23
3.5	2015-09-13
3.4	2014-03-17
3.3	2012-09-29
3.2	2011-02-20
3.1	2009-06-26
3.0	2008-12-03

Python 2.x

Version	Release Date
2.7	2010-07-03
2.6	2008-10-02
2.5	2006-09-19
2.4	2004-11-30
2.3	2003-07-29
2.2	2001-12-21
2.1	2001-04-15
2.0	2000-10-16

Section 1.1: Getting Started

Python is a widely used high-level programming language for general-purpose programming, created by Guido van Rossum and first released in 1991. Python features a dynamic type system and automatic memory management and supports multiple programming paradigms, including object-oriented, imperative, functional programming, and procedural styles. It has a large and comprehensive standard library.

Two major versions of Python are currently in active use:

- Python 3.x is the current version and is under active development.
- Python 2.x is the legacy version and will receive only security updates until 2020. No new features will be implemented. Note that many projects still use Python 2, although migrating to Python 3 is getting easier.

You can download and install either version of Python here. See Python 3 vs. Python 2 for a comparison between them. In addition, some third-parties offer re-packaged versions of Python that add commonly used libraries and other features to ease setup for common use cases, such as math, data analysis or scientific use. See the list at the official site.

Verify if Python is installed

To confirm that Python was installed correctly, you can verify that by running the following command in your favorite terminal (If you are using Windows OS, you need to add path of python to the environment variable before using it in command prompt):

```
$ python --version
```

Python 3.x `Version ≥ 3.0`

If you have *Python 3* installed, and it is your default version (see **Troubleshooting** for more details) you should see something like this:

```
$ python --version
Python 3.6.0
```

Python 2.x `Version ≤ 2.7`

If you have *Python 2* installed, and it is your default version (see **Troubleshooting** for more details) you should see something like this:

```
$ python --version
Python 2.7.13
```

If you have installed Python 3, but `$ python --version` outputs a Python 2 version, you also have Python 2 installed. This is often the case on MacOS, and many Linux distributions. Use `$ python3` instead to explicitly use the Python 3 interpreter.

Hello, World in Python using IDLE

IDLE is a simple editor for Python, that comes bundled with Python.

How to create Hello, World program in IDLE

- Open IDLE on your system of choice.
  - In older versions of Windows, it can be found at `All Programs` under the Windows menu.
  - In Windows 8+, search for `IDLE` or find it in the apps that are present in your system.
  - On Unix-based (including Mac) systems you can open it from the shell by typing `$ idle python_file.py`.
- It will open a shell with options along the top.

In the shell, there is a prompt of three right angle brackets:

```
>>>
```

Now write the following code in the prompt:

```
>>> print("Hello, World")
```

Hit `Enter`.

```
>>> print("Hello, World")
Hello, World
```

Hello World Python file

Create a new file `hello.py` that contains the following line:

Python 3.x `Version ≥ 3.0`

```
print('Hello, World')
```

Python 2.x `Version ≥ 2.6`

You can use the Python 3 `print` function in Python 2 with the following `import` statement:

```
from __future__ import print_function
```

Python 2 has a number of functionalities that can be optionally imported from Python 3 using the `__future__` module, as discussed here.

Python 2.x `Version ≤ 2.7`

If using Python 2, you may also type the line below. Note that this is not valid in Python 3 and thus not recommended because it reduces cross-version code compatibility.

```
print 'Hello, World'
```

In your terminal, navigate to the directory containing the file `hello.py`.

Type `python hello.py`, then hit the `Enter` key.

```
$ python hello.py
Hello, World
```

You should see `Hello, World` printed to the console.

You can also substitute `hello.py` with the path to your file. For example, if you have the file in your home directory and your user is "user" on Linux, you can type `python /home/user/hello.py`.

Launch an interactive Python shell

By executing (running) the `python` command in your terminal, you are presented with an interactive Python shell. This is also known as the Python Interpreter or a REPL (for 'Read Evaluate Print Loop').

```
$ python
Python 2.7.12 (default, Jun 28 2016, 08:46:01)
[GCC 6.1.1 20160602] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'Hello, World'
Hello, World
>>>
```

If you want to run Python 3 from your terminal, execute the command `python3`.

```
$ python3
Python 3.6.0 (default, Jan 13 2017, 00:00:00)
[GCC 6.1.1 20160602] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello, World')
Hello, World
>>>
```

Alternatively, start the interactive prompt and load file with `python -i <file.py>`.

In command line, run:

```
$ python -i hello.py
"Hello World"
>>>
```

There are multiple ways to close the Python shell:

```
>>> exit()
```

or

```
>>> quit()
```

Alternatively, `CTRL + D` will close the shell and put you back on your terminal's command line.

If you want to cancel a command you're in the middle of typing and get back to a clean command prompt, while staying inside the Interpreter shell, use `CTRL + C`.

Try an interactive Python shell online.

Other Online Shells

Various websites provide online access to Python shells.

Online shells may be useful for the following purposes:

- Run a small code snippet from a machine which lacks python installation(smartphones, tablets etc).
- Learn or teach basic Python.
- Solve online judge problems.

Examples:

> Disclaimer: documentation author(s) are not affiliated with any resources listed below.

- https://www.python.org/shell/ - The online Python shell hosted by the official Python website.
- https://ideone.com/ - Widely used on the Net to illustrate code snippet behavior.
- https://repl.it/languages/python3 - Powerful and simple online compiler, IDE and interpreter. Code, compile, and run code in Python.
- https://www.tutorialspoint.com/execute_python_online.php - Full-featured UNIX shell, and a user-friendly project explorer.
- http://rextester.com/l/python3_online_compiler - Simple and easy to use IDE which shows execution time

Run commands as a string

Python can be passed arbitrary code as a string in the shell:

```
$ python -c 'print("Hello, World")'
Hello, World
```

This can be useful when concatenating the results of scripts together in the shell.

Shells and Beyond

Package Management - The PyPA recommended tool for installing Python packages is PIP. To install, on your command line execute `pip install <the package name>`. For instance, `pip install numpy`. (Note: On windows you must add pip to your PATH environment variables. To avoid this, use `python -m pip install <the package name>`)

Shells - So far, we have discussed different ways to run code using Python's native interactive shell. Shells use Python's interpretive power for experimenting with code real-time. Alternative shells include IDLE - a pre-bundled

GUI, IPython - known for extending the interactive experience, etc.

Programs - For long-term storage you can save content to .py files and edit/execute them as scripts or programs with external tools e.g. shell, IDEs (such as PyCharm), Jupyter notebooks, etc. Intermediate users may use these tools; however, the methods discussed here are sufficient for getting started.

Python tutor allows you to step through Python code so you can visualize how the program will flow, and helps you to understand where your program went wrong.

PEP8 defines guidelines for formatting Python code. Formatting code well is important so you can quickly read what the code does.

Section 1.2: Creating variables and assigning values

To create a variable in Python, all you need to do is specify the variable name, and then assign a value to it.

```
<variable name> = <value>
```

Python uses `=` to assign values to variables. There's no need to declare a variable in advance (or to assign a data type to it), assigning a value to a variable itself declares and initializes the variable with that value. There's no way to declare a variable without assigning it an initial value.

```
# Integer
a = 2
print(a)
# Output: 2

# Integer
b = 9223372036854775807
print(b)
# Output: 9223372036854775807

# Floating point
pi = 3.14
print(pi)
# Output: 3.14

# String
c = 'A'
print(c)
# Output: A

# String
name = 'John Doe'
print(name)
# Output: John Doe

# Boolean
q = True
print(q)
# Output: True

# Empty value or null data type
x = None
print(x)
# Output: None
```

Variable assignment works from left to right. So the following will give you an syntax error.

```
0 = x
=> Output: SyntaxError: can't assign to literal
```

You can not use python's keywords as a valid variable name. You can see the list of keyword by:

```
import keyword
print(keyword.kwlist)
```

Rules for variable naming:

1. Variables names must start with a letter or an underscore.

```
x  = True   # valid
_y = True   # valid

9x = False  # starts with numeral
=> SyntaxError: invalid syntax

$y = False #  starts with symbol
=> SyntaxError: invalid syntax
```

2. The remainder of your variable name may consist of letters, numbers and underscores.

```
has_0_in_it = "Still Valid"
```

3. Names are case sensitive.

```
x = 9
y = X*5
=>NameError: name 'X' is not defined
```

Even though there's no need to specify a data type when declaring a variable in Python, while allocating the necessary area in memory for the variable, the Python interpreter automatically picks the most suitable built-in type for it:

```
a = 2
print(type(a))
# Output: <type 'int'>

b = 9223372036854775807
print(type(b))
# Output: <type 'int'>

pi = 3.14
print(type(pi))
# Output: <type 'float'>

c = 'A'
print(type(c))
# Output: <type 'str'>

name = 'John Doe'
print(type(name))
# Output: <type 'str'>

q = True
print(type(q))
# Output: <type 'bool'>
```

```
x = None
print(type(x))
# Output: <type 'NoneType'>
```

Now you know the basics of assignment, let's get this subtlety about assignment in python out of the way.

When you use `=` to do an assignment operation, what's on the left of `=` is a ***name*** for the ***object*** on the right. Finally, what `=` does is assign the ***reference*** of the object on the right to the ***name*** on the left.

That is:

```
a_name = an_object  # "a_name" is now a name for the reference to the object "an_object"
```

So, from many assignment examples above, if we pick `pi = 3.14`, then `pi` is **a** name (not **the** name, since an object can have multiple names) for the object `3.14`. If you don't understand something below, come back to this point and read this again! Also, you can take a look at this for a better understanding.

You can assign multiple values to multiple variables in one line. Note that there must be the same number of arguments on the right and left sides of the `=` operator:

```
a, b, c = 1, 2, 3
print(a, b, c)
# Output: 1 2 3

a, b, c = 1, 2
=> Traceback (most recent call last):
=>   File "name.py", line N, in <module>
=>     a, b, c = 1, 2
=> ValueError: need more than 2 values to unpack

a, b = 1, 2, 3
=> Traceback (most recent call last):
=>   File "name.py", line N, in <module>
=>     a, b = 1, 2, 3
=> ValueError: too many values to unpack
```

The error in last example can be obviated by assigning remaining values to equal number of arbitrary variables. This dummy variable can have any name, but it is conventional to use the underscore (_) for assigning unwanted values:

```
a, b, _ = 1, 2, 3
print(a, b)
# Output: 1, 2
```

Note that the number of _ and number of remaining values must be equal. Otherwise 'too many values to unpack error' is thrown as above:

```
a, b, _ = 1,2,3,4
=>Traceback (most recent call last):
=>File "name.py", line N, in <module>
=>a, b, _ = 1,2,3,4
=>ValueError: too many values to unpack (expected 3)
```

You can also assign a single value to several variables simultaneously.

```
a = b = c = 1
print(a, b, c)
```

```
# Output: 1 1 1
```

When using such cascading assignment, it is important to note that all three variables `a`, `b` and `c` refer to the *same object* in memory, an `int` object with the value of 1. In other words, `a`, `b` and `c` are three different names given to the same int object. Assigning a different object to one of them afterwards doesn't change the others, just as expected:

```
a = b = c = 1    # all three names a, b and c refer to same int object with value 1
print(a, b, c)
# Output: 1 1 1
b = 2            # b now refers to another int object, one with a value of 2
print(a, b, c)
# Output: 1 2 1  # so output is as expected.
```

The above is also true for mutable types (like `list`, `dict`, etc.) just as it is true for immutable types (like `int`, `string`, `tuple`, etc.):

```
x = y = [7, 8, 9]   # x and y refer to the same list object just created, [7, 8, 9]
x = [13, 8, 9]      # x now refers to a different list object just created, [13, 8, 9]
print(y)            # y still refers to the list it was first assigned
# Output: [7, 8, 9]
```

So far so good. Things are a bit different when it comes to *modifying* the object (in contrast to *assigning* the name to a different object, which we did above) when the cascading assignment is used for mutable types. Take a look below, and you will see it first hand:

```
x = y = [7, 8, 9]     # x and y are two different names for the same list object just created, [7,
8, 9]
x[0] = 13             # we are updating the value of the list [7, 8, 9] through one of its names, x
in this case
print(y)              # printing the value of the list using its other name
# Output: [13, 8, 9]  # hence, naturally the change is reflected
```

Nested lists are also valid in python. This means that a list can contain another list as an element.

```
x = [1, 2, [3, 4, 5], 6, 7] # this is nested list
print x[2]
# Output: [3, 4, 5]
print x[2][1]
# Output: 4
```

Lastly, variables in Python do not have to stay the same type as which they were first defined -- you can simply use `=` to assign a new value to a variable, even if that value is of a different type.

```
a = 2
print(a)
# Output: 2

a = "New value"
print(a)
# Output: New value
```

If this bothers you, think about the fact that what's on the left of `=` is just a name for an object. First you call the `int` object with value 2 `a`, then you change your mind and decide to give the name `a` to a `string` object, having value 'New value'. Simple, right?

Section 1.3: Block Indentation

Python uses indentation to define control and loop constructs. This contributes to Python's readability, however, it requires the programmer to pay close attention to the use of whitespace. Thus, editor miscalibration could result in code that behaves in unexpected ways.

Python uses the colon symbol (`:`) and indentation for showing where blocks of code begin and end (If you come from another language, do not confuse this with somehow being related to the ternary operator). That is, blocks in Python, such as functions, loops, `if` clauses and other constructs, have no ending identifiers. All blocks start with a colon and then contain the indented lines below it.

For example:

```
def my_function():    # This is a function definition. Note the colon (:)
    a = 2             # This line belongs to the function because it's indented
    return a          # This line also belongs to the same function
print(my_function())  # This line is OUTSIDE the function block
```

or

```
if a > b:             # If block starts here
    print(a)          # This is part of the if block
else:                 # else must be at the same level as if
    print(b)          # This line is part of the else block
```

Blocks that contain exactly one single-line statement may be put on the same line, though this form is generally not considered good style:

```
if a > b: print(a)
else: print(b)
```

Attempting to do this with more than a single statement will *not* work:

```
if x > y: y = x
    print(y) # IndentationError: unexpected indent

if x > y: while y != z: y -= 1  # SyntaxError: invalid syntax
```

An empty block causes an `IndentationError`. Use `pass` (a command that does nothing) when you have a block with no content:

```
def will_be_implemented_later():
    pass
```

Spaces vs. Tabs

In short: **always** use 4 spaces for indentation.

Using tabs exclusively is possible but PEP 8, the style guide for Python code, states that spaces are preferred.

Python 3.x `Version ≥ 3.0`

Python 3 disallows mixing the use of tabs and spaces for indentation. In such case a compile-time error is generated: `Inconsistent use of tabs and spaces in indentation` and the program will not run.

Python 2.x `Version ≤ 2.7`

Python 2 allows mixing tabs and spaces in indentation; this is strongly discouraged. The tab character completes the previous indentation to be a multiple of **8** spaces. Since it is common that editors are configured to show tabs as multiple of **4** spaces, this can cause subtle bugs.

Citing PEP 8:

> When invoking the Python 2 command line interpreter with the `-t` option, it issues warnings about code that illegally mixes tabs and spaces. When using `-tt` these warnings become errors. These options are highly recommended!

Many editors have "tabs to spaces" configuration. When configuring the editor, one should differentiate between the tab *character* ('\t') and the Tab key.

- The tab *character* should be configured to show 8 spaces, to match the language semantics - at least in cases when (accidental) mixed indentation is possible. Editors can also automatically convert the tab character to spaces.
- However, it might be helpful to configure the editor so that pressing the Tab key will insert 4 spaces, instead of inserting a tab character.

Python source code written with a mix of tabs and spaces, or with non-standard number of indentation spaces can be made pep8-conformant using autopep8. (A less powerful alternative comes with most Python installations: reindent.py)

Section 1.4: Datatypes

Built-in Types
Booleans

`bool`: A boolean value of either `True` or `False`. Logical operations like `and`, `or`, `not` can be performed on booleans.

```
x or y    # if x is False then y otherwise x
x and y   # if x is False then x otherwise y
not x     # if x is True then False, otherwise True
```

In Python 2.x and in Python 3.x, a boolean is also an `int`. The `bool` type is a subclass of the `int` type and `True` and `False` are its only instances:

```
issubclass(bool, int) # True

isinstance(True, bool) # True
isinstance(False, bool) # True
```

If boolean values are used in arithmetic operations, their integer values (`1` and `0` for `True` and `False`) will be used to return an integer result:

```
True + False == 1 # 1 + 0 == 1
True * True  == 1 # 1 * 1 == 1
```

Numbers

- `int`: Integer number

```
a = 2
b = 100
c = 123456789
d = 38563846326424324
```

Integers in Python are of arbitrary sizes.

Note: in older versions of Python, a `long` type was available and this was distinct from `int`. The two have been unified.

- `float`: Floating point number; precision depends on the implementation and system architecture, for CPython the `float` datatype corresponds to a C double.

```
a = 2.0
b = 100.e0
c = 123456789.e1
```

- `complex`: Complex numbers

```
a = 2 + 1j
b = 100 + 10j
```

The `<`, `<=`, `>` and `>=` operators will raise a `TypeError` exception when any operand is a complex number.

Strings

Python 3.x `Version ≥ 3.0`

- `str`: a **unicode string**. The type of `'hello'`
- `bytes`: a **byte string**. The type of `b'hello'`

Python 2.x `Version ≤ 2.7`

- `str`: a **byte string**. The type of `'hello'`
- `bytes`: synonym for `str`
- `unicode`: a **unicode string**. The type of `u'hello'`

Sequences and collections

Python differentiates between ordered sequences and unordered collections (such as `set` and `dict`).

- strings (`str`, `bytes`, `unicode`) are sequences
- `reversed`: A reversed order of `str` with `reversed` function

```
a = reversed('hello')
```

- `tuple`: An ordered collection of `n` values of any type (`n >= 0`).

```
a = (1, 2, 3)
b = ('a', 1, 'python', (1, 2))
b[2] = 'something else' # returns a TypeError
```

Supports indexing; immutable; hashable if all its members are hashable

- `list`: An ordered collection of `n` values (`n >= 0`)

```
a = [1, 2, 3]
b = ['a', 1, 'python', (1, 2), [1, 2]]
b[2] = 'something else' # allowed
```

Not hashable; mutable.

- `set`: An unordered collection of unique values. Items must be hashable.

```
a = {1, 2, 'a'}
```

- `dict`: An unordered collection of unique key-value pairs; keys must be hashable.

```
a = {1: 'one',
     2: 'two'}

b = {'a': [1, 2, 3],
     'b': 'a string'}
```

> 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 equality must have the same hash value.

Built-in constants

In conjunction with the built-in datatypes there are a small number of built-in constants in the built-in namespace:

- `True`: The true value of the built-in type `bool`
- `False`: The false value of the built-in type `bool`
- `None`: A singleton object used to signal that a value is absent.
- `Ellipsis` or `...`: used in core Python3+ anywhere and limited usage in Python2.7+ as part of array notation. `numpy` and related packages use this as a 'include everything' reference in arrays.
- `NotImplemented`: a singleton used to indicate to Python that a special method doesn't support the specific arguments, and Python will try alternatives if available.

```
a = None # No value will be assigned. Any valid datatype can be assigned later
```

Python 3.x `Version ≥ 3.0`

`None` doesn't have any natural ordering. Using ordering comparison operators (`<`, `<=`, `>=`, `>`) isn't supported anymore and will raise a `TypeError`.

Python 2.x `Version ≤ 2.7`

`None` is always less than any number (`None < -32` evaluates to `True`).

Testing the type of variables

In python, we can check the datatype of an object using the built-in function `type`.

```
a = '123'
print(type(a))
```

```
# Out: <class 'str'>
b = 123
print(type(b))
# Out: <class 'int'>
```

In conditional statements it is possible to test the datatype with `isinstance`. However, it is usually not encouraged to rely on the type of the variable.

```
i = 7
if isinstance(i, int):
    i += 1
elif isinstance(i, str):
    i = int(i)
    i += 1
```

For information on the differences between `type()` and `isinstance()` read: Differences between isinstance and type in Python

To test if something is of `NoneType`:

```
x = None
if x is None:
    print('Not a surprise, I just defined x as None.')
```

Converting between datatypes

You can perform explicit datatype conversion.

For example, '123' is of `str` type and it can be converted to integer using `int` function.

```
a = '123'
b = int(a)
```

Converting from a float string such as '123.456' can be done using `float` function.

```
a = '123.456'
b = float(a)
c = int(a)    # ValueError: invalid literal for int() with base 10: '123.456'
d = int(b)    # 123
```

You can also convert sequence or collection types

```
a = 'hello'
list(a)  # ['h', 'e', 'l', 'l', 'o']
set(a)   # {'o', 'e', 'l', 'h'}
tuple(a) # ('h', 'e', 'l', 'l', 'o')
```

Explicit string type at definition of literals

With one letter labels just in front of the quotes you can tell what type of string you want to define.

- `b'foo bar'`: results `bytes` in Python 3, `str` in Python 2
- `u'foo bar'`: results `str` in Python 3, `unicode` in Python 2
- `'foo bar'`: results `str`
- `r'foo bar'`: results so called raw string, where escaping special characters is not necessary, everything is taken verbatim as you typed

```
normal  = 'foo\nbar'   # foo
```

```
                        # bar
escaped = 'foo\\nbar'   # foo\nbar
raw     = r'foo\nbar'   # foo\nbar
```

Mutable and Immutable Data Types

An object is called *mutable* if it can be changed. For example, when you pass a list to some function, the list can be changed:

```
def f(m):
    m.append(3)  # adds a number to the list. This is a mutation.

x = [1, 2]
f(x)
x == [1, 2]  # False now, since an item was added to the list
```

An object is called *immutable* if it cannot be changed in any way. For example, integers are immutable, since there's no way to change them:

```
def bar():
    x = (1, 2)
    g(x)
    x == (1, 2)  # Will always be True, since no function can change the object (1, 2)
```

Note that **variables** themselves are mutable, so we can reassign the *variable* `x`, but this does not change the object that `x` had previously pointed to. It only made `x` point to a new object.

Data types whose instances are mutable are called *mutable data types*, and similarly for immutable objects and datatypes.

Examples of immutable Data Types:

- `int`, `long`, `float`, `complex`
- `str`
- `bytes`
- `tuple`
- `frozenset`

Examples of mutable Data Types:

- `bytearray`
- `list`
- `set`
- `dict`

Section 1.5: Collection Types

There are a number of collection types in Python. While types such as `int` and `str` hold a single value, collection types hold multiple values.

Lists

The `list` type is probably the most commonly used collection type in Python. Despite its name, a list is more like an array in other languages, mostly JavaScript. In Python, a list is merely an ordered collection of valid Python values. A list can be created by enclosing values, separated by commas, in square brackets:

```
int_list = [1, 2, 3]
string_list = ['abc', 'defghi']
```

A list can be empty:

```
empty_list = []
```

The elements of a list are not restricted to a single data type, which makes sense given that Python is a dynamic language:

```
mixed_list = [1, 'abc', True, 2.34, None]
```

A list can contain another list as its element:

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

The elements of a list can be accessed via an *index*, or numeric representation of their position. Lists in Python are *zero-indexed* meaning that the first element in the list is at index 0, the second element is at index 1 and so on:

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
print(names[0]) # Alice
print(names[2]) # Craig
```

Indices can also be negative which means counting from the end of the list (`-1` being the index of the last element). So, using the list from the above example:

```
print(names[-1]) # Eric
print(names[-4]) # Bob
```

Lists are mutable, so you can change the values in a list:

```
names[0] = 'Ann'
print(names)
# Outputs ['Ann', 'Bob', 'Craig', 'Diana', 'Eric']
```

Besides, it is possible to add and/or remove elements from a list:

Append object to end of list with `L.append(object)`, returns `None`.

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
names.append("Sia")
print(names)
# Outputs ['Alice', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
```

Add a new element to list at a specific index. `L.insert(index, object)`

```
names.insert(1, "Nikki")
print(names)
# Outputs ['Alice', 'Nikki', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
```

Remove the first occurrence of a value with `L.remove(value)`, returns `None`

```
names.remove("Bob")
print(names) # Outputs ['Alice', 'Nikki', 'Craig', 'Diana', 'Eric', 'Sia']
```

Get the index in the list of the first item whose value is x. It will show an error if there is no such item.

```
name.index("Alice")
0
```

Count length of list

```
len(names)
6
```

count occurrence of any item in list

```
a = [1, 1, 1, 2, 3, 4]
a.count(1)
3
```

Reverse the list

```
a.reverse()
[4, 3, 2, 1, 1, 1]
# or
a[::-1]
[4, 3, 2, 1, 1, 1]
```

Remove and return item at index (defaults to the last item) with `L.pop([index])`, returns the item

```
names.pop() # Outputs 'Sia'
```

You can iterate over the list elements like below:

```
for element in my_list:
    print (element)
```

Tuples

A `tuple` is similar to a list except that it is fixed-length and immutable. So the values in the tuple cannot be changed nor the values be added to or removed from the tuple. Tuples are commonly used for small collections of values that will not need to change, such as an IP address and port. Tuples are represented with parentheses instead of square brackets:

```
ip_address = ('10.20.30.40', 8080)
```

The same indexing rules for lists also apply to tuples. Tuples can also be nested and the values can be any valid Python valid.

A tuple with only one member must be defined (note the comma) this way:

```
one_member_tuple = ('Only member',)
```

or

```
one_member_tuple = 'Only member',    # No brackets
```

or just using `tuple` syntax

```
one_member_tuple = tuple(['Only member'])
```

Dictionaries

A `dictionary` in Python is a collection of key-value pairs. The dictionary is surrounded by curly braces. Each pair is separated by a comma and the key and value are separated by a colon. Here is an example:

```
state_capitals = {
    'Arkansas': 'Little Rock',
    'Colorado': 'Denver',
    'California': 'Sacramento',
    'Georgia': 'Atlanta'
}
```

To get a value, refer to it by its key:

```
ca_capital = state_capitals['California']
```

You can also get all of the keys in a dictionary and then iterate over them:

```
for k in state_capitals.keys():
    print('{} is the capital of {}'.format(state_capitals[k], k))
```

Dictionaries strongly resemble JSON syntax. The native `json` module in the Python standard library can be used to convert between JSON and dictionaries.

set

A `set` is a collection of elements with no repeats and without insertion order but sorted order. They are used in situations where it is only important that some things are grouped together, and not what order they were included. For large groups of data, it is much faster to check whether or not an element is in a `set` than it is to do the same for a `list`.

Defining a `set` is very similar to defining a `dictionary`:

```
first_names = {'Adam', 'Beth', 'Charlie'}
```

Or you can build a `set` using an existing `list`:

```
my_list = [1,2,3]
my_set = set(my_list)
```

Check membership of the `set` using `in`:

```
if name in first_names:
    print(name)
```

You can iterate over a `set` exactly like a list, but remember: the values will be in an arbitrary, implementation-defined order.

defaultdict

A `defaultdict` is a dictionary with a default value for keys, so that keys for which no value has been explicitly defined can be accessed without errors. `defaultdict` is especially useful when the values in the dictionary are collections (lists, dicts, etc) in the sense that it does not need to be initialized every time when a new key is used.

A `defaultdict` will never raise a KeyError. Any key that does not exist gets the default value returned.

For example, consider the following dictionary

```
>>> state_capitals = {
    'Arkansas': 'Little Rock',
    'Colorado': 'Denver',
    'California': 'Sacramento',
    'Georgia': 'Atlanta'
}
```

If we try to access a non-existent key, python returns us an error as follows

```
>>> state_capitals['Alabama']
Traceback (most recent call last):

  File "<ipython-input-61-236329695e6f>", line 1, in <module>
    state_capitals['Alabama']

KeyError: 'Alabama'
```

Let us try with a `defaultdict`. It can be found in the collections module.

```
>>> from collections import defaultdict
>>> state_capitals = defaultdict(lambda: 'Boston')
```

What we did here is to set a default value (**Boston**) in case the give key does not exist. Now populate the dict as before:

```
>>> state_capitals['Arkansas'] = 'Little Rock'
>>> state_capitals['California'] = 'Sacramento'
>>> state_capitals['Colorado'] = 'Denver'
>>> state_capitals['Georgia'] = 'Atlanta'
```

If we try to access the dict with a non-existent key, python will return us the default value i.e. Boston

```
>>> state_capitals['Alabama']
'Boston'
```

and returns the created values for existing key just like a normal `dictionary`

```
>>> state_capitals['Arkansas']
'Little Rock'
```

Section 1.6: IDLE - Python GUI

IDLE is Python's Integrated Development and Learning Environment and is an alternative to the command line. As the name may imply, IDLE is very useful for developing new code or learning python. On Windows this comes with the Python interpreter, but in other operating systems you may need to install it through your package manager.

The main purposes of IDLE are:

- Multi-window text editor with syntax highlighting, autocompletion, and smart indent
- Python shell with syntax highlighting
- Integrated debugger with stepping, persistent breakpoints, and call stack visibility
- Automatic indentation (useful for beginners learning about Python's indentation)

- Saving the Python program as .py files and run them and edit them later at any them using IDLE.

In IDLE, hit `F5` or `run Python Shell` to launch an interpreter. Using IDLE can be a better learning experience for new users because code is interpreted as the user writes.

Note that there are lots of alternatives, see for example this discussion or this list.

Troubleshooting

- **Windows**

  If you're on Windows, the default command is `python`. If you receive a `"'python' is not recognized"` error, the most likely cause is that Python's location is not in your system's `PATH` environment variable. This can be accessed by right-clicking on 'My Computer' and selecting 'Properties' or by navigating to 'System' through 'Control Panel'. Click on 'Advanced system settings' and then 'Environment Variables...'. Edit the `PATH` variable to include the directory of your Python installation, as well as the Script folder (usually `C:\Python27;C:\Python27\Scripts`). This requires administrative privileges and may require a restart.

  When using multiple versions of Python on the same machine, a possible solution is to rename one of the `python.exe` files. For example, naming one version `python27.exe` would cause `python27` to become the Python command for that version.

  You can also use the Python Launcher for Windows, which is available through the installer and comes by default. It allows you to select the version of Python to run by using `py -[x.y]` instead of `python[x.y]`. You can use the latest version of Python 2 by running scripts with `py -2` and the latest version of Python 3 by running scripts with `py -3`.

- **Debian/Ubuntu/MacOS**

  This section assumes that the location of the `python` executable has been added to the `PATH` environment variable.

  If you're on Debian/Ubuntu/MacOS, open the terminal and type `python` for Python 2.x or `python3` for Python 3.x.

  Type `which python` to see which Python interpreter will be used.

- **Arch Linux**

  The default Python on Arch Linux (and descendants) is Python 3, so use `python` or `python3` for Python 3.x and `python2` for Python 2.x.

- **Other systems**

  Python 3 is sometimes bound to `python` instead of `python3`. To use Python 2 on these systems where it is installed, you can use `python2`.

Section 1.7: User Input

Interactive input

To get input from the user, use the `input` function (**note**: in Python 2.x, the function is called `raw_input` instead, although Python 2.x has its own version of `input` that is completely different):

Python 2.x `Version ≥ 2.3`

```
name = raw_input("What is your name? ")
# Out: What is your name? _
```

> **Security Remark** Do not use `input()` in Python2 - the entered text will be evaluated as if it were a Python expression (equivalent to `eval(input())` in Python3), which might easily become a vulnerability. See this article for further information on the risks of using this function.

Python 3.x `Version ≥ 3.0`

```
name = input("What is your name? ")
# Out: What is your name? _
```

The remainder of this example will be using Python 3 syntax.

The function takes a string argument, which displays it as a prompt and returns a string. The above code provides a prompt, waiting for the user to input.

```
name = input("What is your name? ")
# Out: What is your name?
```

If the user types "Bob" and hits enter, the variable `name` will be assigned to the string `"Bob"`:

```
name = input("What is your name? ")
# Out: What is your name? Bob
print(name)
# Out: Bob
```

Note that the `input` is always of type `str`, which is important if you want the user to enter numbers. Therefore, you need to convert the `str` before trying to use it as a number:

```
x = input("Write a number:")
# Out: Write a number: 10
x / 2
# Out: TypeError: unsupported operand type(s) for /: 'str' and 'int'
float(x) / 2
# Out: 5.0
```

NB: It's recommended to use `try`/`except` blocks to catch exceptions when dealing with user inputs. For instance, if your code wants to cast a `raw_input` into an `int`, and what the user writes is uncastable, it raises a `ValueError`.

Section 1.8: Built in Modules and Functions

A module is a file containing Python definitions and statements. Function is a piece of code which execute some logic.

```
>>> pow(2,3)     #8
```

To check the built in function in python we can use `dir()`. If called without an argument, return the names in the current scope. Else, return an alphabetized list of names comprising (some of) the attribute of the given object, and of attributes reachable from it.

```
>>> dir(__builtins__)
[
    'ArithmeticError',
    'AssertionError',
    'AttributeError',
    'BaseException',
    'BufferError',
    'BytesWarning',
    'DeprecationWarning',
    'EOFError',
    'Ellipsis',
    'EnvironmentError',
    'Exception',
    'False',
    'FloatingPointError',
    'FutureWarning',
    'GeneratorExit',
    'IOError',
    'ImportError',
    'ImportWarning',
    'IndentationError',
    'IndexError',
    'KeyError',
    'KeyboardInterrupt',
    'LookupError',
    'MemoryError',
    'NameError',
    'None',
    'NotImplemented',
    'NotImplementedError',
    'OSError',
    'OverflowError',
    'PendingDeprecationWarning',
    'ReferenceError',
    'RuntimeError',
    'RuntimeWarning',
    'StandardError',
    'StopIteration',
    'SyntaxError',
    'SyntaxWarning',
    'SystemError',
    'SystemExit',
    'TabError',
    'True',
    'TypeError',
    'UnboundLocalError',
    'UnicodeDecodeError',
    'UnicodeEncodeError',
    'UnicodeError',
    'UnicodeTranslateError',
    'UnicodeWarning',
    'UserWarning',
    'ValueError',
    'Warning',
    'ZeroDivisionError',
    '__debug__',
    '__doc__',
```

```
    '__import__',
    '__name__',
    '__package__',
    'abs',
    'all',
    'any',
    'apply',
    'basestring',
    'bin',
    'bool',
    'buffer',
    'bytearray',
    'bytes',
    'callable',
    'chr',
    'classmethod',
    'cmp',
    'coerce',
    'compile',
    'complex',
    'copyright',
    'credits',
    'delattr',
    'dict',
    'dir',
    'divmod',
    'enumerate',
    'eval',
    'execfile',
    'exit',
    'file',
    'filter',
    'float',
    'format',
    'frozenset',
    'getattr',
    'globals',
    'hasattr',
    'hash',
    'help',
    'hex',
    'id',
    'input',
    'int',
    'intern',
    'isinstance',
    'issubclass',
    'iter',
    'len',
    'license',
    'list',
    'locals',
    'long',
    'map',
    'max',
    'memoryview',
    'min',
    'next',
    'object',
    'oct',
    'open',
    'ord',
```

```
    'pow',
    'print',
    'property',
    'quit',
    'range',
    'raw_input',
    'reduce',
    'reload',
    'repr',
    'reversed',
    'round',
    'set',
    'setattr',
    'slice',
    'sorted',
    'staticmethod',
    'str',
    'sum',
    'super',
    'tuple',
    'type',
    'unichr',
    'unicode',
    'vars',
    'xrange',
    'zip'
]
```

To know the functionality of any function, we can use built in function `help` .

```
>>> help(max)
Help on built-in function max in module __builtin__:
max(...)
    max(iterable[, key=func]) -> value
    max(a, b, c, ...[, key=func]) -> value
    With a single iterable argument, return its largest item.
    With two or more arguments, return the largest argument.
```

Built in modules contains extra functionalities. For example to get square root of a number we need to include `math` module.

```
>>> import math
>>> math.sqrt(16) # 4.0
```

To know all the functions in a module we can assign the functions list to a variable, and then print the variable.

```
>>> import math
>>> dir(math)

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

it seems `__doc__` is useful to provide some documentation in, say, functions

```
>>> math.__doc__
'This module is always available.  It provides access to the\nmathematical
 functions defined by the C standard.'
```

In addition to functions, documentation can also be provided in modules. So, if you have a file named `helloWorld.py` like this:

```
"""This is the module docstring."""

def sayHello():
    """This is the function docstring."""
    return 'Hello World'
```

You can access its docstrings like this:

```
>>> import helloWorld
>>> helloWorld.__doc__
'This is the module docstring.'
>>> helloWorld.sayHello.__doc__
'This is the function docstring.'
```

- For any user defined type, its attributes, its class's attributes, and recursively the attributes of its class's base classes can be retrieved using dir()

```
>>> class MyClassObject(object):
...     pass
...
>>> dir(MyClassObject)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__',
'__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

Any data type can be simply converted to string using a builtin function called `str`. This function is called by default when a data type is passed to `print`

```
>>> str(123)    # "123"
```

Section 1.9: Creating a module

A module is an importable file containing definitions and statements.

A module can be created by creating a `.py` file.

```
# hello.py
def say_hello():
    print("Hello!")
```

Functions in a module can be used by importing the module.

For modules that you have made, they will need to be in the same directory as the file that you are importing them into. (However, you can also put them into the Python lib directory with the pre-included modules, but should be avoided if possible.)

```
$ python
>>> import hello
>>> hello.say_hello()
```

```
=> "Hello!"
```

Modules can be imported by other modules.

```
# greet.py
import hello
hello.say_hello()
```

Specific functions of a module can be imported.

```
# greet.py
from hello import say_hello
say_hello()
```

Modules can be aliased.

```
# greet.py
import hello as ai
ai.say_hello()
```

A module can be stand-alone runnable script.

```
# run_hello.py
if __name__ == '__main__':
    from hello import say_hello
    say_hello()
```

Run it!

```
$ python run_hello.py
=> "Hello!"
```

If the module is inside a directory and needs to be detected by python, the directory should contain a file named `__init__.py`.

Section 1.10: Installation of Python 2.7.x and 3.x

> **Note**: Following instructions are written for Python 2.7 (unless specified): instructions for Python 3.x are similar.

Windows

First, download the latest version of Python 2.7 from the official Website (https://www.python.org/downloads/). Version is provided as an MSI package. To install it manually, just double-click the file.

By default, Python installs to a directory:

```
C:\Python27\
```

Warning: installation does not automatically modify the PATH environment variable.

Assuming that your Python installation is in C:\Python27, add this to your PATH:

```
C:\Python27\;C:\Python27\Scripts\
```

Now to check if Python installation is valid write in cmd:

```
python --version
```

Python 2.x and 3.x Side-By-Side

To install and use both Python 2.x and 3.x side-by-side on a Windows machine:

1. Install Python 2.x using the MSI installer.
    - Ensure Python is installed for all users.
    - Optional: add Python to `PATH` to make Python 2.x callable from the command-line using `python`.
2. Install Python 3.x using its respective installer.
    - Again, ensure Python is installed for all users.
    - Optional: add Python to `PATH` to make Python 3.x callable from the command-line using `python`. This may override Python 2.x `PATH` settings, so double-check your `PATH` and ensure it's configured to your preferences.
    - Make sure to install the `py launcher` for all users.

Python 3 will install the Python launcher which can be used to launch Python 2.x and Python 3.x interchangeably from the command-line:

```
P:\>py -3
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>

C:\>py -2
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC v.1500 32 Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

To use the corresponding version of `pip` for a specific Python version, use:

```
C:\>py -3 -m pip -V
pip 9.0.1 from C:\Python36\lib\site-packages (python 3.6)

C:\>py -2 -m pip -V
pip 9.0.1 from C:\Python27\lib\site-packages (python 2.7)
```

Linux

The latest versions of CentOS, Fedora, Red Hat Enterprise (RHEL) and Ubuntu come with Python 2.7.

To install Python 2.7 on linux manually, just do the following in terminal:

```
wget --no-check-certificate https://www.python.org/ftp/python/2.7.X/Python-2.7.X.tgz
tar -xzf Python-2.7.X.tgz
cd Python-2.7.X
./configure
make
```

```
sudo make install
```

Also add the path of new python in PATH environment variable. If new python is in `/root/python-2.7.X` then run `export PATH = $PATH:/root/python-2.7.X`

Now to check if Python installation is valid write in terminal:

```
python --version
```

Ubuntu (From Source)

If you need Python 3.6 you can install it from source as shown below (Ubuntu 16.10 and 17.04 have 3.6 version in the universal repository). Below steps have to be followed for Ubuntu 16.04 and lower versions:

```
sudo apt install build-essential checkinstall
sudo apt install libreadline-gplv2-dev libncursesw5-dev libssl-dev libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev
wget https://www.python.org/ftp/python/3.6.1/Python-3.6.1.tar.xz
tar xvf Python-3.6.1.tar.xz
cd Python-3.6.1/
./configure --enable-optimizations
sudo make altinstall
```

macOS

As we speak, macOS comes installed with Python 2.7.10, but this version is outdated and slightly modified from the regular Python.

> The version of Python that ships with OS X is great for learning but it's not good for development. The version shipped with OS X may be out of date from the official current Python release, which is considered the stable production version. (source)

Install Homebrew:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Install Python 2.7:

```
brew install python
```

For Python 3.x, use the command `brew install python3` instead.

Section 1.11: String function - str() and repr()

There are two functions that can be used to obtain a readable representation of an object.

`repr(x)` calls `x.__repr__()`: a representation of `x`. `eval` will usually convert the result of this function back to the original object.

`str(x)` calls `x.__str__()`: a human-readable string that describes the object. This may elide some technical detail.

repr()

For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to `eval()`. Otherwise, the representation is a string enclosed in angle brackets that contains the name of the type of the object along with additional information. This often includes the name and address of the object.

str()

For strings, this returns the string itself. The difference between this and `repr(object)` is that `str(object)` does not always attempt to return a string that is acceptable to `eval()`. Rather, its goal is to return a printable or 'human readable' string. If no argument is given, this returns the empty string, `''`.

Example 1:

```
s = """w'o"w"""
repr(s) # Output: '\'w\\\'o"w\''
str(s)  # Output: 'w\'o"w'
eval(str(s)) == s  # Gives a SyntaxError
eval(repr(s)) == s # Output: True
```

Example 2:

```
import datetime
today = datetime.datetime.now()
str(today)  # Output: '2016-09-15 06:58:46.915000'
repr(today) # Output: 'datetime.datetime(2016, 9, 15, 6, 58, 46, 915000)'
```

When writing a class, you can override these methods to do whatever you want:

```
class Represent(object):

    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "Represent(x={},y=\"{}\")".format(self.x, self.y)

    def __str__(self):
        return "Representing x as {} and y as {}".format(self.x, self.y)
```

Using the above class we can see the results:

```
r = Represent(1, "Hopper")
print(r)  # prints __str__
print(r.__repr__)  # prints __repr__: '<bound method Represent.__repr__ of
Represent(x=1,y="Hopper")>'
rep = r.__repr__()  # sets the execution of __repr__ to a new variable
print(rep)  # prints 'Represent(x=1,y="Hopper")'
r2 = eval(rep) # evaluates rep
print(r2)  # prints __str__ from new object
print(r2 == r)  # prints 'False' because they are different objects
```

Section 1.12: Installing external modules using pip

`pip` is your friend when you need to install any package from the plethora of choices available at the python package index (PyPI). `pip` is already installed if you're using Python 2 >= 2.7.9 or Python 3 >= 3.4 downloaded from python.org. For computers running Linux or another *nix with a native package manager, `pip` must often be manually installed.

On instances with both Python 2 and Python 3 installed, `pip` often refers to Python 2 and `pip3` to Python 3. Using `pip` will only install packages for Python 2 and `pip3` will only install packages for Python 3.

Finding / installing a package

Searching for a package is as simple as typing

```
$ pip search <query>
# Searches for packages whose name or summary contains <query>
```

Installing a package is as simple as typing *(in a terminal / command-prompt, not in the Python interpreter)*

```
$ pip install [package_name]           # latest version of the package

$ pip install [package_name]==x.x.x    # specific version of the package

$ pip install '[package_name]>=x.x.x'  # minimum version of the package
```

where `x.x.x` is the version number of the package you want to install.

When your server is behind proxy, you can install package by using below command:

```
$ pip --proxy http://<server address>:<port> install
```

Upgrading installed packages

When new versions of installed packages appear they are not automatically installed to your system. To get an overview of which of your installed packages have become outdated, run:

```
$ pip list --outdated
```

To upgrade a specific package use

```
$ pip install [package_name] --upgrade
```

Updating all outdated packages is not a standard functionality of `pip`.

Upgrading pip

You can upgrade your existing pip installation by using the following commands

- On Linux or macOS X:

  ```
  $ pip install -U pip
  ```

  You may need to use `sudo` with pip on some Linux Systems

- On Windows:

  ```
  py -m pip install -U pip
  ```

  or

  ```
  python -m pip install -U pip
  ```

For more information regarding pip do read here.

Section 1.13: Help Utility

Python has several functions built into the interpreter. If you want to get information of keywords, built-in functions, modules or topics open a Python console and enter:

```
>>> help()
```

You will receive information by entering keywords directly:

```
>>> help(help)
```

or within the utility:

```
help> help
```

which will show an explanation:

```
Help on _Helper in module _sitebuiltins object:

class _Helper(builtins.object)
| Define the builtin 'help'.
|
| This is a wrapper around pydoc.help that provides a helpful message
| when 'help' is typed at the Python interactive prompt.
|
| Calling help() at the Python prompt starts an interactive help session.
| Calling help(thing) prints help for the python object 'thing'.
|
| Methods defined here:
|
| __call__(self, *args, **kwds)
|
| __repr__(self)
|
| ----------------------------------------------------------------------
| Data descriptors defined here:
|
| __dict__
| dictionary for instance variables (if defined)
|
| __weakref__
| list of weak references to the object (if defined)
```

You can also request subclasses of modules:

```
help(pymysql.connections)
```

You can use help to access the docstrings of the different modules you have imported, e.g., try the following:

```
>>> help(math)
```

and you'll get an error

```
>>> import math
```

```
>>> help(math)
```

And now you will get a list of the available methods in the module, but only AFTER you have imported it.

Close the helper with `quit`

Chapter 2: Python Data Types

Data types are nothing but variables you use to reserve some space in memory. Python variables do not need an explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable.

Section 2.1: String Data Type

String are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Strings are immutable sequence data type, i.e each time one makes any changes to a string, completely new string object is created.

```
a_str = 'Hello World'
print(a_str)    #output will be whole string. Hello World
print(a_str[0])    #output will be first character. H
print(a_str[0:5])    #output will be first five characters. Hello
```

Section 2.2: Set Data Types

Sets are unordered collections of unique objects, there are two types of set:

1. Sets - They are mutable and new elements can be added once sets are defined

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)            # duplicates will be removed
> {'orange', 'banana', 'pear', 'apple'}
a = set('abracadabra')
print(a)                 # unique letters in a
> {'a', 'r', 'b', 'c', 'd'}
a.add('z')
print(a)
> {'a', 'c', 'r', 'b', 'z', 'd'}
```

2. Frozen Sets - They are immutable and new elements cannot added after its defined.

```
b = frozenset('asdfagsa')
print(b)
> frozenset({'f', 'g', 'd', 'a', 's'})
cities = frozenset(["Frankfurt", "Basel","Freiburg"])
print(cities)
> frozenset({'Frankfurt', 'Basel', 'Freiburg'})
```

Section 2.3: Numbers data type

Numbers have four types in Python. Int, float, complex, and long.

```
int_num = 10    #int value
float_num = 10.2    #float value
complex_num = 3.14j    #complex value
long_num = 1234567L    #long value
```

Section 2.4: List Data Type

A list contains items separated by commas and enclosed within square brackets [].lists are almost similar to arrays in C. One difference is that all the items belonging to a list can be of different data type.

```
list = [123,'abcd',10.2,'d']    #can be an array of any data type or single data type.
list1 = ['hello','world']
print(list)    #will output whole list. [123,'abcd',10.2,'d']
print(list[0:2])    #will output first two element of list. [123,'abcd']
print(list1 * 2)    #will gave list1 two times. ['hello','world','hello','world']
print(list + list1)    #will gave concatenation of both the lists.
[123,'abcd',10.2,'d','hello','world']
```

Section 2.5: Dictionary Data Type

Dictionary consists of key-value pairs. It is enclosed by curly braces {} and values can be assigned and accessed using square brackets[].

```
dic={'name':'red','age':10}
print(dic)    #will output all the key-value pairs. {'name':'red','age':10}
print(dic['name'])    #will output only value with 'name' key. 'red'
print(dic.values())    #will output list of values in dic. ['red',10]
print(dic.keys())    #will output list of keys. ['name','age']
```

Section 2.6: Tuple Data Type

Lists are enclosed in brackets [ ] and their elements and size can be changed, while tuples are enclosed in parentheses ( ) and cannot be updated. Tuples are immutable.

```
tuple = (123,'hello')
tuple1 = ('world')
print(tuple)    #will output whole tuple. (123,'hello')
print(tuple[0])    #will output first value. (123)
print(tuple + tuple1)    #will output (123,'hello','world')
tuple[1]='update'    #this will give you error.
```

Chapter 3: Indentation

Section 3.1: Simple example

For Python, Guido van Rossum based the grouping of statements on indentation. The reasons for this are explained in the first section of the "Design and History Python FAQ". Colons, `:`, are used to declare an indented code block, such as the following example:

```
class ExampleClass:
    #Every function belonging to a class must be indented equally
    def __init__(self):
        name = "example"

    def someFunction(self, a):
        #Notice everything belonging to a function must be indented
        if a > 5:
            return True
        else:
            return False

#If a function is not indented to the same level it will not be considers as part of the parent class
def separateFunction(b):
    for i in b:
    #Loops are also indented and nested conditions start a new indentation
        if i == 1:
            return True
    return False

separateFunction([2,3,5,6,1])
```

Spaces or Tabs?

The recommended indentation is 4 spaces but tabs or spaces can be used so long as they are consistent. ***Do not mix tabs and spaces in Python*** as this will cause an error in Python 3 and can causes errors in Python 2.

Section 3.2: How Indentation is Parsed

Whitespace is handled by the lexical analyzer before being parsed.

The lexical analyzer uses a stack to store indentation levels. At the beginning, the stack contains just the value 0, which is the leftmost position. Whenever a nested block begins, the new indentation level is pushed on the stack, and an "INDENT" token is inserted into the token stream which is passed to the parser. There can never be more than one "INDENT" token in a row (`IndentationError`).

When a line is encountered with a smaller indentation level, values are popped from the stack until a value is on top which is equal to the new indentation level (if none is found, a syntax error occurs). For each value popped, a "DEDENT" token is generated. Obviously, there can be multiple "DEDENT" tokens in a row.

The lexical analyzer skips empty lines (those containing only whitespace and possibly comments), and will never generate either "INDENT" or "DEDENT" tokens for them.

At the end of the source code, "DEDENT" tokens are generated for each indentation level left on the stack, until just the 0 is left.

For example:

```
if foo:
    if bar:
        x = 42
else:
    print foo
```

is analyzed as:

```
<if> <foo> <:>                  [0]
<INDENT> <if> <bar> <:>         [0, 4]
<INDENT> <x> <=> <42>           [0, 4, 8]
<DEDENT> <DEDENT> <else> <:>    [0]
<INDENT> <print> <foo>          [0, 2]
<DEDENT>
```

The parser than handles the "INDENT" and "DEDENT" tokens as block delimiters.

Section 3.3: Indentation Errors

The spacing should be even and uniform throughout. Improper indentation can cause an `IndentationError` or cause the program to do something unexpected. The following example raises an `IndentationError`:

```
a = 7
if a > 5:
  print "foo"
else:
  print "bar"
 print "done"
```

Or if the line following a colon is not indented, an `IndentationError` will also be raised:

```
if True:
print "true"
```

If you add indentation where it doesn't belong, an `IndentationError` will be raised:

```
if  True:
    a = 6
        b = 5
```

If you forget to un-indent functionality could be lost. In this example `None` is returned instead of the expected `False`:

```
def isEven(a):
    if a%2 ==0:
        return True
        #this next line should be even with the if
        return False
print isEven(7)
```

Chapter 4: Comments and Documentation

Section 4.1: Single line, inline and multiline comments

Comments are used to explain code when the basic code itself isn't clear.

Python ignores comments, and so will not execute code in there, or raise syntax errors for plain English sentences.

Single-line comments begin with the hash character (#) and are terminated by the end of line.

- Single line comment:

```
# This is a single line comment in Python
```

- Inline comment:

```
print("Hello World")  # This line prints "Hello World"
```

- Comments spanning multiple lines have `"""` or `'''` on either end. This is the same as a multiline string, but they can be used as comments:

```
"""
This type of comment spans multiple lines.
These are mostly used for documentation of functions, classes and modules.
"""
```

Section 4.2: Programmatically accessing docstrings

Docstrings are - unlike regular comments - stored as an attribute of the function they document, meaning that you can access them programmatically.

An example function

```
def func():
    """This is a function that does nothing at all"""
    return
```

The docstring can be accessed using the `__doc__` attribute:

```
print(func.__doc__)
```

> This is a function that does nothing at all

```
help(func)
```

> Help on function `func` in module `__main__`:
>
> `func()`
>
> This is a function that does nothing at all

Another example function

`function.__doc__` is just the actual docstring as a string, while the `help` function provides general information about a function, including the docstring. Here's a more helpful example:

```
def greet(name, greeting="Hello"):
    """Print a greeting to the user `name`

    Optional parameter `greeting` can change what they're greeted with."""

    print("{} {}".format(greeting, name))
```

```
help(greet)
```

> Help on function `greet` in module `__main__`:
>
> `greet(name, greeting='Hello')`
>
> Print a greeting to the user `name`
> Optional parameter `greeting` can change what they're greeted with.

Advantages of docstrings over regular comments

Just putting no docstring or a regular comment in a function makes it a lot less helpful.

```
def greet(name, greeting="Hello"):
    # Print a greeting to the user `name`
    # Optional parameter `greeting` can change what they're greeted with.

    print("{} {}".format(greeting, name))
```

```
print(greet.__doc__)
```

> None

```
help(greet)
```

> Help on function greet in module **main**:
>
> `greet(name, greeting='Hello')`

Section 4.3: Write documentation using docstrings

A docstring is a multi-line comment used to document modules, classes, functions and methods. It has to be the first statement of the component it describes.

```
def hello(name):
    """Greet someone.

    Print a greeting ("Hello") for the person with the given name.
    """

    print("Hello "+name)
```

```
class Greeter:
    """An object used to greet people.
```

```
    It contains multiple greeting functions for several languages
    and times of the  day.
    """
```

The value of the docstring can be accessed within the program and is - for example - used by the `help` command.

Syntax conventions
PEP 257

PEP 257 defines a syntax standard for docstring comments. It basically allows two types:

- One-line Docstrings:

According to PEP 257, they should be used with short and simple functions. Everything is placed in one line, e.g:

```
def hello():
    """Say hello to your friends."""
    print("Hello my friends!")
```

The docstring shall end with a period, the verb should be in the imperative form.

- Multi-line Docstrings:

Multi-line docstring should be used for longer, more complex functions, modules or classes.

```
def hello(name, language="en"):
    """Say hello to a person.

    Arguments:
    name: the name of the person
    language: the language in which the person should be greeted
    """

    print(greeting[language]+" "+name)
```

They start with a short summary (equivalent to the content of a one-line docstring) which can be on the same line as the quotation marks or on the next line, give additional detail and list parameters and return values.

Note PEP 257 defines what information should be given within a docstring, it doesn't define in which format it should be given. This was the reason for other parties and documentation parsing tools to specify their own standards for documentation, some of which are listed below and in this question.

Sphinx

Sphinx is a tool to generate HTML based documentation for Python projects based on docstrings. Its markup language used is reStructuredText. They define their own standards for documentation, pythonhosted.org hosts a very good description of them. The Sphinx format is for example used by the pyCharm IDE.

A function would be documented like this using the Sphinx/reStructuredText format:

```
def hello(name, language="en"):
    """Say hello to a person.

    :param name: the name of the person
    :type name: str
    :param language: the language in which the person should be greeted
    :type language: str
```

```
    :return: a number
    :rtype: int
    """

    print(greeting[language]+" "+name)
    return 4
```

Google Python Style Guide

Google has published Google Python Style Guide which defines coding conventions for Python, including documentation comments. In comparison to the Sphinx/reST many people say that documentation according to Google's guidelines is better human-readable.

The pythonhosted.org page mentioned above also provides some examples for good documentation according to the Google Style Guide.

Using the Napoleon plugin, Sphinx can also parse documentation in the Google Style Guide-compliant format.

A function would be documented like this using the Google Style Guide format:

```
def hello(name, language="en"):
    """Say hello to a person.

    Args:
        name: the name of the person as string
        language: the language code string

    Returns:
        A number.
    """

    print(greeting[language]+" "+name)
    return 4
```

Chapter 5: Date and Time

Section 5.1: Parsing a string into a timezone aware datetime object

Python 3.2+ has support for `%z` format when parsing a string into a `datetime` object.

> UTC offset in the form `+HHMM` or `-HHMM` (empty string if the object is naive).

Python 3.x `Version ≥ 3.2`

```
import datetime
dt = datetime.datetime.strptime("2016-04-15T08:27:18-0500", "%Y-%m-%dT%H:%M:%S%z")
```

For other versions of Python, you can use an external library such as `dateutil`, which makes parsing a string with timezone into a `datetime` object is quick.

```
import dateutil.parser
dt = dateutil.parser.parse("2016-04-15T08:27:18-0500")
```

The `dt` variable is now a `datetime` object with the following value:

```
datetime.datetime(2016, 4, 15, 8, 27, 18, tzinfo=tzoffset(None, -18000))
```

Section 5.2: Constructing timezone-aware datetimes

By default all `datetime` objects are naive. To make them timezone-aware, you must attach a `tzinfo` object, which provides the UTC offset and timezone abbreviation as a function of date and time.

Fixed Offset Time Zones

For time zones that are a fixed offset from UTC, in Python 3.2+, the `datetime` module provides the `timezone` class, a concrete implementation of `tzinfo`, which takes a `timedelta` and an (optional) name parameter:

Python 3.x `Version ≥ 3.2`

```
from datetime import datetime, timedelta, timezone
JST = timezone(timedelta(hours=+9))

dt = datetime(2015, 1, 1, 12, 0, 0, tzinfo=JST)
print(dt)
# 2015-01-01 12:00:00+09:00

print(dt.tzname())
# UTC+09:00

dt = datetime(2015, 1, 1, 12, 0, 0, tzinfo=timezone(timedelta(hours=9), 'JST'))
print(dt.tzname)
# 'JST'
```

For Python versions before 3.2, it is necessary to use a third party library, such as `dateutil`. `dateutil` provides an equivalent class, `tzoffset`, which (as of version 2.5.3) takes arguments of the form `dateutil.tz.tzoffset(tzname, offset)`, where `offset` is specified in seconds:

Python 3.x `Version < 3.2`

Python 2.x `Version < 2.7`

```
from datetime import datetime, timedelta
from dateutil import tz

JST = tz.tzoffset('JST', 9 * 3600) # 3600 seconds per hour
dt = datetime(2015, 1, 1, 12, 0, tzinfo=JST)
print(dt)
# 2015-01-01 12:00:00+09:00
print(dt.tzname)
# 'JST'
```

Zones with daylight savings time

For zones with daylight savings time, python standard libraries do not provide a standard class, so it is necessary to use a third party library. `pytz` and `dateutil` are popular libraries providing time zone classes.

In addition to static time zones, `dateutil` provides time zone classes that use daylight savings time (see the documentation for the `tz` module). You can use the `tz.gettz()` method to get a time zone object, which can then be passed directly to the `datetime` constructor:

```
from datetime import datetime
from dateutil import tz
local = tz.gettz() # Local time
PT = tz.gettz('US/Pacific') # Pacific time

dt_l = datetime(2015, 1, 1, 12, tzinfo=local) # I am in EST
dt_pst = datetime(2015, 1, 1, 12, tzinfo=PT)
dt_pdt = datetime(2015, 7, 1, 12, tzinfo=PT) # DST is handled automatically
print(dt_l)
# 2015-01-01 12:00:00-05:00
print(dt_pst)
# 2015-01-01 12:00:00-08:00
print(dt_pdt)
# 2015-07-01 12:00:00-07:00
```

CAUTION: As of version 2.5.3, `dateutil` does not handle ambiguous datetimes correctly, and will always default to the *later* date. There is no way to construct an object with a `dateutil` timezone representing, for example `2015-11-01 1:30 EDT-4`, since this is *during* a daylight savings time transition.

All edge cases are handled properly when using `pytz`, but `pytz` time zones should *not* be directly attached to time zones through the constructor. Instead, a `pytz` time zone should be attached using the time zone's `localize` method:

```
from datetime import datetime, timedelta
import pytz

PT = pytz.timezone('US/Pacific')
dt_pst = PT.localize(datetime(2015, 1, 1, 12))
dt_pdt = PT.localize(datetime(2015, 11, 1, 0, 30))
print(dt_pst)
# 2015-01-01 12:00:00-08:00
print(dt_pdt)
# 2015-11-01 00:30:00-07:00
```

Be aware that if you perform datetime arithmetic on a `pytz`-aware time zone, you must either perform the calculations in UTC (if you want absolute elapsed time), or you must call `normalize()` on the result:

```
dt_new = dt_pdt + timedelta(hours=3) # This should be 2:30 AM PST
print(dt_new)
# 2015-11-01 03:30:00-07:00
dt_corrected = PT.normalize(dt_new)
print(dt_corrected)
# 2015-11-01 02:30:00-08:00
```

Section 5.3: Computing time differences

the `timedelta` module comes in handy to compute differences between times:

```
from datetime import datetime, timedelta
now = datetime.now()
then = datetime(2016, 5, 23)      # datetime.datetime(2016, 05, 23, 0, 0, 0)
```

Specifying time is optional when creating a new `datetime` object

```
delta = now-then
```

`delta` is of type `timedelta`

```
print(delta.days)
# 60
print(delta.seconds)
# 40826
```

To get n day's after and n day's before date we could use:

n day's after date:

```
def get_n_days_after_date(date_format="%d %B %Y", add_days=120):

    date_n_days_after = datetime.datetime.now() + timedelta(days=add_days)
    return date_n_days_after.strftime(date_format)
```

n day's before date:

```
 def get_n_days_before_date(self, date_format="%d %B %Y", days_before=120):

        date_n_days_ago = datetime.datetime.now() - timedelta(days=days_before)
        return date_n_days_ago.strftime(date_format)
```

Section 5.4: Basic datetime objects usage

The datetime module contains three primary types of objects - date, time, and datetime.

```
import datetime

# Date object
today = datetime.date.today()
new_year = datetime.date(2017, 01, 01) #datetime.date(2017, 1, 1)

# Time object
noon = datetime.time(12, 0, 0) #datetime.time(12, 0)

# Current datetime
now = datetime.datetime.now()
```

```
# Datetime object
millenium_turn = datetime.datetime(2000, 1, 1, 0, 0, 0) #datetime.datetime(2000, 1, 1, 0, 0)
```

Arithmetic operations for these objects are only supported within same datatype and performing simple arithmetic with instances of different types will result in a TypeError.

```
# subtraction of noon from today
noon-today
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'datetime.time' and 'datetime.date'
However, it is straightforward to convert between types.

# Do this instead
print('Time since the millenium at midnight: ',
      datetime.datetime(today.year, today.month, today.day) - millenium_turn)

# Or this
print('Time since the millenium at noon: ',
      datetime.datetime.combine(today, noon) - millenium_turn)
```

Section 5.5: Switching between time zones

To switch between time zones, you need datetime objects that are timezone-aware.

```
from datetime import datetime
from dateutil import tz

utc = tz.tzutc()
local = tz.tzlocal()

utc_now = datetime.utcnow()
utc_now # Not timezone-aware.

utc_now = utc_now.replace(tzinfo=utc)
utc_now # Timezone-aware.

local_now = utc_now.astimezone(local)
local_now # Converted to local time.
```

Section 5.6: Simple date arithmetic

Dates don't exist in isolation. It is common that you will need to find the amount of time between dates or determine what the date will be tomorrow. This can be accomplished using `timedelta` objects

```
import datetime

today = datetime.date.today()
print('Today:', today)

yesterday = today - datetime.timedelta(days=1)
print('Yesterday:', yesterday)

tomorrow = today + datetime.timedelta(days=1)
print('Tomorrow:', tomorrow)

print('Time between tomorrow and yesterday:', tomorrow - yesterday)
```

This will produce results similar to:

```
Today: 2016-04-15
Yesterday: 2016-04-14
Tomorrow: 2016-04-16
Difference between tomorrow and yesterday: 2 days, 0:00:00
```

Section 5.7: Converting timestamp to datetime

The `datetime` module can convert a POSIX `timestamp` to a ITC `datetime` object.

The Epoch is January 1st, 1970 midnight.

```
import time
from datetime import datetime
seconds_since_epoch=time.time()  #1469182681.709

utc_date=datetime.utcfromtimestamp(seconds_since_epoch) #datetime.datetime(2016, 7, 22, 10, 18, 1,
709000)
```

Section 5.8: Subtracting months from a date accurately

Using the `calendar` module

```
import calendar
from datetime import date

def monthdelta(date, delta):
    m, y = (date.month+delta) % 12, date.year + ((date.month)+delta-1) // 12
    if not m: m = 12
    d = min(date.day, calendar.monthrange(y, m)[1])
    return date.replace(day=d,month=m, year=y)

next_month = monthdelta(date.today(), 1) #datetime.date(2016, 10, 23)
```

Using the `dateutils` module

```
import datetime
import dateutil.relativedelta

d = datetime.datetime.strptime("2013-03-31", "%Y-%m-%d")
d2 = d - dateutil.relativedelta.relativedelta(months=1)  #datetime.datetime(2013, 2, 28, 0, 0)
```

Section 5.9: Parsing an arbitrary ISO 8601 timestamp with minimal libraries

Python has only limited support for parsing ISO 8601 timestamps. For `strptime` you need to know exactly what format it is in. As a complication the stringification of a `datetime` is an ISO 8601 timestamp, with space as a separator and 6 digit fraction:

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 555555))
# '2016-07-22 09:25:59.555555'
```

but if the fraction is 0, no fractional part is output

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 0))
# '2016-07-22 09:25:59'
```

But these 2 forms need a *different* format for `strptime`. Furthermore, `strptime' does not support at all parsing minute timezones that have a:in it, thus2016-07-22 09:25:59+0300can be parsed, but the standard format2016-07-22 09:25:59+03:00` cannot.

There is a single-file library called `iso8601` which properly parses ISO 8601 timestamps and only them.

It supports fractions and timezones, and the `T` separator all with a single function:

```
import iso8601
iso8601.parse_date('2016-07-22 09:25:59')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
iso8601.parse_date('2016-07-22 09:25:59+03:00')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<FixedOffset '+03:00' ...>)
iso8601.parse_date('2016-07-22 09:25:59Z')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
iso8601.parse_date('2016-07-22T09:25:59.000111+03:00')
# datetime.datetime(2016, 7, 22, 9, 25, 59, 111, tzinfo=<FixedOffset '+03:00' ...>)
```

If no timezone is set, `iso8601.parse_date` defaults to UTC. The default zone can be changed with `default_zone` keyword argument. Notably, if this is `None` instead of the default, then those timestamps that do not have an explicit timezone are returned as naive datetimes instead:

```
iso8601.parse_date('2016-07-22T09:25:59', default_timezone=None)
# datetime.datetime(2016, 7, 22, 9, 25, 59)
iso8601.parse_date('2016-07-22T09:25:59Z', default_timezone=None)
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
```

Section 5.10: Get an ISO 8601 timestamp

Without timezone, with microseconds

```
from datetime import datetime

datetime.now().isoformat()
# Out: '2016-07-31T23:08:20.886783'
```

With timezone, with microseconds

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).isoformat()
# Out: '2016-07-31T23:09:43.535074-07:00'
```

With timezone, without microseconds

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).replace(microsecond=0).isoformat()
# Out: '2016-07-31T23:10:30-07:00'
```

See ISO 8601 for more information about the ISO 8601 format.

Section 5.11: Parsing a string with a short time zone name into

a timezone aware datetime object

Using the `dateutil` library as in the previous example on parsing timezone-aware timestamps, it is also possible to parse timestamps with a specified "short" time zone name.

For dates formatted with short time zone names or abbreviations, which are generally ambiguous (e.g. CST, which could be Central Standard Time, China Standard Time, Cuba Standard Time, etc - more can be found here) or not necessarily available in a standard database, it is necessary to specify a mapping between time zone abbreviation and `tzinfo` object.

```
from dateutil import tz
from dateutil.parser import parse

ET = tz.gettz('US/Eastern')
CT = tz.gettz('US/Central')
MT = tz.gettz('US/Mountain')
PT = tz.gettz('US/Pacific')

us_tzinfos = {'CST': CT, 'CDT': CT,
              'EST': ET, 'EDT': ET,
              'MST': MT, 'MDT': MT,
              'PST': PT, 'PDT': PT}

dt_est = parse('2014-01-02 04:00:00 EST', tzinfos=us_tzinfos)
dt_pst = parse('2016-03-11 16:00:00 PST', tzinfos=us_tzinfos)
```

After running this:

```
dt_est
# datetime.datetime(2014, 1, 2, 4, 0, tzinfo=tzfile('/usr/share/zoneinfo/US/Eastern'))
dt_pst
# datetime.datetime(2016, 3, 11, 16, 0, tzinfo=tzfile('/usr/share/zoneinfo/US/Pacific'))
```

It is worth noting that if using a `pytz` time zone with this method, it will *not* be properly localized:

```
from dateutil.parser import parse
import pytz

EST = pytz.timezone('America/New_York')
dt = parse('2014-02-03 09:17:00 EST', tzinfos={'EST': EST})
```

This simply attaches the `pytz` time zone to the datetime:

```
dt.tzinfo # Will be in Local Mean Time!
# <DstTzInfo 'America/New_York' LMT-1 day, 19:04:00 STD>
```

If using this method, you should probably re-`localize` the naive portion of the datetime after parsing:

```
dt_fixed = dt.tzinfo.localize(dt.replace(tzinfo=None))
dt_fixed.tzinfo # Now it's EST.
# <DstTzInfo 'America/New_York' EST-1 day, 19:00:00 STD>)
```

Section 5.12: Fuzzy datetime parsing (extracting datetime out of a text)

It is possible to extract a date out of a text using the dateutil parser in a "fuzzy" mode, where components of the

string not recognized as being part of a date are ignored.

```
from dateutil.parser import parse

dt = parse("Today is January 1, 2047 at 8:21:00AM", fuzzy=True)
print(dt)
```

`dt` is now a *`datetime` object* and you would see `datetime.datetime(2047, 1, 1, 8, 21)` printed.

Section 5.13: Iterate over dates

Sometimes you want to iterate over a range of dates from a start date to some end date. You can do it using `datetime` library and `timedelta` object:

```
import datetime

# The size of each step in days
day_delta = datetime.timedelta(days=1)

start_date = datetime.date.today()
end_date = start_date + 7*day_delta

for i in range((end_date - start_date).days):
    print(start_date + i*day_delta)
```

Which produces:

```
2016-07-21
2016-07-22
2016-07-23
2016-07-24
2016-07-25
2016-07-26
2016-07-27
```

Chapter 6: Date Formatting

Section 6.1: Time between two date-times

```
from datetime import datetime

a = datetime(2016,10,06,0,0,0)
b = datetime(2016,10,01,23,59,59)

a-b
# datetime.timedelta(4, 1)

(a-b).days
# 4
(a-b).total_seconds()
# 518399.0
```

Section 6.2: Outputting datetime object to string

Uses C standard format codes.

```
from datetime import datetime
datetime_for_string = datetime(2016,10,1,0,0)
datetime_string_format = '%b %d %Y, %H:%M:%S'
datetime.strftime(datetime_for_string,datetime_string_format)
# Oct 01 2016, 00:00:00
```

Section 6.3: Parsing string to datetime object

Uses C standard format codes.

```
from datetime import datetime
datetime_string = 'Oct 1 2016, 00:00:00'
datetime_string_format = '%b %d %Y, %H:%M:%S'
datetime.strptime(datetime_string, datetime_string_format)
# datetime.datetime(2016, 10, 1, 0, 0)
```

Chapter 7: Enum

Section 7.1: Creating an enum (Python 2.4 through 3.3)

Enums have been backported from Python 3.4 to Python 2.4 through Python 3.3. You can get this the enum34 backport from PyPI.

```
pip install enum34
```

Creation of an enum is identical to how it works in Python 3.4+

```
from enum import Enum

class Color(Enum):
    red = 1
    green = 2
    blue = 3

print(Color.red)  # Color.red
print(Color(1))  # Color.red
print(Color['red'])  # Color.red
```

Section 7.2: Iteration

Enums are iterable:

```
class Color(Enum):
    red = 1
    green = 2
    blue = 3

[c for c in Color]  # [<Color.red: 1>, <Color.green: 2>, <Color.blue: 3>]
```

Chapter 8: Set

Section 8.1: Operations on sets

with other sets

```
# Intersection
{1, 2, 3, 4, 5}.intersection({3, 4, 5, 6})  # {3, 4, 5}
{1, 2, 3, 4, 5} & {3, 4, 5, 6}              # {3, 4, 5}

# Union
{1, 2, 3, 4, 5}.union({3, 4, 5, 6})  # {1, 2, 3, 4, 5, 6}
{1, 2, 3, 4, 5} | {3, 4, 5, 6}       # {1, 2, 3, 4, 5, 6}

# Difference
{1, 2, 3, 4}.difference({2, 3, 5})  # {1, 4}
{1, 2, 3, 4} - {2, 3, 5}            # {1, 4}

# Symmetric difference with
{1, 2, 3, 4}.symmetric_difference({2, 3, 5})  # {1, 4, 5}
{1, 2, 3, 4} ^ {2, 3, 5}                      # {1, 4, 5}

# Superset check
{1, 2}.issuperset({1, 2, 3})  # False
{1, 2} >= {1, 2, 3}           # False

# Subset check
{1, 2}.issubset({1, 2, 3})  # True
{1, 2} <= {1, 2, 3}         # True

# Disjoint check
{1, 2}.isdisjoint({3, 4})  # True
{1, 2}.isdisjoint({1, 4})  # False
```

with single elements

```
# Existence check
2 in {1,2,3}      # True
4 in {1,2,3}      # False
4 not in {1,2,3}  # True

# Add and Remove
s = {1,2,3}
s.add(4)        # s == {1,2,3,4}

s.discard(3)    # s == {1,2,4}
s.discard(5)    # s == {1,2,4}

s.remove(2)     # s == {1,4}
s.remove(2)     # KeyError!
```

Set operations return new sets, but have the corresponding in-place versions:

method	in-place operation	in-place method
union	s |= t	update
intersection	s &= t	intersection_update
difference	s -= t	difference_update

symmetric_difference	s ^= t
symmetric_difference_update	

For example:

```
s = {1, 2}
s.update({3, 4})   # s == {1, 2, 3, 4}
```

Section 8.2: Get the unique elements of a list

Let's say you've got a list of restaurants -- maybe you read it from a file. You care about the *unique* restaurants in the list. The best way to get the unique elements from a list is to turn it into a set:

```
restaurants = ["McDonald's", "Burger King", "McDonald's", "Chicken Chicken"]
unique_restaurants = set(restaurants)
print(unique_restaurants)
# prints {'Chicken Chicken', "McDonald's", 'Burger King'}
```

Note that the set is not in the same order as the original list; that is because sets are *unordered*, just like `dict`s.

This can easily be transformed back into a `List` with Python's built in `list` function, giving another list that is the same list as the original but without duplicates:

```
list(unique_restaurants)
# ['Chicken Chicken', "McDonald's", 'Burger King']
```

It's also common to see this as one line:

```
# Removes all duplicates and returns another list
list(set(restaurants))
```

Now any operations that could be performed on the original list can be done again.

Section 8.3: Set of Sets

```
{{1,2}, {3,4}}
```

leads to:

```
TypeError: unhashable type: 'set'
```

Instead, use `frozenset`:

```
{frozenset({1, 2}), frozenset({3, 4})}
```

Section 8.4: Set Operations using Methods and Builtins

We define two sets `a` and `b`

```
>>> a = {1, 2, 2, 3, 4}
>>> b = {3, 3, 4, 4, 5}
```

> NOTE: `{1}` creates a set of one element, but `{}` creates an empty `dict`. The correct way to create an empty set is `set()`.

Intersection

`a.intersection(b)` returns a new set with elements present in both `a` and `b`

```
>>> a.intersection(b)
{3, 4}
```

Union

`a.union(b)` returns a new set with elements present in either `a` and `b`

```
>>> a.union(b)
{1, 2, 3, 4, 5}
```

Difference

`a.difference(b)` returns a new set with elements present in `a` but not in `b`

```
>>> a.difference(b)
{1, 2}
>>> b.difference(a)
{5}
```

Symmetric Difference

`a.symmetric_difference(b)` returns a new set with elements present in either `a` or `b` but not in both

```
>>> a.symmetric_difference(b)
{1, 2, 5}
>>> b.symmetric_difference(a)
{1, 2, 5}
```

NOTE: `a.symmetric_difference(b) == b.symmetric_difference(a)`

Subset and superset

`c.issubset(a)` tests whether each element of `c` is in `a`.

`a.issuperset(c)` tests whether each element of `c` is in `a`.

```
>>> c = {1, 2}
>>> c.issubset(a)
True
>>> a.issuperset(c)
True
```

The latter operations have equivalent operators as shown below:

Method	Operator
`a.intersection(b)`	`a & b`
`a.union(b)`	`a|b`
`a.difference(b)`	`a - b`
`a.symmetric_difference(b)`	`a ^ b`
`a.issubset(b)`	`a <= b`
`a.issuperset(b)`	`a >= b`

Disjoint sets

Sets `a` and `d` are disjoint if no element in `a` is also in `d` and vice versa.

```
>>> d = {5, 6}
>>> a.isdisjoint(b) # {2, 3, 4} are in both sets
False
>>> a.isdisjoint(d)
True

# This is an equivalent check, but less efficient
>>> len(a & d) == 0
True

# This is even less efficient
>>> a & d == set()
True
```

Testing membership

The builtin `in` keyword searches for occurances

```
>>> 1 in a
True
>>> 6 in a
False
```

Length

The builtin `len()` function returns the number of elements in the set

```
>>> len(a)
4
>>> len(b)
3
```

Section 8.5: Sets versus multisets

Sets are unordered collections of distinct elements. But sometimes we want to work with unordered collections of elements that are not necessarily distinct and keep track of the elements' multiplicities.

Consider this example:

```
>>> setA = {'a','b','b','c'}
>>> setA
set(['a', 'c', 'b'])
```

By saving the strings `'a'`, `'b'`, `'b'`, `'c'` into a set data structure we've lost the information on the fact that `'b'` occurs twice. Of course saving the elements to a list would retain this information

```
>>> listA = ['a','b','b','c']
>>> listA
['a', 'b', 'b', 'c']
```

but a list data structure introduces an extra unneeded ordering that will slow down our computations.

For implementing multisets Python provides the `Counter` class from the `collections` module (starting from version 2.7):

Python 2.x `Version ≥ 2.7`

```
>>> from collections import Counter
>>> counterA = Counter(['a','b','b','c'])
>>> counterA
Counter({'b': 2, 'a': 1, 'c': 1})
```

`Counter` is a dictionary where where elements are stored as dictionary keys and their counts are stored as dictionary values. And as all dictionaries, it is an unordered collection.

Chapter 9: Simple Mathematical Operators

Numerical types and their metaclasses

The `numbers` module contains the abstract metaclasses for the numerical types:

subclasses	numbers.Number	numbers.Integral	numbers.Rational	numbers.Real	numbers.Complex
bool	✓	✓	✓	✓	✓
int	✓	✓	✓	✓	✓
fractions.Fraction	✓	–	✓	✓	✓
float	✓	–	–	✓	✓
complex	✓	–	–	–	✓
decimal.Decimal	✓	–	–	–	–

Python does common mathematical operators on its own, including integer and float division, multiplication, exponentiation, addition, and subtraction. The math module (included in all standard Python versions) offers expanded functionality like trigonometric functions, root operations, logarithms, and many more.

Section 9.1: Division

Python does integer division when both operands are integers. The behavior of Python's division operators have changed from Python 2.x and 3.x (see also Integer Division).

```
a, b, c, d, e = 3, 2, 2.0, -3, 10
```

Python 2.x `Version ≤ 2.7`

In Python 2 the result of the ' / ' operator depends on the type of the numerator and denominator.

```
a / b                # = 1

a / c                # = 1.5

d / b                # = -2

b / a                # = 0

d / e                # = -1
```

Note that because both `a` and `b` are `int`s, the result is an `int`.

The result is always rounded down (floored).

Because `c` is a float, the result of `a / c` is a `float`.

You can also use the operator module:

```
import operator          # the operator module provides 2-argument arithmetic functions
operator.div(a, b)       # = 1
operator.__div__(a, b) # = 1
```

Python 2.x `Version ≥ 2.2`

What if you want float division:

Recommended:

```
from __future__ import division # applies Python 3 style division to the entire module
a / b                   # = 1.5
a // b                  # = 1
```

Okay (if you don't want to apply to the whole module):

```
a / (b * 1.0)           # = 1.5
1.0 * a / b             # = 1.5
a / b * 1.0             # = 1.0    (careful with order of operations)

from operator import truediv
truediv(a, b)           # = 1.5
```

Not recommended (may raise TypeError, eg if argument is complex):

```
float(a) / b            # = 1.5
a / float(b)            # = 1.5
```

Python 2.x `Version ≥ 2.2`

The ' // ' operator in Python 2 forces floored division regardless of type.

```
a // b                # = 1
a // c                # = 1.0
```

Python 3.x `Version ≥ 3.0`

In Python 3 the `/` operator performs 'true' division regardless of types. The `//` operator performs floor division and maintains type.

```
a / b                   # = 1.5
e / b                   # = 5.0
a // b                  # = 1
a // c                  # = 1.0

import operator             # the operator module provides 2-argument arithmetic functions
operator.truediv(a, b)      # = 1.5
operator.floordiv(a, b)     # = 1
operator.floordiv(a, c)     # = 1.0
```

Possible combinations (builtin types):

- `int` and `int` (gives an `int` in Python 2 and a `float` in Python 3)
- `int` and `float` (gives a `float`)
- `int` and `complex` (gives a `complex`)
- `float` and `float` (gives a `float`)
- `float` and `complex` (gives a `complex`)
- `complex` and `complex` (gives a `complex`)

See PEP 238 for more information.

Section 9.2: Addition

```
a, b = 1, 2

# Using the "+" operator:
a + b                   # = 3
```

```
# Using the "in-place" "+=" operator to add and assign:
a += b                  # a = 3 (equivalent to a = a + b)

import operator         # contains 2 argument arithmetic functions for the examples

operator.add(a, b)      # = 5  since a is set to 3 right before this line

# The "+=" operator is equivalent to:
a = operator.iadd(a, b)    # a = 5 since a is set to 3 right before this line
```

Possible combinations (builtin types):

- `int` and `int` (gives an `int`)
- `int` and `float` (gives a `float`)
- `int` and `complex` (gives a `complex`)
- `float` and `float` (gives a `float`)
- `float` and `complex` (gives a `complex`)
- `complex` and `complex` (gives a `complex`)

Note: the `+` operator is also used for concatenating strings, lists and tuples:

```
"first string " + "second string"    # = 'first string second string'

[1, 2, 3] + [4, 5, 6]                # = [1, 2, 3, 4, 5, 6]
```

Section 9.3: Exponentiation

```
a, b = 2, 3

(a ** b)                # = 8
pow(a, b)               # = 8

import math
math.pow(a, b)          # = 8.0 (always float; does not allow complex results)

import operator
operator.pow(a, b)      # = 8
```

Another difference between the built-in `pow` and `math.pow` is that the built-in `pow` can accept three arguments:

```
a, b, c = 2, 3, 2

pow(2, 3, 2)            # 0, calculates (2 ** 3) % 2, but as per Python docs,
                        #    does so more efficiently
```

Special functions

The function `math.sqrt(x)` calculates the square root of `x`.

```
import math
import cmath
c = 4
math.sqrt(c)            # = 2.0 (always float; does not allow complex results)
cmath.sqrt(c)           # = (2+0j) (always complex)
```

To compute other roots, such as a cube root, raise the number to the reciprocal of the degree of the root. This could be done with any of the exponential functions or operator.

```
import math
x = 8
math.pow(x, 1/3) # evaluates to 2.0
x**(1/3) # evaluates to 2.0
```

The function `math.exp(x)` computes `e ** x`.

```
math.exp(0)  # 1.0
math.exp(1)  # 2.718281828459045 (e)
```

The function `math.expm1(x)` computes `e ** x - 1`. When x is small, this gives significantly better precision than `math.exp(x) - 1`.

```
math.expm1(0)       # 0.0

math.exp(1e-6) - 1  # 1.0000004999621837e-06
math.expm1(1e-6)    # 1.0000005000001665e-06
# exact result      # 1.0000005000001666667083333416666...
```

Section 9.4: Trigonometric Functions

```
a, b = 1, 2

import math

math.sin(a)  # returns the sine of 'a' in radians
# Out: 0.8414709848078965

math.cosh(b)  # returns the inverse hyperbolic cosine of 'b' in radians
# Out: 3.7621956910836314

math.atan(math.pi)  # returns the arc tangent of 'pi' in radians
# Out: 1.2626272556789115

math.hypot(a, b) # returns the Euclidean norm, same as math.sqrt(a*a + b*b)
# Out: 2.23606797749979
```

> Note that `math.hypot(x, y)` is also the length of the vector (or Euclidean distance) from the origin `(0, 0)` to the point `(x, y)`.
>
> To compute the Euclidean distance between two points `(x1, y1)` & `(x2, y2)` you can use `math.hypot` as follows
>
> ```
> math.hypot(x2-x1, y2-y1)
> ```

To convert from radians -> degrees and degrees -> radians respectively use `math.degrees` and `math.radians`

```
math.degrees(a)
# Out: 57.29577951308232

math.radians(57.29577951308232)
# Out: 1.0
```

Section 9.5: Inplace Operations

It is common within applications to need to have code like this:

```
a = a + 1
```

or

```
a = a * 2
```

There is an effective shortcut for these in place operations:

```
a += 1
# and
a *= 2
```

Any mathematic operator can be used before the '=' character to make an inplace operation:

- `-=` decrement the variable in place
- `+=` increment the variable in place
- `*=` multiply the variable in place
- `/=` divide the variable in place
- `//=` floor divide the variable in place # Python 3
- `%=` return the modulus of the variable in place
- `**=` raise to a power in place

Other in place operators exist for the bitwise operators (`^`, `|` etc)

Section 9.6: Subtraction

```
a, b = 1, 2

# Using the "-" operator:
b - a                   # = 1


import operator         # contains 2 argument arithmetic functions
operator.sub(b, a)      # = 1
```

Possible combinations (builtin types):

- `int` and `int` (gives an `int`)
- `int` and `float` (gives a `float`)
- `int` and `complex` (gives a `complex`)
- `float` and `float` (gives a `float`)
- `float` and `complex` (gives a `complex`)
- `complex` and `complex` (gives a `complex`)

Section 9.7: Multiplication

```
a, b = 2, 3

a * b                   # = 6

import operator
```

```
operator.mul(a, b)      # = 6
```

Possible combinations (builtin types):

- `int` and `int` (gives an `int`)
- `int` and `float` (gives a `float`)
- `int` and `complex` (gives a `complex`)
- `float` and `float` (gives a `float`)
- `float` and `complex` (gives a `complex`)
- `complex` and `complex` (gives a `complex`)

Note: The `*` operator is also used for repeated concatenation of strings, lists, and tuples:

```
3 * 'ab'  # = 'abababab'
3 * ('a', 'b')  # = ('a', 'b', 'a', 'b', 'a', 'b')
```

Section 9.8: Logarithms

By default, the `math.log` function calculates the logarithm of a number, base e. You can optionally specify a base as the second argument.

```
import math
import cmath

math.log(5)          # = 1.6094379124341003
# optional base argument. Default is math.e
math.log(5, math.e) # = 1.6094379124341003
cmath.log(5)         # = (1.6094379124341003+0j)
math.log(1000, 10)   # 3.0 (always returns float)
cmath.log(1000, 10)  # (3+0j)
```

Special variations of the `math.log` function exist for different bases.

```
# Logarithm base e - 1 (higher precision for low values)
math.log1p(5)        # = 1.791759469228055

# Logarithm base 2
math.log2(8)         # = 3.0

# Logarithm base 10
math.log10(100)      # = 2.0
cmath.log10(100)     # = (2+0j)
```

Section 9.9: Modulus

Like in many other languages, Python uses the `%` operator for calculating modulus.

```
3 % 4     # 3
10 % 2    # 0
6 % 4     # 2
```

Or by using the `operator` module:

```
import operator

operator.mod(3 , 4)      # 3
```

```
operator.mod(10 , 2)    # 0
operator.mod(6 , 4)     # 2
```

You can also use negative numbers.

```
-9 % 7      # 5
9 % -7      # -5
-9 % -7     # -2
```

If you need to find the result of integer division and modulus, you can use the `divmod` function as a shortcut:

```
quotient, remainder = divmod(9, 4)
# quotient = 2, remainder = 1 as 4 * 2 + 1 == 9
```

Chapter 10: Bitwise Operators

Bitwise operations alter binary strings at the bit level. These operations are incredibly basic and are directly supported by the processor. These few operations are necessary in working with device drivers, low-level graphics, cryptography, and network communications. This section provides useful knowledge and examples of Python's bitwise operators.

Section 10.1: Bitwise NOT

The `~` operator will flip all of the bits in the number. Since computers use signed number representations — most notably, the two's complement notation to encode negative binary numbers where negative numbers are written with a leading one (1) instead of a leading zero (0).

This means that if you were using 8 bits to represent your two's-complement numbers, you would treat patterns from `0000 0000` to `0111 1111` to represent numbers from 0 to 127 and reserve `1xxx xxxx` to represent negative numbers.

Eight-bit two's-complement numbers

Bits	Unsigned Value	Two's-complement Value
0000 0000	0	0
0000 0001	1	1
0000 0010	2	2
0111 1110	126	126
0111 1111	127	127
1000 0000	128	-128
1000 0001	129	-127
1000 0010	130	-126
1111 1110	254	-2
1111 1111	255	-1

In essence, this means that whereas `1010 0110` has an unsigned value of 166 (arrived at by adding `(128 * 1) + (64 * 0) + (32 * 1) + (16 * 0) + (8 * 0) + (4 * 1) + (2 * 1) + (1 * 0)`), it has a two's-complement value of -90 (arrived at by adding `(128 * 1) - (64 * 0) - (32 * 1) - (16 * 0) - (8 * 0) - (4 * 1) - (2 * 1) - (1 * 0)`, and complementing the value).

In this way, negative numbers range down to -128 (`1000 0000`). Zero (0) is represented as `0000 0000`, and minus one (-1) as `1111 1111`.

In general, though, this means `~n = -n - 1`.

```
# 0 = 0b0000 0000
~0
# Out: -1
# -1 = 0b1111 1111

# 1 = 0b0000 0001
~1
# Out: -2
# -2 = 1111 1110

# 2 = 0b0000 0010
~2
```

```
# Out: -3
# -3 = 0b1111 1101

# 123 = 0b0111 1011
~123
# Out: -124
# -124 = 0b1000 0100
```

Note, the overall effect of this operation when applied to positive numbers can be summarized:

> `~n -> -|n+1|`

And then, when applied to negative numbers, the corresponding effect is:

> `~-n -> |n-1|`

The following examples illustrate this last rule...

```
# -0 = 0b0000 0000
~-0
# Out: -1
# -1 = 0b1111 1111
# 0 is the obvious exception to this rule, as -0 == 0 always

# -1 = 0b1000 0001
~-1
# Out: 0
# 0 = 0b0000 0000

# -2 = 0b1111 1110
~-2
# Out: 1
# 1 = 0b0000 0001

# -123 = 0b1111 1011
~-123
# Out: 122
# 122 = 0b0111 1010
```

Section 10.2: Bitwise XOR (Exclusive OR)

The `^` operator will perform a binary **XOR** in which a binary `1` is copied if and only if it is the value of exactly **one** operand. Another way of stating this is that the result is `1` only if the operands are different. Examples include:

```
# 0 ^ 0 = 0
# 0 ^ 1 = 1
# 1 ^ 0 = 1
# 1 ^ 1 = 0

# 60 = 0b111100
# 30 = 0b011110
60 ^ 30
# Out: 34
# 34 = 0b100010

bin(60 ^ 30)
```

```
# Out: 0b100010
```

Section 10.3: Bitwise AND

The `&` operator will perform a binary **AND**, where a bit is copied if it exists in **both** operands. That means:

```
# 0 & 0 = 0
# 0 & 1 = 0
# 1 & 0 = 0
# 1 & 1 = 1

# 60 = 0b111100
# 30 = 0b011110
60 & 30
# Out: 28
# 28 = 0b11100

bin(60 & 30)
# Out: 0b11100
```

Section 10.4: Bitwise OR

The `|` operator will perform a binary "or," where a bit is copied if it exists in either operand. That means:

```
# 0 | 0 = 0
# 0 | 1 = 1
# 1 | 0 = 1
# 1 | 1 = 1

# 60 = 0b111100
# 30 = 0b011110
60 | 30
# Out: 62
# 62 = 0b111110

bin(60 | 30)
# Out: 0b111110
```

Section 10.5: Bitwise Left Shift

The `<<` operator will perform a bitwise "left shift," where the left operand's value is moved left by the number of bits given by the right operand.

```
# 2 = 0b10
2 << 2
# Out: 8
# 8 = 0b1000

bin(2 << 2)
# Out: 0b1000
```

Performing a left bit shift of `1` is equivalent to multiplication by `2`:

```
7 << 1
# Out: 14
```

Performing a left bit shift of `n` is equivalent to multiplication by `2**n`:

```
3 << 4
# Out: 48
```

Section 10.6: Bitwise Right Shift

The `>>` operator will perform a bitwise "right shift," where the left operand's value is moved right by the number of bits given by the right operand.

```
# 8 = 0b1000
8 >> 2
# Out: 2
# 2 = 0b10

bin(8 >> 2)
# Out: 0b10
```

Performing a right bit shift of `1` is equivalent to integer division by `2`:

```
36 >> 1
# Out: 18

15 >> 1
# Out: 7
```

Performing a right bit shift of `n` is equivalent to integer division by `2**n`:

```
48 >> 4
# Out: 3

59 >> 3
# Out: 7
```

Section 10.7: Inplace Operations

All of the Bitwise operators (except ~) have their own in place versions

```
a = 0b001
a &= 0b010
# a = 0b000

a = 0b001
a |= 0b010
# a = 0b011

a = 0b001
a <<= 2
# a = 0b100

a = 0b100
a >>= 2
# a = 0b001

a = 0b101
a ^= 0b011
# a = 0b110
```

Chapter 11: Boolean Operators

Section 11.1: `and` and `or` are not guaranteed to return a boolean

When you use `or`, it will either return the first value in the expression if it's true, else it will blindly return the second value. I.e. `or` is equivalent to:

```
def or_(a, b):
    if a:
        return a
    else:
        return b
```

For `and`, it will return its first value if it's false, else it returns the last value:

```
def and_(a, b):
    if not a:
        return a
    else:
        return b
```

Section 11.2: A simple example

In Python you can compare a single element using two binary operators--one on either side:

```
if 3.14 < x < 3.142:
    print("x is near pi")
```

In many (most?) programming languages, this would be evaluated in a way contrary to regular math: `(3.14 < x) < 3.142`, but in Python it is treated like `3.14 < x and x < 3.142`, just like most non-programmers would expect.

Section 11.3: Short-circuit evaluation

Python minimally evaluates Boolean expressions.

```
>>> def true_func():
...     print("true_func()")
...     return True
...
>>> def false_func():
...     print("false_func()")
...     return False
...
>>> true_func() or false_func()
true_func()
True
>>> false_func() or true_func()
false_func()
true_func()
True
>>> true_func() and false_func()
true_func()
false_func()
False
>>> false_func() and false_func()
```

```
false_func()
False
```

Section 11.4: and

Evaluates to the second argument if and only if both of the arguments are truthy. Otherwise evaluates to the first falsey argument.

```
x = True
y = True
z = x and y # z = True

x = True
y = False
z = x and y # z = False

x = False
y = True
z = x and y # z = False

x = False
y = False
z = x and y # z = False

x = 1
y = 1
z = x and y # z = y, so z = 1, see `and` and `or` are not guaranteed to be a boolean

x = 0
y = 1
z = x and y # z = x, so z = 0 (see above)

x = 1
y = 0
z = x and y # z = y, so z = 0 (see above)

x = 0
y = 0
z = x and y # z = x, so z = 0 (see above)
```

The `1`'s in the above example can be changed to any truthy value, and the `0`'s can be changed to any falsey value.

Section 11.5: or

Evaluates to the first truthy argument if either one of the arguments is truthy. If both arguments are falsey, evaluates to the second argument.

```
x = True
y = True
z = x or y # z = True

x = True
y = False
z = x or y # z = True

x = False
y = True
z = x or y # z = True
```

```
x = False
y = False
z = x or y # z = False

x = 1
y = 1
z = x or y # z = x, so z = 1, see `and` and `or` are not guaranteed to be a boolean

x = 1
y = 0
z = x or y # z = x, so z = 1 (see above)

x = 0
y = 1
z = x or y # z = y, so z = 1 (see above)

x = 0
y = 0
z = x or y # z = y, so z = 0 (see above)
```

The `1`'s in the above example can be changed to any truthy value, and the `0`'s can be changed to any falsey value.

Section 11.6: not

It returns the opposite of the following statement:

```
x = True
y = not x # y = False

x = False
y = not x # y = True
```

Chapter 12: Operator Precedence

Python operators have a set **order of precedence**, which determines what operators are evaluated first in a potentially ambiguous expression. For instance, in the expression 3 * 2 + 7, first 3 is multiplied by 2, and then the result is added to 7, yielding 13. The expression is not evaluated the other way around, because * has a higher precedence than +.

Below is a list of operators by precedence, and a brief description of what they (usually) do.

Section 12.1: Simple Operator Precedence Examples in python

Python follows PEMDAS rule. PEMDAS stands for Parentheses, Exponents, Multiplication and Division, and Addition and Subtraction.

Example:

```
>>> a, b, c, d = 2, 3, 5, 7
>>> a ** (b + c)  # parentheses
256
>>> a * b ** c  # exponent: same as `a * (b ** c)`
7776
>>> a + b * c / d  # multiplication / division: same as `a + (b * c / d)`
4.142857142857142
```

Extras: mathematical rules hold, but not always:

```
>>> 300 / 300 * 200
200.0
>>> 300 * 200 / 300
200.0
>>> 1e300 / 1e300 * 1e200
1e+200
>>> 1e300 * 1e200 / 1e300
inf
```

Chapter 13: Variable Scope and Binding

Section 13.1: Nonlocal Variables

Python 3.x `Version ≥ 3.0`

Python 3 added a new keyword called **nonlocal**. The nonlocal keyword adds a scope override to the inner scope. You can read all about it in PEP 3104. This is best illustrated with a couple of code examples. One of the most common examples is to create function that can increment:

```
def counter():
    num = 0
    def incrementer():
        num += 1
        return num
    return incrementer
```

If you try running this code, you will receive an **UnboundLocalError** because the **num** variable is referenced before it is assigned in the innermost function. Let's add nonlocal to the mix:

```
def counter():
    num = 0
    def incrementer():
        nonlocal num
        num += 1
        return num
    return incrementer

c = counter()
c() # = 1
c() # = 2
c() # = 3
```

Basically `nonlocal` will allow you to assign to variables in an outer scope, but not a global scope. So you can't use `nonlocal` in our `counter` function because then it would try to assign to a global scope. Give it a try and you will quickly get a `SyntaxError`. Instead you must use `nonlocal` in a nested function.

(Note that the functionality presented here is better implemented using generators.)

Section 13.2: Global Variables

In Python, variables inside functions are considered local if and only if they appear in the left side of an assignment statement, or some other binding occurrence; otherwise such a binding is looked up in enclosing functions, up to the global scope. This is true even if the assignment statement is never executed.

```
x = 'Hi'

def read_x():
    print(x)   # x is just referenced, therefore assumed global

read_x()       # prints Hi

def read_y():
    print(y)   # here y is just referenced, therefore assumed global
```

```
read_y()       # NameError: global name 'y' is not defined

def read_y():
    y = 'Hey'  # y appears in an assignment, therefore it's local
    print(y)   # will find the local y

read_y()       # prints Hey

def read_x_local_fail():
    if False:
        x = 'Hey'  # x appears in an assignment, therefore it's local
    print(x)   # will look for the _local_ z, which is not assigned, and will not be found

read_x_local_fail()   # UnboundLocalError: local variable 'x' referenced before assignment
```

Normally, an assignment inside a scope will shadow any outer variables of the same name:

```
x = 'Hi'

def change_local_x():
    x = 'Bye'
    print(x)
change_local_x()  # prints Bye
print(x)  # prints Hi
```

Declaring a name `global` means that, for the rest of the scope, any assignments to the name will happen at the module's top level:

```
x = 'Hi'

def change_global_x():
    global x
    x = 'Bye'
    print(x)

change_global_x()  # prints Bye
print(x)  # prints Bye
```

The `global` keyword means that assignments will happen at the module's top level, not at the program's top level. Other modules will still need the usual dotted access to variables within the module.

To summarize: in order to know whether a variable x is local to a function, you should read the *entire* function:

1. if you've found `global x`, then x is a **global** variable
2. If you've found `nonlocal x`, then x belongs to an enclosing function, and is neither local nor global
3. If you've found `x = 5` or `for x in range(3)` or some other binding, then x is a **local** variable
4. Otherwise x belongs to some enclosing scope (function scope, global scope, or builtins)

Section 13.3: Local Variables

If a name is *bound* inside a function, it is by default accessible only within the function:

```
def foo():
    a = 5
    print(a) # ok

print(a) #  NameError: name 'a' is not defined
```

Control flow constructs have no impact on the scope (with the exception of `except`), but accessing variable that was not assigned yet is an error:

```
def foo():
    if True:
        a = 5
    print(a) # ok

b = 3
def bar():
    if False:
        b = 5
    print(b) # UnboundLocalError: local variable 'b' referenced before assignment
```

Common binding operations are assignments, `for` loops, and augmented assignments such as `a += 5`

Section 13.4: The del command

This command has several related yet distinct forms.

`del v`

If `v` is a variable, the command `del v` removes the variable from its scope. For example:

```
x = 5
print(x) # out: 5
del x
print(x) # NameError: name 'f' is not defined
```

> Note that `del` is a *binding occurrence*, which means that unless explicitly stated otherwise (using `nonlocal` or `global`), `del v` will make `v` local to the current scope. If you intend to delete `v` in an outer scope, use `nonlocal v` or `global v` in the same scope of the `del v` statement.

In all the following, the intention of a command is a default behavior but is not enforced by the language. A class might be written in a way that invalidates this intention.

`del v.name`

This command triggers a call to `v.__delattr__(name)`.

The intention is to make the attribute `name` unavailable. For example:

```
class A:
    pass

a = A()
a.x = 7
print(a.x) # out: 7
del a.x
print(a.x) # error: AttributeError: 'A' object has no attribute 'x'
```

`del v[item]`

This command triggers a call to `v.__delitem__(item)`.

The intention is that `item` will not belong in the mapping implemented by the object `v`. For example:

```
x = {'a': 1, 'b': 2}
del x['a']
print(x) #  out: {'b': 2}
print(x['a']) # error: KeyError: 'a'
```

`del v[a:b]`

This actually calls `v.__delslice__(a, b)`.

The intention is similar to the one described above, but with slices - ranges of items instead of a single item. For example:

```
x = [0, 1, 2, 3, 4]
del x[1:3]
print(x) #  out: [0, 3, 4]
```

See also Garbage Collection#The del command.

Section 13.5: Functions skip class scope when looking up names

Classes have a local scope during definition, but functions inside the class do not use that scope when looking up names. Because lambdas are functions, and comprehensions are implemented using function scope, this can lead to some surprising behavior.

```
a = 'global'

class Fred:
    a = 'class'  # class scope
    b = (a for i in range(10))  # function scope
    c = [a for i in range(10)]  # function scope
    d = a  # class scope
    e = lambda: a  # function scope
    f = lambda a=a: a  # default argument uses class scope

    @staticmethod  # or @classmethod, or regular instance method
    def g():  # function scope
        return a

print(Fred.a)  # class
print(next(Fred.b))  # global
print(Fred.c[0])  # class in Python 2, global in Python 3
print(Fred.d)  # class
print(Fred.e())  # global
print(Fred.f())  # class
print(Fred.g()) # global
```

Users unfamiliar with how this scope works might expect `b`, `c`, and `e` to print `class`.

From PEP 227:

> Names in class scope are not accessible. Names are resolved in the innermost enclosing function scope. If a class definition occurs in a chain of nested scopes, the resolution process skips class definitions.

From Python's documentation on naming and binding:

> The scope of names defined in a class block is limited to the class block; it does not extend to the code blocks of methods – this includes comprehensions and generator expressions since they are implemented using a function scope. This means that the following will fail:
>
> ```
> class A:
>     a = 42
>     b = list(a + i for i in range(10))
> ```

This example uses references from this answer by Martijn Pieters, which contains more in depth analysis of this behavior.

Section 13.6: Local vs Global Scope

What are local and global scope?

All Python variables which are accessible at some point in code are either in *local scope* or in *global scope*.

The explanation is that local scope includes all variables defined in the current function and global scope includes variables defined outside of the current function.

```
foo = 1  # global

def func():
    bar = 2  # local
    print(foo)  # prints variable foo from global scope
    print(bar)  # prints variable bar from local scope
```

One can inspect which variables are in which scope. Built-in functions `locals()` and `globals()` return the whole scopes as dictionaries.

```
foo = 1

def func():
    bar = 2
    print(globals().keys())  # prints all variable names in global scope
    print(locals().keys())  # prints all variable names in local scope
```

What happens with name clashes?

```
foo = 1

def func():
    foo = 2  # creates a new variable foo in local scope, global foo is not affected

    print(foo)  # prints 2

    # global variable foo still exists, unchanged:
    print(globals()['foo'])  # prints 1
    print(locals()['foo'])  # prints 2
```

To modify a global variable, use keyword `global`:

```
foo = 1

def func():
    global foo
    foo = 2  # this modifies the global foo, rather than creating a local variable
```

The scope is defined for the whole body of the function!

What it means is that a variable will never be global for a half of the function and local afterwards, or vice-versa.

```
foo = 1

def func():
    # This function has a local variable foo, because it is defined down below.
    # So, foo is local from this point. Global foo is hidden.

    print(foo) # raises UnboundLocalError, because local foo is not yet initialized
    foo = 7
    print(foo)
```

Likewise, the opposite:

```
foo = 1

def func():
    # In this function, foo is a global variable from the beginning

    foo = 7  # global foo is modified

    print(foo)  # 7
    print(globals()['foo'])  # 7

    global foo  # this could be anywhere within the function
    print(foo)  # 7
```

Functions within functions

There may be many levels of functions nested within functions, but within any one function there is only one local scope for that function and the global scope. There are no intermediate scopes.

```
foo = 1

def f1():
    bar = 1

    def f2():
        baz = 2
        # here, foo is a global variable, baz is a local variable
        # bar is not in either scope
        print(locals().keys())  # ['baz']
        print('bar' in locals())  # False
        print('bar' in globals())  # False

    def f3():
        baz = 3
        print(bar)  # bar from f1 is referenced so it enters local scope of f3 (closure)
        print(locals().keys())  # ['bar', 'baz']
        print('bar' in locals())  # True
        print('bar' in globals())  # False

    def f4():
        bar = 4  # a new local bar which hides bar from local scope of f1
        baz = 4
        print(bar)
        print(locals().keys())  # ['bar', 'baz']
        print('bar' in locals())  # True
```

```
        print('bar' in globals())  # False
```

`global` vs `nonlocal` (Python 3 only)

Both these keywords are used to gain write access to variables which are not local to the current functions.

The `global` keyword declares that a name should be treated as a global variable.

```
foo = 0  # global foo

def f1():
    foo = 1  # a new foo local in f1

    def f2():
        foo = 2  # a new foo local in f2

        def f3():
            foo = 3  # a new foo local in f3
            print(foo)  # 3
            foo = 30  # modifies local foo in f3 only

        def f4():
            global foo
            print(foo)  # 0
            foo = 100  # modifies global foo
```

On the other hand, `nonlocal` (see Nonlocal Variables), available in Python 3, takes a *local* variable from an enclosing scope into the local scope of current function.

From the Python documentation on `nonlocal`:

> The nonlocal statement causes the listed identifiers to refer to previously bound variables in the nearest enclosing scope excluding globals.

Python 3.x `Version ≥ 3.0`

```
def f1():

    def f2():
        foo = 2  # a new foo local in f2

        def f3():
            nonlocal foo  # foo from f2, which is the nearest enclosing scope
            print(foo)  # 2
            foo = 20  # modifies foo from f2!
```

Section 13.7: Binding Occurrence

```
x = 5
x += 7
for x in iterable: pass
```

Each of the above statements is a *binding occurrence* - `x` become bound to the object denoted by `5`. If this statement appears inside a function, then `x` will be function-local by default. See the "Syntax" section for a list of binding statements.

Chapter 14: Conditionals

Conditional expressions, involving keywords such as if, elif, and else, provide Python programs with the ability to perform different actions depending on a boolean condition: True or False. This section covers the use of Python conditionals, boolean logic, and ternary statements.

Section 14.1: Conditional Expression (or "The Ternary Operator")

The ternary operator is used for inline conditional expressions. It is best used in simple, concise operations that are easily read.

- The order of the arguments is different from many other languages (such as C, Ruby, Java, etc.), which may lead to bugs when people unfamiliar with Python's "surprising" behaviour use it (they may reverse the order).
- Some find it "unwieldy", since it goes contrary to the normal flow of thought (thinking of the condition first and then the effects).

```
n = 5

"Greater than 2" if n > 2 else "Smaller than or equal to 2"
# Out: 'Greater than 2'
```

The result of this expression will be as it is read in English - if the conditional expression is True, then it will evaluate to the expression on the left side, otherwise, the right side.

Ternary operations can also be nested, as here:

```
n = 5
"Hello" if n > 10 else "Goodbye" if n > 5 else "Good day"
```

They also provide a method of including conditionals in lambda functions.

Section 14.2: if, elif, and else

In Python you can define a series of conditionals using `if` for the first one, `elif` for the rest, up until the final (optional) `else` for anything not caught by the other conditionals.

```
number = 5

if number > 2:
    print("Number is bigger than 2.")
elif number < 2:  # Optional clause (you can have multiple elifs)
    print("Number is smaller than 2.")
else:  # Optional clause (you can only have one else)
    print("Number is 2.")
```

Outputs `Number is bigger than 2`

Using `else if` instead of `elif` will trigger a syntax error and is not allowed.

Section 14.3: Truth Values

The following values are considered falsey, in that they evaluate to `False` when applied to a boolean operator.

- None
- False
- `0`, or any numerical value equivalent to zero, for example `0L`, `0.0`, `0j`
- Empty sequences: `''`, `""`, `()`, `[]`
- Empty mappings: `{}`
- User-defined types where the `__bool__` or `__len__` methods return 0 or `False`

All other values in Python evaluate to `True`.

Note: A common mistake is to simply check for the Falseness of an operation which returns different Falsey values where the difference matters. For example, using `if foo()` rather than the more explicit `if foo() is None`

Section 14.4: Boolean Logic Expressions

Boolean logic expressions, in addition to evaluating to `True` or `False`, return the *value* that was interpreted as `True` or `False`. It is Pythonic way to represent logic that might otherwise require an if-else test.

And operator

The `and` operator evaluates all expressions and returns the last expression if all expressions evaluate to `True`. Otherwise it returns the first value that evaluates to `False`:

```
>>> 1 and 2
2

>>> 1 and 0
0

>>> 1 and "Hello World"
"Hello World"

>>> "" and "Pancakes"
""
```

Or operator

The `or` operator evaluates the expressions left to right and returns the first value that evaluates to `True` or the last value (if none are `True`).

```
>>> 1 or 2
1

>>> None or 1
1

>>> 0 or []
[]
```

Lazy evaluation

When you use this approach, remember that the evaluation is lazy. Expressions that are not required to be evaluated to determine the result are not evaluated. For example:

```
>>> def print_me():
        print('I am here!')
>>> 0 and print_me()
0
```

In the above example, `print_me` is never executed because Python can determine the entire expression is `False` when it encounters the `0` (`False`). Keep this in mind if `print_me` needs to execute to serve your program logic.

Testing for multiple conditions

A common mistake when checking for multiple conditions is to apply the logic incorrectly.

This example is trying to check if two variables are each greater than 2. The statement is evaluated as - `if (a) and (b > 2)`. This produces an unexpected result because `bool(a)` evaluates as `True` when `a` is not zero.

```
>>> a = 1
>>> b = 6
>>> if a and b > 2:
...     print('yes')
... else:
...     print('no')

yes
```

Each variable needs to be compared separately.

```
>>> if a > 2 and b > 2:
...     print('yes')
... else:
...     print('no')

no
```

Another, similar, mistake is made when checking if a variable is one of multiple values. The statement in this example is evaluated as - `if (a == 3) or (4) or (6)`. This produces an unexpected result because `bool(4)` and `bool(6)` each evaluate to `True`

```
>>> a = 1
>>> if a == 3 or 4 or 6:
...     print('yes')
... else:
...     print('no')

yes
```

Again each comparison must be made separately

```
>>> if a == 3 or a == 4 or a == 6:
...     print('yes')
... else:
...     print('no')

no
```

Using the in operator is the canonical way to write this.

```
>>> if a in (3, 4, 6):
...     print('yes')
... else:
...     print('no')

no
```

Section 14.5: Using the cmp function to get the comparison result of two objects

Python 2 includes a `cmp` function which allows you to determine if one object is less than, equal to, or greater than another object. This function can be used to pick a choice out of a list based on one of those three options.

Suppose you need to print `'greater than'` if `x > y`, `'less than'` if `x < y` and `'equal'` if `x == y`.

```
['equal', 'greater than', 'less than', ][cmp(x,y)]

# x,y = 1,1 output: 'equal'
# x,y = 1,2 output: 'less than'
# x,y = 2,1 output: 'greater than'
```

`cmp(x,y)` returns the following values

Comparison	Result
x < y	-1
x == y	0
x > y	1

This function is removed on Python 3. You can use the `cmp_to_key(func)` helper function located in `functools` in Python 3 to convert old comparison functions to key functions.

Section 14.6: Else statement

```
if condition:
    body
else:
    body
```

The else statement will execute it's body only if preceding conditional statements all evaluate to False.

```
if True:
    print "It is true!"
else:
    print "This won't get printed.."

# Output: It is true!

if False:
    print "This won't get printed.."
else:
    print "It is false!"

# Output: It is false!
```

Section 14.7: Testing if an object is None and assigning it

You'll often want to assign something to an object if it is `None`, indicating it has not been assigned. We'll use `aDate`.

The simplest way to do this is to use the `is None` test.

```
if aDate is None:
    aDate=datetime.date.today()
```

(Note that it is more Pythonic to say `is None` instead of `== None`.)

But this can be optimized slightly by exploiting the notion that `not None` will evaluate to `True` in a boolean expression. The following code is equivalent:

```
if not aDate:
    aDate=datetime.date.today()
```

But there is a more Pythonic way. The following code is also equivalent:

```
aDate=aDate or datetime.date.today()
```

This does a Short Circuit evaluation. If `aDate` is initialized and is `not None`, then it gets assigned to itself with no net effect. If it `is None`, then the `datetime.date.today()` gets assigned to `aDate`.

Section 14.8: If statement

```
if condition:
    body
```

The `if` statements checks the condition. If it evaluates to `True`, it executes the body of the `if` statement. If it evaluates to `False`, it skips the body.

```
if True:
    print "It is true!"
>> It is true!

if False:
    print "This won't get printed.."
```

The condition can be any valid expression:

```
if 2 + 2 == 4:
    print "I know math!"
>> I know math!
```

Chapter 15: Comparisons

Parameter	Details
x	First item to be compared
y	Second item to be compared

Section 15.1: Chain Comparisons

You can compare multiple items with multiple comparison operators with chain comparison. For example

```
x > y > z
```

is just a short form of:

```
x > y and y > z
```

This will evaluate to `True` only if both comparisons are `True`.

The general form is

```
a OP b OP c OP d ...
```

Where `OP` represents one of the multiple comparison operations you can use, and the letters represent arbitrary valid expressions.

> Note that `0 != 1 != 0` evaluates to `True`, even though `0 != 0` is `False`. Unlike the common mathematical notation in which `x != y != z` means that `x`, `y` and `z` have different values. Chaining `==` operations has the natural meaning in most cases, since equality is generally transitive.

Style

There is no theoretical limit on how many items and comparison operations you use as long you have proper syntax:

```
1 > -1 < 2 > 0.5 < 100 != 24
```

The above returns `True` if each comparison returns `True`. However, using convoluted chaining is not a good style. A good chaining will be "directional", not more complicated than

```
1 > x > -4 > y != 8
```

Side effects

As soon as one comparison returns `False`, the expression evaluates immediately to `False`, skipping all remaining comparisons.

Note that the expression `exp` in `a > exp > b` will be evaluated only once, whereas in the case of

```
a > exp and exp > b
```

`exp` will be computed twice if `a > exp` is true.

Section 15.2: Comparison by `is` vs `==`

A common pitfall is confusing the equality comparison operators `is` and `==`.

`a == b` compares the value of `a` and `b`.

`a is b` will compare the *identities* of `a` and `b`.

To illustrate:

```
a = 'Python is fun!'
b = 'Python is fun!'
a == b # returns True
a is b # returns False

a = [1, 2, 3, 4, 5]
b = a      # b references a
a == b     # True
a is b     # True
b = a[:]   # b now references a copy of a
a == b     # True
a is b     # False [!!]
```

Basically, `is` can be thought of as shorthand for `id(a) == id(b)`.

Beyond this, there are quirks of the run-time environment that further complicate things. Short strings and small integers will return `True` when compared with `is`, due to the Python machine attempting to use less memory for identical objects.

```
a = 'short'
b = 'short'
c = 5
d = 5
a is b # True
c is d # True
```

But longer strings and larger integers will be stored separately.

```
a = 'not so short'
b = 'not so short'
c = 1000
d = 1000
a is b # False
c is d # False
```

You should use `is` to test for `None`:

```
if myvar is not None:
    # not None
    pass
if myvar is None:
    # None
    pass
```

A use of `is` is to test for a “sentinel” (i.e. a unique object).

```
sentinel = object()
def myfunc(var=sentinel):
```

```
    if var is sentinel:
        # value wasn't provided
        pass
    else:
        # value was provided
        pass
```

Section 15.3: Greater than or less than

```
x > y
x < y
```

These operators compare two types of values, they're the less than and greater than operators. For numbers this simply compares the numerical values to see which is larger:

```
12 > 4
# True
12 < 4
# False
1 < 4
# True
```

For strings they will compare lexicographically, which is similar to alphabetical order but not quite the same.

```
"alpha" < "beta"
# True
"gamma" > "beta"
# True
"gamma" < "OMEGA"
# False
```

In these comparisons, lowercase letters are considered 'greater than' uppercase, which is why `"gamma" < "OMEGA"` is false. If they were all uppercase it would return the expected alphabetical ordering result:

```
"GAMMA" < "OMEGA"
# True
```

Each type defines it's calculation with the `<` and `>` operators differently, so you should investigate what the operators mean with a given type before using it.

Section 15.4: Not equal to

```
x != y
```

This returns `True` if `x` and `y` are not equal and otherwise returns `False`.

```
12 != 1
# True
12 != '12'
# True
'12' != '12'
# False
```

Section 15.5: Equal To

```
x == y
```

This expression evaluates if `x` and `y` are the same value and returns the result as a boolean value. Generally both type and value need to match, so the int `12` is not the same as the string `'12'`.

```
12 == 12
# True
12 == 1
# False
'12' == '12'
# True
'spam' == 'spam'
# True
'spam' == 'spam '
# False
'12' == 12
# False
```

Note that each type has to define a function that will be used to evaluate if two values are the same. For builtin types these functions behave as you'd expect, and just evaluate things based on being the same value. However custom types could define equality testing as whatever they'd like, including always returning `True` or always returning `False`.

Section 15.6: Comparing Objects

In order to compare the equality of custom classes, you can override `==` and `!=` by defining `__eq__` and `__ne__` methods. You can also override `__lt__` (`<`), `__le__` (`<=`), `__gt__` (`>`), and `__ge__` (`>`). Note that you only need to override two comparison methods, and Python can handle the rest (`==` is the same as `not <` and `not >`, etc.)

```
class Foo(object):
    def __init__(self, item):
        self.my_item = item
    def __eq__(self, other):
        return self.my_item == other.my_item

a = Foo(5)
b = Foo(5)
a == b     # True
a != b     # False
a is b     # False
```

Note that this simple comparison assumes that `other` (the object being compared to) is the same object type. Comparing to another type will throw an error:

```
class Bar(object):
    def __init__(self, item):
        self.other_item = item
    def __eq__(self, other):
        return self.other_item == other.other_item
    def __ne__(self, other):
        return self.other_item != other.other_item

c = Bar(5)
a == c    # throws AttributeError: 'Foo' object has no attribute 'other_item'
```

Checking `isinstance()` or similar will help prevent this (if desired).

Chapter 16: Loops

Parameter	Details
boolean expression	expression that can be evaluated in a boolean context, e.g. `x < 10`
variable	variable name for the current element from the `iterable`
iterable	anything that implements iterations

As one of the most basic functions in programming, loops are an important piece to nearly every programming language. Loops enable developers to set certain portions of their code to repeat through a number of loops which are referred to as iterations. This topic covers using multiple types of loops and applications of loops in Python.

Section 16.1: Break and Continue in Loops

`break` statement

When a `break` statement executes inside a loop, control flow "breaks" out of the loop immediately:

```
i = 0
while i < 7:
    print(i)
    if i == 4:
        print("Breaking from loop")
        break
    i += 1
```

The loop conditional will not be evaluated after the `break` statement is executed. Note that `break` statements are only allowed *inside loops*, syntactically. A `break` statement inside a function cannot be used to terminate loops that called that function.

Executing the following prints every digit until number `4` when the `break` statement is met and the loop stops:

```
0
1
2
3
4
Breaking from loop
```

`break` statements can also be used inside `for` loops, the other looping construct provided by Python:

```
for i in (0, 1, 2, 3, 4):
    print(i)
    if i == 2:
        break
```

Executing this loop now prints:

```
0
1
2
```

Note that 3 and 4 are not printed since the loop has ended.

If a loop has an `else` clause, it does not execute when the loop is terminated through a `break` statement.

`continue` statement

A `continue` statement will skip to the next iteration of the loop bypassing the rest of the current block but continuing the loop. As with `break`, `continue` can only appear inside loops:

```
for i in (0, 1, 2, 3, 4, 5):
    if i == 2 or i == 4:
        continue
    print(i)

0
1
3
5
```

Note that `2` and `4` aren't printed, this is because `continue` goes to the next iteration instead of continuing on to `print(i)` when `i == 2` or `i == 4`.

Nested Loops

`break` and `continue` only operate on a single level of loop. The following example will only break out of the inner `for` loop, not the outer `while` loop:

```
while True:
    for i in range(1,5):
        if i == 2:
            break    # Will only break out of the inner loop!
```

Python doesn't have the ability to break out of multiple levels of loop at once -- if this behavior is desired, refactoring one or more loops into a function and replacing `break` with `return` may be the way to go.

Use `return` from within a function as a `break`

The `return` statement exits from a function, without executing the code that comes after it.

If you have a loop inside a function, using `return` from inside that loop is equivalent to having a `break` as the rest of the code of the loop is not executed (*note that any code after the loop is not executed either*):

```
def break_loop():
    for i in range(1, 5):
        if (i == 2):
            return(i)
        print(i)
    return(5)
```

If you have nested loops, the `return` statement will break all loops:

```
def break_all():
    for j in range(1, 5):
        for i in range(1,4):
            if i*j == 6:
                return(i)
            print(i*j)
```

will output:

```
1 # 1*1
2 # 1*2
3 # 1*3
4 # 1*4
2 # 2*1
4 # 2*2
# return because 2*3 = 6, the remaining iterations of both loops are not executed
```

Section 16.2: For loops

`for` loops iterate over a collection of items, such as `list` or `dict`, and run a block of code with each element from the collection.

```
for i in [0, 1, 2, 3, 4]:
    print(i)
```

The above `for` loop iterates over a list of numbers.

Each iteration sets the value of `i` to the next element of the list. So first it will be `0`, then `1`, then `2`, etc. The output will be as follow:

```
0
1
2
3
4
```

`range` is a function that returns a series of numbers under an iterable form, thus it can be used in `for` loops:

```
for i in range(5):
    print(i)
```

gives the exact same result as the first `for` loop. Note that `5` is not printed as the range here is the first five numbers counting from `0`.

Iterable objects and iterators

`for` loop can iterate on any iterable object which is an object which defines a `__getitem__` or a `__iter__` function. The `__iter__` function returns an iterator, which is an object with a `next` function that is used to access the next element of the iterable.

Section 16.3: Iterating over lists

To iterate through a list you can use `for`:

```
for x in ['one', 'two', 'three', 'four']:
    print(x)
```

This will print out the elements of the list:

```
one
two
three
four
```

The `range` function generates numbers which are also often used in a for loop.

```
for x in range(1, 6):
    print(x)
```

The result will be a special range sequence type in python >=3 and a list in python <=2. Both can be looped through using the for loop.

```
1
2
3
4
5
```

If you want to loop though both the elements of a list *and* have an index for the elements as well, you can use Python's `enumerate` function:

```
for index, item in enumerate(['one', 'two', 'three', 'four']):
    print(index, '::', item)
```

`enumerate` will generate tuples, which are unpacked into `index` (an integer) and `item` (the actual value from the list). The above loop will print

```
(0, '::', 'one')
(1, '::', 'two')
(2, '::', 'three')
(3, '::', 'four')
```

Iterate over a list with value manipulation using `map` and `lambda`, i.e. apply lambda function on each element in the list:

```
x = map(lambda e :  e.upper(), ['one', 'two', 'three', 'four'])
print(x)
```

Output:

```
['ONE', 'TWO', 'THREE', 'FOUR'] # Python 2.x
```

NB: in Python 3.x `map` returns an iterator instead of a list so you in case you need a list you have to cast the result `print(list(x))`

Section 16.4: Loops with an "else" clause

The `for` and `while` compound statements (loops) can optionally have an `else` clause (in practice, this usage is fairly rare).

The `else` clause only executes after a `for` loop terminates by iterating to completion, or after a `while` loop terminates by its conditional expression becoming false.

```
for i in range(3):
    print(i)
else:
    print('done')

i = 0
```

```
while i < 3:
    print(i)
    i += 1
else:
    print('done')
```

output:

```
0
1
2
done
```

The `else` clause does *not* execute if the loop terminates some other way (through a `break` statement or by raising an exception):

```
for i in range(2):
    print(i)
    if i == 1:
        break
else:
    print('done')
```

output:

```
0
1
```

Most other programming languages lack this optional `else` clause of loops. The use of the keyword `else` in particular is often considered confusing.

The original concept for such a clause dates back to Donald Knuth and the meaning of the `else` keyword becomes clear if we rewrite a loop in terms of `if` statements and `goto` statements from earlier days before structured programming or from a lower-level assembly language.

For example:

```
while loop_condition():
    ...
    if break_condition():
        break
    ...
```

is equivalent to:

```
# pseudocode

<<start>>:
if loop_condition():
    ...
    if break_condition():
        goto <<end>>
    ...
    goto <<start>>
```

```
<<end>>:
```

These remain equivalent if we attach an `else` clause to each of them.

For example:

```
while loop_condition():
    ...
    if break_condition():
        break
    ...
else:
    print('done')
```

is equivalent to:

```
# pseudocode

<<start>>:
if loop_condition():
    ...
    if break_condition():
        goto <<end>>
    ...
    goto <<start>>
else:
    print('done')

<<end>>:
```

A `for` loop with an `else` clause can be understood the same way. Conceptually, there is a loop condition that remains True as long as the iterable object or sequence still has some remaining elements.

Why would one use this strange construct?

The main use case for the `for...else` construct is a concise implementation of search as for instance:

```
a = [1, 2, 3, 4]
for i in a:
    if type(i) is not int:
        print(i)
        break
else:
    print("no exception")
```

To make the `else` in this construct less confusing one can think of it as "*if not break*" or "*if not found*".

Some discussions on this can be found in [Python-ideas] Summary of for...else threads, Why does python use 'else' after for and while loops? , and Else Clauses on Loop Statements

Section 16.5: The Pass Statement

`pass` is a null statement for when a statement is required by Python syntax (such as within the body of a `for` or `while` loop), but no action is required or desired by the programmer. This can be useful as a placeholder for code that is yet to be written.

```
for x in range(10):
```

```
    pass #we don't want to do anything, or are not ready to do anything here, so we'll pass
```

In this example, nothing will happen. The `for` loop will complete without error, but no commands or code will be actioned. `pass` allows us to run our code successfully without having all commands and action fully implemented.

Similarly, `pass` can be used in `while` loops, as well as in selections and function definitions etc.

```
while x == y:
    pass
```

Section 16.6: Iterating over dictionaries

Considering the following dictionary:

```
d = {"a": 1, "b": 2, "c": 3}
```

To iterate through its keys, you can use:

```
for key in d:
    print(key)
```

Output:

```
"a"
"b"
"c"
```

This is equivalent to:

```
for key in d.keys():
    print(key)
```

or in Python 2:

```
for key in d.iterkeys():
    print(key)
```

To iterate through its values, use:

```
for value in d.values():
    print(value)
```

Output:

```
1
2
3
```

To iterate through its keys and values, use:

```
for key, value in d.items():
    print(key, "::", value)
```

Output:

```
a :: 1
b :: 2
c :: 3
```

Note that in Python 2, `.keys()`, `.values()` and `.items()` return a `list` object. If you simply need to iterate through the result, you can use the equivalent `.iterkeys()`, `.itervalues()` and `.iteritems()`.

The difference between `.keys()` and `.iterkeys()`, `.values()` and `.itervalues()`, `.items()` and `.iteritems()` is that the `iter*` methods are generators. Thus, the elements within the dictionary are yielded one by one as they are evaluated. When a `list` object is returned, all of the elements are packed into a list and then returned for further evaluation.

> Note also that in Python 3, Order of items printed in the above manner does not follow any order.

Section 16.7: The "half loop" do-while

Unlike other languages, Python doesn't have a do-until or a do-while construct (this will allow code to be executed once before the condition is tested). However, you can combine a `while True` with a `break` to achieve the same purpose.

```
a = 10
while True:
    a = a-1
    print(a)
    if a<7:
        break
print('Done.')
```

This will print:

```
9
8
7
6
Done.
```

Section 16.8: Looping and Unpacking

If you want to loop over a list of tuples for example:

```
collection = [('a', 'b', 'c'), ('x', 'y', 'z'), ('1', '2', '3')]
```

instead of doing something like this:

```
for item in collection:
    i1 = item[0]
    i2 = item[1]
    i3 = item[2]
    # logic
```

or something like this:

```
for item in collection:
```

```
    i1, i2, i3 = item
    # logic
```

You can simply do this:

```
for i1, i2, i3 in collection:
    # logic
```

This will also work for *most* types of iterables, not just tuples.

Section 16.9: Iterating different portion of a list with different step size

Suppose you have a long list of elements and you are only interested in every other element of the list. Perhaps you only want to examine the first or last elements, or a specific range of entries in your list. Python has strong indexing built-in capabilities. Here are some examples of how to achieve these scenarios.

Here's a simple list that will be used throughout the examples:

```
lst = ['alpha', 'bravo', 'charlie', 'delta', 'echo']
```

Iteration over the whole list

To iterate over each element in the list, a `for` loop like below can be used:

```
for s in lst:
    print s[:1] # print the first letter
```

The `for` loop assigns s for each element of `lst`. This will print:

```
a
b
c
d
e
```

Often you need both the element and the index of that element. The `enumerate` keyword performs that task.

```
for idx, s in enumerate(lst):
    print("%s has an index of %d" % (s, idx))
```

The index `idx` will start with zero and increment for each iteration, while the `s` will contain the element being processed. The previous snippet will output:

```
alpha has an index of 0
bravo has an index of 1
charlie has an index of 2
delta has an index of 3
echo has an index of 4
```

Iterate over sub-list

If we want to iterate over a range (remembering that Python uses zero-based indexing), use the `range` keyword.

```
for i in range(2,4):
    print("lst at %d contains %s" % (i, lst[i]))
```

This would output:

```
lst at 2 contains charlie
lst at 3 contains delta
```

The list may also be sliced. The following slice notation goes from element at index 1 to the end with a step of 2. The two `for` loops give the same result.

```
for s in lst[1::2]:
    print(s)

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

The above snippet outputs:

```
bravo
delta
```

Indexing and slicing is a topic of its own.

Section 16.10: While Loop

A `while` loop will cause the loop statements to be executed until the loop condition is falsey. The following code will execute the loop statements a total of 4 times.

```
i = 0
while i < 4:
    #loop statements
    i = i + 1
```

While the above loop can easily be translated into a more elegant `for` loop, `while` loops are useful for checking if some condition has been met. The following loop will continue to execute until `myObject` is ready.

```
myObject = anObject()
while myObject.isNotReady():
    myObject.tryToGetReady()
```

`while` loops can also run without a condition by using numbers (complex or real) or `True`:

```
import cmath

complex_num = cmath.sqrt(-1)
while complex_num:      # You can also replace complex_num with any number, True or a value of any
type
    print(complex_num)   # Prints 1j forever
```

If the condition is always true the while loop will run forever (infinite loop) if it is not terminated by a break or return statement or an exception.

```
while True:
    print "Infinite loop"
```

```
# Infinite loop
# Infinite loop
# Infinite loop
# ...
```

Chapter 17: Arrays

Parameter	Details
`b`	Represents signed integer of size 1 byte
`B`	Represents unsigned integer of size 1 byte
`c`	Represents character of size 1 byte
`u`	Represents unicode character of size 2 bytes
`h`	Represents signed integer of size 2 bytes
`H`	Represents unsigned integer of size 2 bytes
`i`	Represents signed integer of size 2 bytes
`I`	Represents unsigned integer of size 2 bytes
`w`	Represents unicode character of size 4 bytes
`l`	Represents signed integer of size 4 bytes
`L`	Represents unsigned integer of size 4 bytes
`f`	Represents floating point of size 4 bytes
`d`	Represents floating point of size 8 bytes

"Arrays" in Python are not the arrays in conventional programming languages like C and Java, but closer to lists. A list can be a collection of either homogeneous or heterogeneous elements, and may contain ints, strings or other lists.

Section 17.1: Access individual elements through indexes

Individual elements can be accessed through indexes. Python arrays are zero-indexed. Here is an example:

```
my_array = array('i', [1,2,3,4,5])
print(my_array[1])
# 2
print(my_array[2])
# 3
print(my_array[0])
# 1
```

Section 17.2: Basic Introduction to Arrays

An array is a data structure that stores values of same data type. In Python, this is the main difference between arrays and lists.

While python lists can contain values corresponding to different data types, arrays in python can only contain values corresponding to same data type. In this tutorial, we will understand the Python arrays with few examples.

If you are new to Python, get started with the Python Introduction article.

To use arrays in python language, you need to import the standard `array` module. This is because array is not a fundamental data type like strings, integer etc. Here is how you can import `array` module in python :

```
from array import *
```

Once you have imported the `array` module, you can declare an array. Here is how you do it:

```
arrayIdentifierName = array(typecode, [Initializers])
```

In the declaration above, `arrayIdentifierName` is the name of array, `typecode` lets python know the type of array and `Initializers` are the values with which array is initialized.

Typecodes are the codes that are used to define the type of array values or the type of array. The table in the parameters section shows the possible values you can use when declaring an array and it's type.

Here is a real world example of python array declaration :

```
my_array = array('i',[1,2,3,4])
```

In the example above, typecode used is `i`. This typecode represents signed integer whose size is 2 bytes.

Here is a simple example of an array containing 5 integers

```
from array import *
my_array = array('i', [1,2,3,4,5])
for i in my_array:
    print(i)
# 1
# 2
# 3
# 4
# 5
```

Section 17.3: Append any value to the array using append() method

```
my_array = array('i', [1,2,3,4,5])
my_array.append(6)
# array('i', [1, 2, 3, 4, 5, 6])
```

Note that the value `6` was appended to the existing array values.

Section 17.4: Insert value in an array using insert() method

We can use the `insert()` method to insert a value at any index of the array. Here is an example :

```
my_array = array('i', [1,2,3,4,5])
my_array.insert(0,0)
#array('i', [0, 1, 2, 3, 4, 5])
```

In the above example, the value 0 was inserted at index 0. Note that the first argument is the index while second argument is the value.

Section 17.5: Extend python array using extend() method

A python array can be extended with more than one value using `extend()` method. Here is an example :

```
my_array = array('i', [1,2,3,4,5])
my_extnd_array = array('i', [7,8,9,10])
my_array.extend(my_extnd_array)
# array('i', [1, 2, 3, 4, 5, 7, 8, 9, 10])
```

We see that the array my_array was extended with values from `my_extnd_array`.

Section 17.6: Add items from list into array using fromlist() method

Here is an example:

```
my_array = array('i', [1,2,3,4,5])
c=[11,12,13]
my_array.fromlist(c)
# array('i', [1, 2, 3, 4, 5, 11, 12, 13])
```

So we see that the values 11,12 and 13 were added from list `c` to `my_array`.

Section 17.7: Remove any array element using remove() method

Here is an example :

```
my_array = array('i', [1,2,3,4,5])
my_array.remove(4)
# array('i', [1, 2, 3, 5])
```

We see that the element 4 was removed from the array.

Section 17.8: Remove last array element using pop() method

`pop` removes the last element from the array. Here is an example :

```
my_array = array('i', [1,2,3,4,5])
my_array.pop()
# array('i', [1, 2, 3, 4])
```

So we see that the last element (5) was popped out of array.

Section 17.9: Fetch any element through its index using index() method

`index()` returns first index of the matching value. Remember that arrays are zero-indexed.

```
my_array = array('i', [1,2,3,4,5])
print(my_array.index(5))
# 5
my_array = array('i', [1,2,3,3,5])
print(my_array.index(3))
# 3
```

Note in that second example that only one index was returned, even though the value exists twice in the array

Section 17.10: Reverse a python array using reverse() method

The `reverse()` method does what the name says it will do - reverses the array. Here is an example :

```
my_array = array('i', [1,2,3,4,5])
my_array.reverse()
# array('i', [5, 4, 3, 2, 1])
```

Section 17.11: Get array buffer information through buffer_info() method

This method provides you the array buffer start address in memory and number of elements in array. Here is an example:

```
my_array = array('i', [1,2,3,4,5])
my_array.buffer_info()
(33881712, 5)
```

Section 17.12: Check for number of occurrences of an element using count() method

`count()` will return the number of times and element appears in an array. In the following example we see that the value `3` occurs twice.

```
my_array = array('i', [1,2,3,3,5])
my_array.count(3)
# 2
```

Section 17.13: Convert array to string using tostring() method

`tostring()` converts the array to a string.

```
my_char_array = array('c', ['g','e','e','k'])
# array('c', 'geek')
print(my_char_array.tostring())
# geek
```

Section 17.14: Convert array to a python list with same elements using tolist() method

When you need a Python `list` object, you can utilize the `tolist()` method to convert your array to a list.

```
my_array = array('i', [1,2,3,4,5])
c = my_array.tolist()
# [1, 2, 3, 4, 5]
```

Section 17.15: Append a string to char array using fromstring() method

You are able to append a string to a character array using `fromstring()`

```
my_char_array = array('c', ['g','e','e','k'])
my_char_array.fromstring("stuff")
print(my_char_array)
#array('c', 'geekstuff')
```

Chapter 18: Multidimensional arrays

Section 18.1: Lists in lists

A good way to visualize a 2d array is as a list of lists. Something like this:

```
lst=[[1,2,3],[4,5,6],[7,8,9]]
```

here the outer list `lst` has three things in it. each of those things is another list: The first one is: `[1,2,3]`, the second one is: `[4,5,6]` and the third one is: `[7,8,9]`. You can access these lists the same way you would access another other element of a list, like this:

```
print (lst[0])
#output: [1, 2, 3]

print (lst[1])
#output: [4, 5, 6]

print (lst[2])
#output: [7, 8, 9]
```

You can then access the different elements in each of those lists the same way:

```
print (lst[0][0])
#output: 1

print (lst[0][1])
#output: 2
```

Here the first number inside the `[]` brackets means get the list in that position. In the above example we used the number `0` to mean get the list in the 0th position which is `[1,2,3]`. The second set of `[]` brackets means get the item in that position from the inner list. In this case we used both `0` and `1` the 0th position in the list we got is the number `1` and in the 1st position it is `2`

You can also set values inside these lists the same way:

```
lst[0]=[10,11,12]
```

Now the list is `[[10,11,12],[4,5,6],[7,8,9]]`. In this example we changed the whole first list to be a completely new list.

```
lst[1][2]=15
```

Now the list is `[[10,11,12],[4,5,15],[7,8,9]]`. In this example we changed a single element inside of one of the inner lists. First we went into the list at position 1 and changed the element within it at position 2, which was 6 now it's 15.

Section 18.2: Lists in lists in lists in..

This behaviour can be extended. Here is a 3-dimensional array:

```
[[[111,112,113],[121,122,123],[131,132,133]],[[211,212,213],[221,222,223],[231,232,233]],[[311,312,
313],[321,322,323],[331,332,333]]]
```

As is probably obvious, this gets a bit hard to read. Use backslashes to break up the different dimensions:

```
[[[111,112,113],[121,122,123],[131,132,133]],\
 [[211,212,213],[221,222,223],[231,232,233]],\
 [[311,312,313],[321,322,323],[331,332,333]]]
```

By nesting the lists like this, you can extend to arbitrarily high dimensions.

Accessing is similar to 2D arrays:

```
print(myarray)
print(myarray[1])
print(myarray[2][1])
print(myarray[1][0][2])
etc.
```

And editing is also similar:

```
myarray[1]=new_n-1_d_list
myarray[2][1]=new_n-2_d_list
myarray[1][0][2]=new_n-3_d_list #or a single number if you're dealing with 3D arrays
etc.
```

Chapter 19: Dictionary

Parameter	Details
key	The desired key to lookup
value	The value to set or return

Section 19.1: Introduction to Dictionary

A dictionary is an example of a *key value store* also known as *Mapping* in Python. It allows you to store and retrieve elements by referencing a key. As dictionaries are referenced by key, they have very fast lookups. As they are primarily used for referencing items by key, they are not sorted.

creating a dict

Dictionaries can be initiated in many ways:

literal syntax

```
d = {}                          # empty dict
d = {'key': 'value'}            # dict with initial values
```

Python 3.x `Version ≥ 3.5`

```
# Also unpacking one or multiple dictionaries with the literal syntax is possible

# makes a shallow copy of otherdict
d = {**otherdict}
# also updates the shallow copy with the contents of the yetanotherdict.
d = {**otherdict, **yetanotherdict}
```

dict comprehension

```
d = {k:v for k,v in [('key', 'value',)]}
```

see also: Comprehensions

built-in class: `dict()`

```
d = dict()                      # empty dict
d = dict(key='value')           # explicit keyword arguments
d = dict([('key', 'value')])  # passing in a list of key/value pairs
# make a shallow copy of another dict (only possible if keys are only strings!)
d = dict(**otherdict)
```

modifying a dict

To add items to a dictionary, simply create a new key with a value:

```
d['newkey'] = 42
```

It also possible to add `list` and `dictionary` as value:

```
d['new_list'] = [1, 2, 3]
d['new_dict'] = {'nested_dict': 1}
```

To delete an item, delete the key from the dictionary:

```
del d['newkey']
```

Section 19.2: Avoiding KeyError Exceptions

One common pitfall when using dictionaries is to access a non-existent key. This typically results in a `KeyError` exception

```
mydict = {}
mydict['not there']
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'not there'
```

One way to avoid key errors is to use the `dict.get` method, which allows you to specify a default value to return in the case of an absent key.

```
value = mydict.get(key, default_value)
```

Which returns `mydict[key]` if it exists, but otherwise returns `default_value`. Note that this doesn't add `key` to `mydict`. So if you want to retain that key value pair, you should use `mydict.setdefault(key, default_value)`, which *does* store the key value pair.

```
mydict = {}
print(mydict)
# {}
print(mydict.get("foo", "bar"))
# bar
print(mydict)
# {}
print(mydict.setdefault("foo", "bar"))
# bar
print(mydict)
# {'foo': 'bar'}
```

An alternative way to deal with the problem is catching the exception

```
try:
    value = mydict[key]
except KeyError:
    value = default_value
```

You could also check if the key is `in` the dictionary.

```
if key in mydict:
    value = mydict[key]
else:
    value = default_value
```

Do note, however, that in multi-threaded environments it is possible for the key to be removed from the dictionary after you check, creating a race condition where the exception can still be thrown.

Another option is to use a subclass of dict, collections.defaultdict, that has a default_factory to create new entries in the dict when given a new_key.

Section 19.3: Iterating Over a Dictionary

If you use a dictionary as an iterator (e.g. in a `for` statement), it traverses the **keys** of the dictionary. For example:

```
d = {'a': 1, 'b': 2, 'c':3}
for key in d:
    print(key, d[key])
# c 3
# b 2
# a 1
```

The same is true when used in a comprehension

```
print([key for key in d])
# ['c', 'b', 'a']
```

Python 3.x `Version ≥ 3.0`

The `items()` method can be used to loop over both the **key** and **value** simultaneously:

```
for key, value in d.items():
    print(key, value)
# c 3
# b 2
# a 1
```

While the `values()` method can be used to iterate over only the values, as would be expected:

```
for key, value in d.values():
    print(key, value)
    # 3
    # 2
    # 1
```

Python 2.x `Version ≥ 2.2`

Here, the methods `keys()`, `values()` and `items()` return lists, and there are the three extra methods `iterkeys()` `itervalues()` and `iteritems()` to return iterators.

Section 19.4: Dictionary with default values

Available in the standard library as `defaultdict`

```
from collections import defaultdict

d = defaultdict(int)
d['key']                        # 0
d['key'] = 5
d['key']                        # 5

d = defaultdict(lambda: 'empty')
d['key']                        # 'empty'
d['key'] = 'full'
d['key']                        # 'full'
```

[*] Alternatively, if you must use the built-in `dict` class, `using dict.setdefault()` will allow you to create a default whenever you access a key that did not exist before:

```
>>> d = {}
{}
>>> d.setdefault('Another_key', []).append("This worked!")
>>> d
```

```
{'Another_key': ['This worked!']}
```

Keep in mind that if you have many values to add, `dict.setdefault()` will create a new instance of the initial value (in this example a `[]`) every time it's called - which may create unnecessary workloads.

[*] *Python Cookbook, 3rd edition, by David Beazley and Brian K. Jones (O'Reilly). Copyright 2013 David Beazley and Brian Jones, 978-1-449-34037-7.*

Section 19.5: Merging dictionaries

Consider the following dictionaries:

```
>>> fish = {'name': "Nemo", 'hands': "fins", 'special': "gills"}
>>> dog = {'name': "Clifford", 'hands': "paws", 'color': "red"}
```

Python 3.5+

```
>>> fishdog = {**fish, **dog}
>>> fishdog
{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

As this example demonstrates, duplicate keys map to their lattermost value (for example "Clifford" overrides "Nemo").

Python 3.3+

```
>>> from collections import ChainMap
>>> dict(ChainMap(fish, dog))
{'hands': 'fins', 'color': 'red', 'special': 'gills', 'name': 'Nemo'}
```

With this technique the foremost value takes precedence for a given key rather than the last ("Clifford" is thrown out in favor of "Nemo").

Python 2.x, 3.x

```
>>> from itertools import chain
>>> dict(chain(fish.items(), dog.items()))
{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

This uses the lattermost value, as with the `**`-based technique for merging ("Clifford" overrides "Nemo").

```
>>> fish.update(dog)
>>> fish
{'color': 'red', 'hands': 'paws', 'name': 'Clifford', 'special': 'gills'}
```

`dict.update` uses the latter dict to overwrite the previous one.

Section 19.6: Accessing keys and values

When working with dictionaries, it's often necessary to access all the keys and values in the dictionary, either in a `for` loop, a list comprehension, or just as a plain list.

Given a dictionary like:

```
mydict = {
    'a': '1',
```

```
    'b': '2'
}
```

You can get a list of keys using the `keys()` method:

```
print(mydict.keys())
# Python2: ['a', 'b']
# Python3: dict_keys(['b', 'a'])
```

If instead you want a list of values, use the `values()` method:

```
print(mydict.values())
# Python2: ['1', '2']
# Python3: dict_values(['2', '1'])
```

If you want to work with both the key and its corresponding value, you can use the `items()` method:

```
print(mydict.items())
# Python2: [('a', '1'), ('b', '2')]
# Python3: dict_items([('b', '2'), ('a', '1')])
```

NOTE: Because a `dict` is unsorted, `keys()`, `values()`, and `items()` have no sort order. Use `sort()`, `sorted()`, or an `OrderedDict` if you care about the order that these methods return.

Python 2/3 Difference: In Python 3, these methods return special iterable objects, not lists, and are the equivalent of the Python 2 `iterkeys()`, `itervalues()`, and `iteritems()` methods. These objects can be used like lists for the most part, though there are some differences. See PEP 3106 for more details.

Section 19.7: Accessing values of a dictionary

```
dictionary = {"Hello": 1234, "World": 5678}
print(dictionary["Hello"])
```

The above code will print `1234`.

The string `"Hello"` in this example is called a *key*. It is used to lookup a value in the `dict` by placing the key in square brackets.

The number `1234` is seen after the respective colon in the `dict` definition. This is called the *value* that `"Hello"` *maps to* in this `dict`.

Looking up a value like this with a key that does not exist will raise a `KeyError` exception, halting execution if uncaught. If we want to access a value without risking a `KeyError`, we can use the `dictionary.get` method. By default if the key does not exist, the method will return `None`. We can pass it a second value to return instead of `None` in the event of a failed lookup.

```
w = dictionary.get("whatever")
x = dictionary.get("whatever", "nuh-uh")
```

In this example `w` will get the value `None` and `x` will get the value `"nuh-uh"`.

Section 19.8: Creating a dictionary

Rules for creating a dictionary:

- Every key must be **unique** (otherwise it will be overridden)
- Every key must be **hashable** (can use the `hash` function to hash it; otherwise `TypeError` will be thrown)
- There is no particular order for the keys.

```
# Creating and populating it with values
stock = {'eggs': 5, 'milk': 2}

# Or creating an empty dictionary
dictionary = {}

# And populating it after
dictionary['eggs'] = 5
dictionary['milk'] = 2

# Values can also be lists
mydict = {'a': [1, 2, 3], 'b': ['one', 'two', 'three']}

# Use list.append() method to add new elements to the values list
mydict['a'].append(4)   # => {'a': [1, 2, 3, 4], 'b': ['one', 'two', 'three']}
mydict['b'].append('four')  # => {'a': [1, 2, 3, 4], 'b': ['one', 'two', 'three', 'four']}

# We can also create a dictionary using a list of two-items tuples
iterable = [('eggs', 5), ('milk', 2)]
dictionary = dict(iterables)

# Or using keyword argument:
dictionary = dict(eggs=5, milk=2)

# Another way will be to use the dict.fromkeys:
dictionary = dict.fromkeys((milk, eggs))  # => {'milk': None, 'eggs': None}
dictionary = dict.fromkeys((milk, eggs), (2, 5))  # => {'milk': 2, 'eggs': 5}
```

Section 19.9: Creating an ordered dictionary

You can create an ordered dictionary which will follow a determined order when iterating over the keys in the dictionary.

Use `OrderedDict` from the `collections` module. This will always return the dictionary elements in the original insertion order when iterated over.

```
from collections import OrderedDict

d = OrderedDict()
d['first'] = 1
d['second'] = 2
d['third'] = 3
d['last'] = 4

# Outputs "first 1", "second 2", "third 3", "last 4"
for key in d:
    print(key, d[key])
```

Section 19.10: Unpacking dictionaries using the ** operator

You can use the `**` keyword argument unpacking operator to deliver the key-value pairs in a dictionary into a function's arguments. A simplified example from the official documentation:

```
>>>
```

```
>>> def parrot(voltage, state, action):
...     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 !
```

As of Python 3.5 you can also use this syntax to merge an arbitrary number of `dict` objects.

```
>>> fish = {'name': "Nemo", 'hands': "fins", 'special': "gills"}
>>> dog = {'name': "Clifford", 'hands': "paws", 'color': "red"}
>>> fishdog = {**fish, **dog}
>>> fishdog

{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

As this example demonstrates, duplicate keys map to their lattermost value (for example "Clifford" overrides "Nemo").

Section 19.11: The trailing comma

Like lists and tuples, you can include a trailing comma in your dictionary.

```
role = {"By day": "A typical programmer",
        "By night": "Still a typical programmer", }
```

PEP 8 dictates that you should leave a space between the trailing comma and the closing brace.

Section 19.12: The dict() constructor

The `dict()` constructor can be used to create dictionaries from keyword arguments, or from a single iterable of key-value pairs, or from a single dictionary and keyword arguments.

```
dict(a=1, b=2, c=3)                   # {'a': 1, 'b': 2, 'c': 3}
dict([('d', 4), ('e', 5), ('f', 6)])  # {'d': 4, 'e': 5, 'f': 6}
dict([('a', 1)], b=2, c=3)            # {'a': 1, 'b': 2, 'c': 3}
dict({'a' : 1, 'b' : 2}, c=3)         # {'a': 1, 'b': 2, 'c': 3}
```

Section 19.13: Dictionaries Example

Dictionaries map keys to values.

```
car = {}
car["wheels"] = 4
car["color"] = "Red"
car["model"] = "Corvette"
```

Dictionary values can be accessed by their keys.

```
print "Little " + car["color"] + " " + car["model"] + "!"
# This would print out "Little Red Corvette!"
```

Dictionaries can also be created in a JSON style:

```
car = {"wheels": 4, "color": "Red", "model": "Corvette"}
```

Dictionary values can be iterated over:

```
for key in car:
  print key + ": " + car[key]

# wheels: 4
# color: Red
# model: Corvette
```

Section 19.14: All combinations of dictionary values

```
options = {
    "x": ["a", "b"],
    "y": [10, 20, 30]
}
```

Given a dictionary such as the one shown above, where there is a list representing a set of values to explore for the corresponding key. Suppose you want to explore `"x"="a"` with `"y"=10`, then `"x"="a"` with`"y"=10`, and so on until you have explored all possible combinations.

You can create a list that returns all such combinations of values using the following code.

```
import itertools

options = {
    "x": ["a", "b"],
    "y": [10, 20, 30]}

keys = options.keys()
values = (options[key] for key in keys)
combinations = [dict(zip(keys, combination)) for combination in itertools.product(*values)]
print combinations
```

This gives us the following list stored in the variable `combinations`:

```
[{'x': 'a', 'y': 10},
 {'x': 'b', 'y': 10},
 {'x': 'a', 'y': 20},
 {'x': 'b', 'y': 20},
 {'x': 'a', 'y': 30},
 {'x': 'b', 'y': 30}]
```

Chapter 20: List

The Python **List** is a general data structure widely used in Python programs. They are found in other languages, often referred to as *dynamic arrays*. They are both *mutable* and a *sequence* data type that allows them to be *indexed* and *sliced*. The list can contain different types of objects, including other list objects.

Section 20.1: List methods and supported operators

Starting with a given list `a`:

```
a = [1, 2, 3, 4, 5]
```

1. `append(value)` – appends a new element to the end of the list.

```
# Append values 6, 7, and 7 to the list
a.append(6)
a.append(7)
a.append(7)
# a: [1, 2, 3, 4, 5, 6, 7, 7]

# Append another list
b = [8, 9]
a.append(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9]]

# Append an element of a different type, as list elements do not need to have the same type
my_string = "hello world"
a.append(my_string)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9], "hello world"]
```

Note that the `append()` method only appends one new element to the end of the list. If you append a list to another list, the list that you append becomes a single element at the end of the first list.

```
# Appending a list to another list
a = [1, 2, 3, 4, 5, 6, 7, 7]
b = [8, 9]
a.append(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9]]
a[8]
# Returns: [8,9]
```

2. `extend(enumerable)` – extends the list by appending elements from another enumerable.

```
a = [1, 2, 3, 4, 5, 6, 7, 7]
b = [8, 9, 10]

# Extend list by appending all elements from b
a.extend(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]

# Extend list with elements from a non-list enumerable:
a.extend(range(3))
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10, 0, 1, 2]
```

Lists can also be concatenated with the `+` operator. Note that this does not modify any of the original lists:

```
a = [1, 2, 3, 4, 5, 6] + [7, 7] + b
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
```

3. `index(value, [startIndex])` – gets the index of the first occurrence of the input value. If the input value is not in the list a `ValueError` exception is raised. If a second argument is provided, the search is started at that specified index.

```
a.index(7)
# Returns: 6

a.index(49) # ValueError, because 49 is not in a.

a.index(7, 7)
# Returns: 7

a.index(7, 8) # ValueError, because there is no 7 starting at index 8
```

4. `insert(index, value)` – inserts `value` just before the specified `index`. Thus after the insertion the new element occupies position `index`.

```
a.insert(0, 0)  # insert 0 at position 0
a.insert(2, 5)  # insert 5 at position 2
# a: [0, 1, 5, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
```

5. `pop([index])` – removes and returns the item at `index`. With no argument it removes and returns the last element of the list.

```
a.pop(2)
# Returns: 5
# a: [0, 1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
a.pop(8)
# Returns: 7
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# With no argument:
a.pop()
# Returns: 10
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6. `remove(value)` – removes the first occurrence of the specified value. If the provided value cannot be found, a `ValueError` is raised.

```
a.remove(0)
a.remove(9)
# a: [1, 2, 3, 4, 5, 6, 7, 8]
a.remove(10)
# ValueError, because 10 is not in a
```

7. `reverse()` – reverses the list in-place and returns `None`.

```
a.reverse()
# a: [8, 7, 6, 5, 4, 3, 2, 1]
```

   There are also other ways of reversing a list.

8. `count(value)` – counts the number of occurrences of some value in the list.

```
a.count(7)
# Returns: 2
```

9. `sort()` – sorts the list in numerical and lexicographical order and returns `None`.

```
a.sort()
# a = [1, 2, 3, 4, 5, 6, 7, 8]
# Sorts the list in numerical order
```

Lists can also be reversed when sorted using the `reverse=True` flag in the `sort()` method.

```
a.sort(reverse=True)
# a = [8, 7, 6, 5, 4, 3, 2, 1]
```

If you want to sort by attributes of items, you can use the `key` keyword argument:

```
import datetime

class Person(object):
    def __init__(self, name, birthday, height):
        self.name = name
        self.birthday = birthday
        self.height = height

    def __repr__(self):
        return self.name

l = [Person("John Cena", datetime.date(1992, 9, 12), 175),
     Person("Chuck Norris", datetime.date(1990, 8, 28), 180),
     Person("Jon Skeet", datetime.date(1991, 7, 6), 185)]

l.sort(key=lambda item: item.name)
# l: [Chuck Norris, John Cena, Jon Skeet]

l.sort(key=lambda item: item.birthday)
# l: [Chuck Norris, Jon Skeet, John Cena]

l.sort(key=lambda item: item.height)
# l: [John Cena, Chuck Norris, Jon Skeet]
```

In case of list of dicts the concept is the same:

```
import datetime

l = [{'name':'John Cena', 'birthday': datetime.date(1992, 9, 12),'height': 175},
 {'name': 'Chuck Norris', 'birthday': datetime.date(1990, 8, 28),'height': 180},
 {'name': 'Jon Skeet', 'birthday': datetime.date(1991, 7, 6), 'height': 185}]

l.sort(key=lambda item: item['name'])
# l: [Chuck Norris, John Cena, Jon Skeet]

l.sort(key=lambda item: item['birthday'])
# l: [Chuck Norris, Jon Skeet, John Cena]

l.sort(key=lambda item: item['height'])
# l: [John Cena, Chuck Norris, Jon Skeet]
```

Sort by sub dict:

```
import datetime

l = [{'name':'John Cena', 'birthday': datetime.date(1992, 9, 12),'size': {'height': 175,
'weight': 100}},
 {'name': 'Chuck Norris', 'birthday': datetime.date(1990, 8, 28),'size' : {'height': 180,
'weight': 90}},
 {'name': 'Jon Skeet', 'birthday': datetime.date(1991, 7, 6), 'size': {'height': 185,
'weight': 110}}]

l.sort(key=lambda item: item['size']['height'])
# l: [John Cena, Chuck Norris, Jon Skeet]
```

Better way to sort using `attrgetter` and `itemgetter`

Lists can also be sorted using `attrgetter` and `itemgetter` functions from the operator module. These can help improve readability and reusability. Here are some examples,

```
from operator import itemgetter,attrgetter

people = [{'name':'chandan','age':20,'salary':2000},
          {'name':'chetan','age':18,'salary':5000},
          {'name':'guru','age':30,'salary':3000}]
by_age = itemgetter('age')
by_salary = itemgetter('salary')

people.sort(key=by_age) #in-place sorting by age
people.sort(key=by_salary) #in-place sorting by salary
```

`itemgetter` can also be given an index. This is helpful if you want to sort based on indices of a tuple.

```
list_of_tuples = [(1,2), (3,4), (5,0)]
list_of_tuples.sort(key=itemgetter(1))
print(list_of_tuples) #[(5, 0), (1, 2), (3, 4)]
```

Use the `attrgetter` if you want to sort by attributes of an object,

```
persons = [Person("John Cena", datetime.date(1992, 9, 12), 175),
           Person("Chuck Norris", datetime.date(1990, 8, 28), 180),
           Person("Jon Skeet", datetime.date(1991, 7, 6), 185)] #reusing Person class from above
example

person.sort(key=attrgetter('name')) #sort by name
by_birthday = attrgetter('birthday')
person.sort(key=by_birthday) #sort by birthday
```

10. `clear()` – removes all items from the list

```
a.clear()
# a = []
```

11. **Replication** – multiplying an existing list by an integer will produce a larger list consisting of that many copies of the original. This can be useful for example for list initialization:

```
b = ["blah"] * 3
# b = ["blah", "blah", "blah"]
```

```
b = [1, 3, 5] * 5
# [1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5]
```

Take care doing this if your list contains references to objects (eg a list of lists), see Common Pitfalls - List multiplication and common references.

12. **Element deletion** – it is possible to delete multiple elements in the list using the `del` keyword and slice notation:

```
a = list(range(10))
del a[::2]
# a = [1, 3, 5, 7, 9]
del a[-1]
# a = [1, 3, 5, 7]
del a[:]
# a = []
```

13. **Copying**

The default assignment "=" assigns a reference of the original list to the new name. That is, the original name and new name are both pointing to the same list object. Changes made through any of them will be reflected in another. This is often not what you intended.

```
b = a
a.append(6)
# b: [1, 2, 3, 4, 5, 6]
```

If you want to create a copy of the list you have below options.

You can slice it:

```
new_list = old_list[:]
```

You can use the built in list() function:

```
new_list = list(old_list)
```

You can use generic copy.copy():

```
import copy
new_list = copy.copy(old_list) #inserts references to the objects found in the original.
```

This is a little slower than list() because it has to find out the datatype of old_list first.

If the list contains objects and you want to copy them as well, use generic copy.deepcopy():

```
import copy
new_list = copy.deepcopy(old_list) #inserts copies of the objects found in the original.
```

Obviously the slowest and most memory-needing method, but sometimes unavoidable.

Python 3.x `Version ≥ 3.0`

`copy()` – Returns a shallow copy of the list

```
    aa = a.copy()
    # aa = [1, 2, 3, 4, 5]
```

Section 20.2: Accessing list values

Python lists are zero-indexed, and act like arrays in other languages.

```
lst = [1, 2, 3, 4]
lst[0]  # 1
lst[1]  # 2
```

Attempting to access an index outside the bounds of the list will raise an `IndexError`.

```
lst[4]  # IndexError: list index out of range
```

Negative indices are interpreted as counting from the *end* of the list.

```
lst[-1]  # 4
lst[-2]  # 3
lst[-5]  # IndexError: list index out of range
```

This is functionally equivalent to

```
lst[len(lst)-1]  # 4
```

Lists allow to use *slice notation* as `lst[start:end:step]`. The output of the slice notation is a new list containing elements from index `start` to `end-1`. If options are omitted `start` defaults to beginning of list, `end` to end of list and `step` to 1:

```
lst[1:]      # [2, 3, 4]
lst[:3]      # [1, 2, 3]
lst[::2]     # [1, 3]
lst[::-1]    # [4, 3, 2, 1]
lst[-1:0:-1] # [4, 3, 2]
lst[5:8]     # [] since starting index is greater than length of lst, returns empty list
lst[1:10]    # [2, 3, 4] same as omitting ending index
```

With this in mind, you can print a reversed version of the list by calling

```
lst[::-1]    # [4, 3, 2, 1]
```

When using step lengths of negative amounts, the starting index has to be greater than the ending index otherwise the result will be an empty list.

```
lst[3:1:-1] # [4, 3]
```

Using negative step indices are equivalent to the following code:

```
reversed(lst)[0:2] # 0 = 1 -1
                   # 2 = 3 -1
```

The indices used are 1 less than those used in negative indexing and are reversed.

Advanced slicing

When lists are sliced the `__getitem__()` method of the list object is called, with a `slice` object. Python has a builtin slice method to generate slice objects. We can use this to *store* a slice and reuse it later like so,

```
data = 'chandan purohit    22 2000'  #assuming data fields of fixed length
name_slice = slice(0,19)
age_slice = slice(19,21)
salary_slice = slice(22,None)

#now we can have more readable slices
print(data[name_slice]) #chandan purohit
print(data[age_slice]) #'22'
print(data[salary_slice]) #'2000'
```

This can be of great use by providing slicing functionality to our objects by overriding `__getitem__` in our class.

Section 20.3: Checking if list is empty

The emptiness of a list is associated to the boolean `False`, so you don't have to check `len(lst) == 0`, but just `lst` or `not lst`

```
lst = []
if not lst:
    print("list is empty")

# Output: list is empty
```

Section 20.4: Iterating over a list

Python supports using a `for` loop directly on a list:

```
my_list = ['foo', 'bar', 'baz']
for item in my_list:
    print(item)

# Output: foo
# Output: bar
# Output: baz
```

You can also get the position of each item at the same time:

```
for (index, item) in enumerate(my_list):
    print('The item in position {} is: {}'.format(index, item))

# Output: The item in position 0 is: foo
# Output: The item in position 1 is: bar
# Output: The item in position 2 is: baz
```

The other way of iterating a list based on the index value:

```
for i in range(0,len(my_list)):
    print(my_list[i])
#output:
>>>
foo
bar
```

```
baz
```

Note that changing items in a list while iterating on it may have unexpected results:

```
for item in my_list:
    if item == 'foo':
        del my_list[0]
    print(item)

# Output: foo
# Output: baz
```

In this last example, we deleted the first item at the first iteration, but that caused `bar` to be skipped.

Section 20.5: Checking whether an item is in a list

Python makes it very simple to check whether an item is in a list. Simply use the `in` operator.

```
lst = ['test', 'twest', 'tweast', 'treast']

'test' in lst
# Out: True

'toast' in lst
# Out: False
```

> Note: the `in` operator on sets is asymptotically faster than on lists. If you need to use it many times on potentially large lists, you may want to convert your `list` to a `set`, and test the presence of elements on the `set`.

```
slst = set(lst)
'test' in slst
# Out: True
```

Section 20.6: Any and All

You can use `all()` to determine if all the values in an iterable evaluate to True

```
nums = [1, 1, 0, 1]
all(nums)
# False
chars = ['a', 'b', 'c', 'd']
all(chars)
# True
```

Likewise, `any()` determines if one or more values in an iterable evaluate to True

```
nums = [1, 1, 0, 1]
any(nums)
# True
vals = [None, None, None, False]
any(vals)
# False
```

While this example uses a list, it is important to note these built-ins work with any iterable, including generators.

```
vals = [1, 2, 3, 4]
any(val > 12 for val in vals)
# False
any((val * 2) > 6 for val in vals)
# True
```

Section 20.7: Reversing list elements

You can use the `reversed` function which returns an iterator to the reversed list:

```
In [3]: rev = reversed(numbers)

In [4]: rev
Out[4]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Note that the list "numbers" remains unchanged by this operation, and remains in the same order it was originally.

To reverse in place, you can also use the `reverse` method.

You can also reverse a list (actually obtaining a copy, the original list is unaffected) by using the slicing syntax, setting the third argument (the step) as -1:

```
In [1]: numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

In [2]: numbers[::-1]
Out[2]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Section 20.8: Concatenate and Merge lists

1. **The simplest way to concatenate** `list1` and `list2`:

   ```
   merged = list1 + list2
   ```

2. `zip` **returns a list of tuples**, where the i-th tuple contains the i-th element from each of the argument sequences or iterables:

   ```
   alist = ['a1', 'a2', 'a3']
   blist = ['b1', 'b2', 'b3']

   for a, b in zip(alist, blist):
       print(a, b)

   # Output:
   # a1 b1
   # a2 b2
   # a3 b3
   ```

   If the lists have different lengths then the result will include only as many elements as the shortest one:

   ```
   alist = ['a1', 'a2', 'a3']
   blist = ['b1', 'b2', 'b3', 'b4']
   for a, b in zip(alist, blist):
       print(a, b)

   # Output:
   # a1 b1
   ```

```
# a2 b2
# a3 b3

alist = []
len(list(zip(alist, blist)))

# Output:
# 0
```

For padding lists of unequal length to the longest one with `None`s use `itertools.zip_longest` (`itertools.izip_longest` in Python 2)

```
alist = ['a1', 'a2', 'a3']
blist = ['b1']
clist = ['c1', 'c2', 'c3', 'c4']

for a,b,c in itertools.zip_longest(alist, blist, clist):
    print(a, b, c)

# Output:
# a1 b1 c1
# a2 None c2
# a3 None c3
# None None c4
```

3. **Insert to a specific index values:**

```
alist = [123, 'xyz', 'zara', 'abc']
alist.insert(3, [2009])
print("Final List :", alist)
```

Output:

```
Final List : [123, 'xyz', 'zara', 2009, 'abc']
```

Section 20.9: Length of a list

Use `len()` to get the one-dimensional length of a list.

```
len(['one', 'two'])  # returns 2

len(['one', [2, 3], 'four'])  # returns 3, not 4
```

`len()` also works on strings, dictionaries, and other data structures similar to lists.

Note that `len()` is a built-in function, not a method of a list object.

Also note that the cost of `len()` is `0(1)`, meaning it will take the same amount of time to get the length of a list regardless of its length.

Section 20.10: Remove duplicate values in list

Removing duplicate values in a list can be done by converting the list to a `set` (that is an unordered collection of distinct objects). If a `list` data structure is needed, then the set can be converted back to a list using the function `list()`:

```
names = ["aixk", "duke", "edik", "tofp", "duke"]
list(set(names))
# Out: ['duke', 'tofp', 'aixk', 'edik']
```

Note that by converting a list to a set the original ordering is lost.

To preserve the order of the list one can use an `OrderedDict`

```
import collections
>>> collections.OrderedDict.fromkeys(names).keys()
# Out: ['aixk', 'duke', 'edik', 'tofp']
```

Section 20.11: Comparison of lists

It's possible to compare lists and other sequences lexicographically using comparison operators. Both operands must be of the same type.

```
[1, 10, 100] < [2, 10, 100]
# True, because 1 < 2
[1, 10, 100] < [1, 10, 100]
# False, because the lists are equal
[1, 10, 100] <= [1, 10, 100]
# True, because the lists are equal
[1, 10, 100] < [1, 10, 101]
# True, because 100 < 101
[1, 10, 100] < [0, 10, 100]
# False, because 0 < 1
```

If one of the lists is contained at the start of the other, the shortest list wins.

```
[1, 10] < [1, 10, 100]
# True
```

Section 20.12: Accessing values in nested list

Starting with a three-dimensional list:

```
alist = [[[1,2],[3,4]], [[5,6,7],[8,9,10], [12, 13, 14]]]
```

Accessing items in the list:

```
print(alist[0][0][1])
#2
#Accesses second element in the first list in the first list

print(alist[1][1][2])
#10
#Accesses the third element in the second list in the second list
```

Performing support operations:

```
alist[0][0].append(11)
print(alist[0][0][2])
#11
#Appends 11 to the end of the first list in the first list
```

Using nested for loops to print the list:

```
for row in alist: #One way to loop through nested lists
    for col in row:
        print(col)
#[1, 2, 11]
#[3, 4]
#[5, 6, 7]
#[8, 9, 10]
#[12, 13, 14]
```

Note that this operation can be used in a list comprehension or even as a generator to produce efficiencies, e.g.:

```
[col for row in alist for col in row]
#[[1, 2, 11], [3, 4], [5, 6, 7], [8, 9, 10], [12, 13, 14]]
```

Not all items in the outer lists have to be lists themselves:

```
alist[1].insert(2, 15)
#Inserts 15 into the third position in the second list
```

Another way to use nested for loops. The other way is better but I've needed to use this on occasion:

```
for row in range(len(alist)): #A less Pythonic way to loop through lists
    for col in range(len(alist[row])):
       print(alist[row][col])

#[1, 2, 11]
#[3, 4]
#[5, 6, 7]
#[8, 9, 10]
#15
#[12, 13, 14]
```

Using slices in nested list:

```
print(alist[1][1:])
#[[8, 9, 10], 15, [12, 13, 14]]
#Slices still work
```

The final list:

```
print(alist)
#[[[1, 2, 11], [3, 4]], [[5, 6, 7], [8, 9, 10], 15, [12, 13, 14]]]
```

Section 20.13: Initializing a List to a Fixed Number of Elements

For **immutable** elements (e.g. `None`, string literals etc.):

```
my_list = [None] * 10
my_list = ['test'] * 10
```

For **mutable** elements, the same construct will result in all elements of the list referring to the same object, for example, for a set:

```
>>> my_list=[{1}] * 10
```

```
>>> print(my_list)
[{1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}]
>>> my_list[0].add(2)
>>> print(my_list)
[{1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}]
```

Instead, to initialize the list with a fixed number of **different mutable** objects, use:

```
my_list=[{1} for _ in range(10)]
```

Chapter 21: List comprehensions

List comprehensions in Python are concise, syntactic constructs. They can be utilized to generate lists from other lists by applying functions to each element in the list. The following section explains and demonstrates the use of these expressions.

Section 21.1: List Comprehensions

A list comprehension creates a new `list` by applying an expression to each element of an iterable. The most basic form is:

```
[ <expression> for <element> in <iterable> ]
```

There's also an optional 'if' condition:

```
[ <expression> for <element> in <iterable> if <condition> ]
```

Each **`<element>`** in the **`<iterable>`** is plugged in to the **`<expression>`** if the (optional) **`<condition>`** evaluates to true. All results are returned at once in the new list. Generator expressions are evaluated lazily, but list comprehensions evaluate the entire iterator immediately - consuming memory proportional to the iterator's length.

To create a `list` of squared integers:

```
squares = [x * x for x in (1, 2, 3, 4)]
# squares: [1, 4, 9, 16]
```

The `for` expression sets `x` to each value in turn from `(1, 2, 3, 4)`. The result of the expression `x * x` is appended to an internal `list`. The internal `list` is assigned to the variable `squares` when completed.

Besides a speed increase (as explained here), a list comprehension is roughly equivalent to the following for-loop:

```
squares = []
for x in (1, 2, 3, 4):
    squares.append(x * x)
# squares: [1, 4, 9, 16]
```

The expression applied to each element can be as complex as needed:

```
# Get a list of uppercase characters from a string
[s.upper() for s in "Hello World"]
# ['H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D']

# Strip off any commas from the end of strings in a list
[w.strip(',') for w in ['these,', 'words,,', 'mostly', 'have,commas,']]
# ['these', 'words', 'mostly', 'have,commas']

# Organize letters in words more reasonably - in an alphabetical order
sentence = "Beautiful is better than ugly"
["".join(sorted(word, key = lambda x: x.lower())) for word in sentence.split()]
# ['aBefiltuu', 'is', 'beertt', 'ahnt', 'gluy']
```

else

`else` can be used in List comprehension constructs, but be careful regarding the syntax. The if/else clauses should

be used before `for` loop, not after:

```
# create a list of characters in apple, replacing non vowels with '*'
# Ex - 'apple' --> ['a', '*', '*', '*' ,'e']

[x for x in 'apple' if x in 'aeiou' else '*']
#SyntaxError: invalid syntax

# When using if/else together use them before the loop
[x if x in 'aeiou' else '*' for x in 'apple']
#['a', '*', '*', '*', 'e']
```

Note this uses a different language construct, a conditional expression, which itself is not part of the comprehension syntax. Whereas the `if` after the `for`...`in` *is* a part of list comprehensions and used to *filter* elements from the source iterable.

Double Iteration

Order of double iteration `[... for x in ... for y in ...]` is either natural or counter-intuitive. The rule of thumb is to follow an equivalent `for` loop:

```
def foo(i):
    return i, i + 0.5

for i in range(3):
    for x in foo(i):
        yield str(x)
```

This becomes:

```
[str(x)
    for i in range(3)
        for x in foo(i)
]
```

This can be compressed into one line as `[str(x) for i in range(3) for x in foo(i)]`

In-place Mutation and Other Side Effects

Before using list comprehension, understand the difference between functions called for their side effects (*mutating*, or *in-place* functions) which usually return `None`, and functions that return an interesting value.

Many functions (especially *pure* functions) simply take an object and return some object. An *in-place* function modifies the existing object, which is called a *side effect*. Other examples include input and output operations such as printing.

`list.sort()` sorts a list *in-place* (meaning that it modifies the original list) and returns the value `None`. Therefore, it won't work as expected in a list comprehension:

```
[x.sort() for x in [[2, 1], [4, 3], [0, 1]]]
# [None, None, None]
```

Instead, `sorted()` returns a sorted `list` rather than sorting in-place:

```
[sorted(x) for x in [[2, 1], [4, 3], [0, 1]]]
# [[1, 2], [3, 4], [0, 1]]
```

Using comprehensions for side-effects is possible, such as I/O or in-place functions. Yet a for loop is usually more readable. While this works in Python 3:

```
[print(x) for x in (1, 2, 3)]
```

Instead use:

```
for x in (1, 2, 3):
    print(x)
```

In some situations, side effect functions *are* suitable for list comprehension. `random.randrange()` has the side effect of changing the state of the random number generator, but it also returns an interesting value. Additionally, `next()` can be called on an iterator.

The following random value generator is not pure, yet makes sense as the random generator is reset every time the expression is evaluated:

```
from random import randrange
[randrange(1, 7) for _ in range(10)]
# [2, 3, 2, 1, 1, 5, 2, 4, 3, 5]
```

Whitespace in list comprehensions

More complicated list comprehensions can reach an undesired length, or become less readable. Although less common in examples, it is possible to break a list comprehension into multiple lines like so:

```
[
    x for x
    in 'foo'
    if x not in 'bar'
]
```

Section 21.2: Conditional List Comprehensions

Given a list comprehension you can append one or more `if` conditions to filter values.

```
[<expression> for <element> in <iterable> if <condition>]
```

For each **`<element>`** in **`<iterable>`**; if **`<condition>`** evaluates to `True`, add **`<expression>`** (usually a function of **`<element>`**) to the returned list.

For example, this can be used to extract only even numbers from a sequence of integers:

```
[x for x in range(10) if x % 2 == 0]
# Out: [0, 2, 4, 6, 8]
```

Live demo

The above code is equivalent to:

```
even_numbers = []
```

```
for x in range(10):
    if x % 2 == 0:
        even_numbers.append(x)

print(even_numbers)
# Out: [0, 2, 4, 6, 8]
```

Also, a conditional list comprehension of the form `[e for x in y if c]` (where `e` and `c` are expressions in terms of `x`) is equivalent to `list(filter(lambda x: c, map(lambda x: e, y)))`.

Despite providing the same result, pay attention to the fact that the former example is almost 2x faster than the latter one. For those who are curious, this is a nice explanation of the reason why.

Note that this is quite different from the `... if ... else ...` conditional expression (sometimes known as a ternary expression) that you can use for the **`<expression>`** part of the list comprehension. Consider the following example:

```
[x if x % 2 == 0 else None for x in range(10)]
# Out: [0, None, 2, None, 4, None, 6, None, 8, None]
```

Live demo

Here the conditional expression isn't a filter, but rather an operator determining the value to be used for the list items:

```
<value-if-condition-is-true> if <condition> else <value-if-condition-is-false>
```

This becomes more obvious if you combine it with other operators:

```
[2 * (x if x % 2 == 0 else -1) + 1 for x in range(10)]
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

Live demo

If you are using Python 2.7, `xrange` may be better than `range` for several reasons as described in the `xrange` documentation.

```
[2 * (x if x % 2 == 0 else -1) + 1 for x in xrange(10)]
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

The above code is equivalent to:

```
numbers = []
for x in range(10):
    if x % 2 == 0:
        temp = x
    else:
        temp = -1
    numbers.append(2 * temp + 1)
print(numbers)
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

One can combine ternary expressions and `if` conditions. The ternary operator works on the filtered result:

```
[x if x > 2 else '*' for x in range(10) if x % 2 == 0]
# Out: ['*', '*', 4, 6, 8]
```

The same couldn't have been achieved just by ternary operator only:

```
[x if (x > 2 and x % 2 == 0) else '*' for x in range(10)]
# Out:['*', '*', '*', '*', 4, '*', 6, '*', 8, '*']
```

See also: Filters, which often provide a sufficient alternative to conditional list comprehensions.

Section 21.3: Avoid repetitive and expensive operations using conditional clause

Consider the below list comprehension:

```
>>> def f(x):
...     import time
...     time.sleep(.1)       # Simulate expensive function
...     return x**2

>>> [f(x) for x in range(1000) if f(x) > 10]
[16, 25, 36, ...]
```

This results in two calls to `f(x)` for 1,000 values of `x`: one call for generating the value and the other for checking the `if` condition. If `f(x)` is a particularly expensive operation, this can have significant performance implications. Worse, if calling `f()` has side effects, it can have surprising results.

Instead, you should evaluate the expensive operation only once for each value of `x` by generating an intermediate iterable (generator expression) as follows:

```
>>> [v for v in (f(x) for x in range(1000)) if v > 10]
[16, 25, 36, ...]
```

Or, using the builtin map equivalent:

```
>>> [v for v in map(f, range(1000)) if v > 10]
[16, 25, 36, ...]
```

Another way that could result in a more readable code is to put the partial result (`v` in the previous example) in an iterable (such as a list or a tuple) and then iterate over it. Since `v` will be the only element in the iterable, the result is that we now have a reference to the output of our slow function computed only once:

```
>>> [v for x in range(1000) for v in [f(x)] if v > 10]
[16, 25, 36, ...]
```

However, in practice, the logic of code can be more complicated and it's important to keep it readable. In general, a separate generator function is recommended over a complex one-liner:

```
>>> def process_prime_numbers(iterable):
...     for x in iterable:
...         if is_prime(x):
...             yield f(x)
...
>>> [x for x in process_prime_numbers(range(1000)) if x > 10]
[11, 13, 17, 19, ...]
```

Another way to prevent computing `f(x)` multiple times is to use the `@functools.lru_cache()`(Python 3.2+) decorator on `f(x)`. This way since the output of `f` for the input `x` has already been computed once, the second

function invocation of the original list comprehension will be as fast as a dictionary lookup. This approach uses memoization to improve efficiency, which is comparable to using generator expressions.

Say you have to flatten a list

```
l = [[1, 2, 3], [4, 5, 6], [7], [8, 9]]
```

Some of the methods could be:

```
reduce(lambda x, y: x+y, l)

sum(l, [])

list(itertools.chain(*l))
```

However list comprehension would provide the best time complexity.

```
[item for sublist in l for item in sublist]
```

The shortcuts based on + (including the implied use in sum) are, of necessity, O(L^2) when there are L sublists -- as the intermediate result list keeps getting longer, at each step a new intermediate result list object gets allocated, and all the items in the previous intermediate result must be copied over (as well as a few new ones added at the end). So (for simplicity and without actual loss of generality) say you have L sublists of I items each: the first I items are copied back and forth L-1 times, the second I items L-2 times, and so on; total number of copies is I times the sum of x for x from 1 to L excluded, i.e., I * (L**2)/2.

The list comprehension just generates one list, once, and copies each item over (from its original place of residence to the result list) also exactly once.

Section 21.4: Dictionary Comprehensions

A dictionary comprehension is similar to a list comprehension except that it produces a dictionary object instead of a list.

A basic example:

Python 2.x Version ≥ 2.7

```
{x: x * x for x in (1, 2, 3, 4)}
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

which is just another way of writing:

```
dict((x, x * x) for x in (1, 2, 3, 4))
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

As with a list comprehension, we can use a conditional statement inside the dict comprehension to produce only the dict elements meeting some criterion.

Python 2.x Version ≥ 2.7

```
{name: len(name) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6}
# Out: {'Exchange': 8, 'Overflow': 8}
```

Or, rewritten using a generator expression.

```
dict((name, len(name)) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6)
# Out: {'Exchange': 8, 'Overflow': 8}
```

Starting with a dictionary and using dictionary comprehension as a key-value pair filter

Python 2.x Version ≥ 2.7

```
initial_dict = {'x': 1, 'y': 2}
{key: value for key, value in initial_dict.items() if key == 'x'}
# Out: {'x': 1}
```

Switching key and value of dictionary (invert dictionary)

If you have a dict containing simple *hashable* values (duplicate values may have unexpected results):

```
my_dict = {1: 'a', 2: 'b', 3: 'c'}
```

and you wanted to swap the keys and values you can take several approaches depending on your coding style:

- `swapped = {v: k for k, v in my_dict.items()}`
- `swapped = dict((v, k) for k, v in my_dict.iteritems())`
- `swapped = dict(zip(my_dict.values(), my_dict))`
- `swapped = dict(zip(my_dict.values(), my_dict.keys()))`
- `swapped = dict(map(reversed, my_dict.items()))`

```
print(swapped)
# Out: {a: 1, b: 2, c: 3}
```

Python 2.x Version ≥ 2.3

If your dictionary is large, consider *importing* *itertools* and utilize `izip` or `imap`.

Merging Dictionaries

Combine dictionaries and optionally override old values with a nested dictionary comprehension.

```
dict1 = {'w': 1, 'x': 1}
dict2 = {'x': 2, 'y': 2, 'z': 2}

{k: v for d in [dict1, dict2] for k, v in d.items()}
# Out: {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

However, dictionary unpacking (PEP 448) may be a preferred.

Python 3.x Version ≥ 3.5

```
{**dict1, **dict2}
# Out: {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

Note: dictionary comprehensions were added in Python 3.0 and backported to 2.7+, unlike list comprehensions, which were added in 2.0. Versions < 2.7 can use generator expressions and the `dict()` builtin to simulate the behavior of dictionary comprehensions.

Section 21.5: List Comprehensions with Nested Loops

List Comprehensions can use nested `for` loops. You can code any number of nested for loops within a list comprehension, and each `for` loop may have an optional associated `if` test. When doing so, the order of the `for`

constructs is the same order as when writing a series of nested `for` statements. The general structure of list comprehensions looks like this:

```
[ expression for target1 in iterable1 [if condition1]
              for target2 in iterable2 [if condition2]...
              for targetN in iterableN [if conditionN] ]
```

For example, the following code flattening a list of lists using multiple `for` statements:

```
data = [[1, 2], [3, 4], [5, 6]]
output = []
for each_list in data:
    for element in each_list:
        output.append(element)
print(output)
# Out: [1, 2, 3, 4, 5, 6]
```

can be equivalently written as a list comprehension with multiple `for` constructs:

```
data = [[1, 2], [3, 4], [5, 6]]
output = [element for each_list in data for element in each_list]
print(output)
# Out: [1, 2, 3, 4, 5, 6]
```

Live Demo

In both the expanded form and the list comprehension, the outer loop (first for statement) comes first.

In addition to being more compact, the nested comprehension is also significantly faster.

```
In [1]: data = [[1,2],[3,4],[5,6]]
In [2]: def f():
   ...:     output=[]
   ...:     for each_list in data:
   ...:         for element in each_list:
   ...:             output.append(element)
   ...:     return output
In [3]: timeit f()
1000000 loops, best of 3: 1.37 µs per loop
In [4]: timeit [inner for outer in data for inner in outer]
1000000 loops, best of 3: 632 ns per loop
```

The overhead for the function call above is about *140ns*.

Inline `if`s are nested similarly, and may occur in any position after the first `for`:

```
data = [[1], [2, 3], [4, 5]]
output = [element for each_list in data
                  if len(each_list) == 2
                  for element in each_list
                  if element != 5]
print(output)
# Out: [2, 3, 4]
```

Live Demo

For the sake of readability, however, you should consider using traditional *for-loops*. This is especially true when nesting is more than 2 levels deep, and/or the logic of the comprehension is too complex. multiple nested loop list

comprehension could be error prone or it gives unexpected result.

Section 21.6: Generator Expressions

Generator expressions are very similar to list comprehensions. The main difference is that it does not create a full set of results at once; it creates a generator object which can then be iterated over.

For instance, see the difference in the following code:

```
# list comprehension
[x**2 for x in range(10)]
# Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Python 2.x `Version ≥ 2.4`

```
# generator comprehension
(x**2 for x in xrange(10))
# Output: <generator object <genexpr> at 0x11b4b7c80>
```

These are two very different objects:

- the list comprehension returns a `list` object whereas the generator comprehension returns a `generator`.
- `generator` objects cannot be indexed and makes use of the `next` function to get items in order.

Note: We use `xrange` since it too creates a generator object. If we would use range, a list would be created. Also, `xrange` exists only in later version of python 2. In python 3, `range` just returns a generator. For more information, see the *Differences between range and xrange functions* example.

Python 2.x `Version ≥ 2.4`

```
g = (x**2 for x in xrange(10))
print(g[0])
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object has no attribute '__getitem__'
```

```
g.next()  # 0
g.next()  # 1
g.next()  # 4
...
g.next()  # 81

g.next()  # Throws StopIteration Exception
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Python 3.x `Version ≥ 3.0`

> NOTE: The function `g.next()` should be substituted by `next(g)` and `xrange` with `range` since `Iterator.next()` and `xrange()` do not exist in Python 3.

Although both of these can be iterated in a similar way:

```
for i in [x**2 for x in range(10)]:
    print(i)

"""
Out:
0
1
4
...
81
"""
```

Python 2.x `Version ≥ 2.4`

```
for i in (x**2 for x in xrange(10)):
    print(i)

"""
Out:
0
1
4
.
.
.
81
"""
```

Use cases

Generator expressions are lazily evaluated, which means that they generate and return each value only when the generator is iterated. This is often useful when iterating through large datasets, avoiding the need to create a duplicate of the dataset in memory:

```
for square in (x**2 for x in range(1000000)):
    #do something
```

Another common use case is to avoid iterating over an entire iterable if doing so is not necessary. In this example, an item is retrieved from a remote API with each iteration of `get_objects()`. Thousands of objects may exist, must be retrieved one-by-one, and we only need to know if an object matching a pattern exists. By using a generator expression, when we encounter an object matching the pattern.

```
def get_objects():
    """Gets objects from an API one by one"""
    while True:
        yield get_next_item()

def object_matches_pattern(obj):
    # perform potentially complex calculation
    return matches_pattern

def right_item_exists():
    items = (object_matched_pattern(each) for each in get_objects())
    for item in items:
        if item.is_the_right_one:


            return True
    return False
```

Section 21.7: Set Comprehensions

Set comprehension is similar to list and dictionary comprehension, but it produces a set, which is an unordered collection of unique elements.

Python 2.x Version ≥ 2.7

```
# A set containing every value in range(5):
{x for x in range(5)}
# Out: {0, 1, 2, 3, 4}

# A set of even numbers between 1 and 10:
{x for x in range(1, 11) if x % 2 == 0}
# Out: {2, 4, 6, 8, 10}

# Unique alphabetic characters in a string of text:
text = "When in the Course of human events it becomes necessary for one people..."
{ch.lower() for ch in text if ch.isalpha()}
# Out: set(['a', 'c', 'b', 'e', 'f', 'i', 'h', 'm', 'l', 'o',
#           'n', 'p', 's', 'r', 'u', 't', 'w', 'v', 'y'])
```

Live Demo

Keep in mind that sets are unordered. This means that the order of the results in the set may differ from the one presented in the above examples.

Note: Set comprehension is available since python 2.7+, unlike list comprehensions, which were added in 2.0. In Python 2.2 to Python 2.6, the `set()` function can be used with a generator expression to produce the same result:

Python 2.x Version ≥ 2.2

```
set(x for x in range(5))
# Out: {0, 1, 2, 3, 4}
```

Section 21.8: Refactoring filter and map to list comprehensions

The `filter` or `map` functions should often be replaced by list comprehensions. Guido Van Rossum describes this well in an open letter in 2005:

> `filter(P, S)` is almost always written clearer as `[x for x in S if P(x)]`, and this has the huge advantage that the most common usages involve predicates that are comparisons, e.g. `x==42`, and defining a lambda for that just requires much more effort for the reader (plus the lambda is slower than the list comprehension). Even more so for `map(F, S)` which becomes `[F(x) for x in S]`. Of course, in many cases you'd be able to use generator expressions instead.

The following lines of code are considered "*not pythonic*" and will raise errors in many python linters.

```
filter(lambda x: x % 2 == 0, range(10)) # even numbers < 10
map(lambda x: 2*x, range(10)) # multiply each number by two
reduce(lambda x,y: x+y, range(10)) # sum of all elements in list
```

Taking what we have learned from the previous quote, we can break down these `filter` and `map` expressions into their equivalent *list comprehensions*; also removing the *lambda* functions from each - making the code more readable in the process.

```
# Filter:
# P(x) = x % 2 == 0
# S = range(10)
[x for x in range(10) if x % 2 == 0]

# Map
# F(x) = 2*x
# S = range(10)
[2*x for x in range(10)]
```

Readability becomes even more apparent when dealing with chaining functions. Where due to readability, the results of one map or filter function should be passed as a result to the next; with simple cases, these can be replaced with a single list comprehension. Further, we can easily tell from the list comprehension what the outcome of our process is, where there is more cognitive load when reasoning about the chained Map & Filter process.

```
# Map & Filter
filtered = filter(lambda x: x % 2 == 0, range(10))
results = map(lambda x: 2*x, filtered)

# List comprehension
results = [2*x for x in range(10) if x % 2 == 0]
```

Refactoring - Quick Reference

- **Map**

```
map(F, S) == [F(x) for x in S]
```

- **Filter**

```
filter(P, S) == [x for x in S if P(x)]
```

where `F` and `P` are functions which respectively transform input values and return a `bool`

Section 21.9: Comprehensions involving tuples

The `for` clause of a list comprehension can specify more than one variable:

```
[x + y for x, y in [(1, 2), (3, 4), (5, 6)]]
# Out: [3, 7, 11]

[x + y for x, y in zip([1, 3, 5], [2, 4, 6])]
# Out: [3, 7, 11]
```

This is just like regular `for` loops:

```
for x, y in [(1,2), (3,4), (5,6)]:
    print(x+y)
# 3
# 7
# 11
```

Note however, if the expression that begins the comprehension is a tuple then it must be parenthesized:

```
[x, y for x, y in [(1, 2), (3, 4), (5, 6)]]
```

```
# SyntaxError: invalid syntax

[(x, y) for x, y in [(1, 2), (3, 4), (5, 6)]]
# Out: [(1, 2), (3, 4), (5, 6)]
```

Section 21.10: Counting Occurrences Using Comprehension

When we want to count the number of items in an iterable, that meet some condition, we can use comprehension to produce an idiomatic syntax:

```
# Count the numbers in `range(1000)` that are even and contain the digit `9`:
print (sum(
    1 for x in range(1000)
    if x % 2 == 0 and
    '9' in str(x)
))
# Out: 95
```

The basic concept can be summarized as:

1. Iterate over the elements in `range(1000)`.
2. Concatenate all the needed `if` conditions.
3. Use 1 as *expression* to return a 1 for each item that meets the conditions.
4. Sum up all the `1`s to determine number of items that meet the conditions.

Note: Here we are not collecting the `1`s in a list (note the absence of square brackets), but we are passing the ones directly to the `sum` function that is summing them up. This is called a *generator expression*, which is similar to a Comprehension.

Section 21.11: Changing Types in a List

Quantitative data is often read in as strings that must be converted to numeric types before processing. The types of all list items can be converted with either a List Comprehension or the `map()` function.

```
# Convert a list of strings to integers.
items = ["1","2","3","4"]
[int(item) for item in items]
# Out: [1, 2, 3, 4]

# Convert a list of strings to float.
items = ["1","2","3","4"]
map(float, items)
# Out:[1.0, 2.0, 3.0, 4.0]
```

Section 21.12: Nested List Comprehensions

Nested list comprehensions, unlike list comprehensions with nested loops, are List comprehensions within a list comprehension. The initial expression can be any arbitrary expression, including another list comprehension.

```
#List Comprehension with nested loop
[x + y for x in [1, 2, 3] for y in [3, 4, 5]]
#Out: [4, 5, 6, 5, 6, 7, 6, 7, 8]

#Nested List Comprehension
[[x + y for x in [1, 2, 3]] for y in [3, 4, 5]]
#Out: [[4, 5, 6], [5, 6, 7], [6, 7, 8]]
```

The Nested example is equivalent to

```
l = []
for y in [3, 4, 5]:
    temp = []
    for x in [1, 2, 3]:
        temp.append(x + y)
    l.append(temp)
```

One example where a nested comprehension can be used it to transpose a matrix.

```
matrix = [[1,2,3],
          [4,5,6],
          [7,8,9]]

[[row[i] for row in matrix] for i in range(len(matrix))]
# [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Like nested `for` loops, there is no limit to how deep comprehensions can be nested.

```
[[[i + j + k for k in 'cd'] for j in 'ab'] for i in '12']
# Out: [[['1ac', '1ad'], ['1bc', '1bd']], [['2ac', '2ad'], ['2bc', '2bd']]]
```

Section 21.13: Iterate two or more list simultaneously within list comprehension

For iterating more than two lists simultaneously within *list comprehension*, one may use `zip()` as:

```
>>> list_1 = [1, 2, 3 , 4]
>>> list_2 = ['a', 'b', 'c', 'd']
>>> list_3 = ['6', '7', '8', '9']

# Two lists
>>> [(i, j) for i, j in zip(list_1, list_2)]
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]

# Three lists
>>> [(i, j, k) for i, j, k in zip(list_1, list_2, list_3)]
[(1, 'a', '6'), (2, 'b', '7'), (3, 'c', '8'), (4, 'd', '9')]

# so on ...
```

Chapter 22: List slicing (selecting parts of lists)

Section 22.1: Using the third "step" argument

```
lst = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

lst[::2]
# Output: ['a', 'c', 'e', 'g']

lst[::3]
# Output: ['a', 'd', 'g']
```

Section 22.2: Selecting a sublist from a list

```
lst = ['a', 'b', 'c', 'd', 'e']

lst[2:4]
# Output: ['c', 'd']

lst[2:]
# Output: ['c', 'd', 'e']

lst[:4]
# Output: ['a', 'b', 'c', 'd']
```

Section 22.3: Reversing a list with slicing

```
a = [1, 2, 3, 4, 5]

# steps through the list backwards (step=-1)
b = a[::-1]

# built-in list method to reverse 'a'
a.reverse()

if a = b:
    print(True)

print(b)

# Output:
# True
# [5, 4, 3, 2, 1]
```

Section 22.4: Shifting a list using slicing

```
def shift_list(array, s):
    """Shifts the elements of a list to the left or right.

    Args:
        array - the list to shift
        s - the amount to shift the list ('+': right-shift, '-': left-shift)

    Returns:
        shifted_array - the shifted list
```

```
    """
    # calculate actual shift amount (e.g., 11 --> 1 if length of the array is 5)
    s %= len(array)

    # reverse the shift direction to be more intuitive
    s *= -1

    # shift array with list slicing
    shifted_array = array[s:] + array[:s]

    return shifted_array

my_array = [1, 2, 3, 4, 5]

# negative numbers
shift_list(my_array, -7)
>>> [3, 4, 5, 1, 2]

# no shift on numbers equal to the size of the array
shift_list(my_array, 5)
>>> [1, 2, 3, 4, 5]

# works on positive numbers
shift_list(my_array, 3)
>>> [3, 4, 5, 1, 2]
```

Chapter 23: groupby()

Parameter	Details
iterable	Any python iterable
key	Function(criteria) on which to group the iterable

In Python, the `itertools.groupby()` method allows developers to group values of an iterable class based on a specified property into another iterable set of values.

Section 23.1: Example 4

In this example we see what happens when we use different types of iterable.

```
things = [("animal", "bear"), ("animal", "duck"), ("plant", "cactus"), ("vehicle", "harley"), \
          ("vehicle", "speed boat"), ("vehicle", "school bus")]
dic = {}
f = lambda x: x[0]
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)
dic
```

Results in

```
{'animal': [('animal', 'bear'), ('animal', 'duck')],
 'plant': [('plant', 'cactus')],
 'vehicle': [('vehicle', 'harley'),
  ('vehicle', 'speed boat'),
  ('vehicle', 'school bus')]}
```

This example below is essentially the same as the one above it. The only difference is that I have changed all the tuples to lists.

```
things = [["animal", "bear"], ["animal", "duck"], ["vehicle", "harley"], ["plant", "cactus"], \
          ["vehicle", "speed boat"], ["vehicle", "school bus"]]
dic = {}
f = lambda x: x[0]
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)
dic
```

Results

```
{'animal': [['animal', 'bear'], ['animal', 'duck']],
 'plant': [['plant', 'cactus']],
 'vehicle': [['vehicle', 'harley'],
  ['vehicle', 'speed boat'],
  ['vehicle', 'school bus']]}
```

Section 23.2: Example 2

This example illustrates how the default key is chosen if we do not specify any

```
c = groupby(['goat', 'dog', 'cow', 1, 1, 2, 3, 11, 10, ('persons', 'man', 'woman')])
dic = {}
for k, v in c:
```

```
    dic[k] = list(v)
dic
```

Results in

```
{1: [1, 1],
 2: [2],
 3: [3],
 ('persons', 'man', 'woman'): [('persons', 'man', 'woman')],
 'cow': ['cow'],
 'dog': ['dog'],
 10: [10],
 11: [11],
 'goat': ['goat']}
```

Notice here that the tuple as a whole counts as one key in this list

Section 23.3: Example 3

Notice in this example that mulato and camel don't show up in our result. Only the last element with the specified key shows up. The last result for c actually wipes out two previous results. But watch the new version where I have the data sorted first on same key.

```
list_things = ['goat', 'dog', 'donkey', 'mulato', 'cow', 'cat', ('persons', 'man', 'woman'), \
               'wombat', 'mongoose', 'malloo', 'camel']
c = groupby(list_things, key=lambda x: x[0])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Results in

```
{'c': ['camel'],
 'd': ['dog', 'donkey'],
 'g': ['goat'],
 'm': ['mongoose', 'malloo'],
 'persons': [('persons', 'man', 'woman')],
 'w': ['wombat']}
```

Sorted Version

```
list_things = ['goat', 'dog', 'donkey', 'mulato', 'cow', 'cat', ('persons', 'man', 'woman'), \
               'wombat', 'mongoose', 'malloo', 'camel']
sorted_list = sorted(list_things, key = lambda x: x[0])
print(sorted_list)
print()
c = groupby(sorted_list, key=lambda x: x[0])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Results in

```
['cow', 'cat', 'camel', 'dog', 'donkey', 'goat', 'mulato', 'mongoose', 'malloo', ('persons', 'man',
'woman'), 'wombat']
```

```
{'c': ['cow', 'cat', 'camel'],
 'd': ['dog', 'donkey'],
 'g': ['goat'],
 'm': ['mulato', 'mongoose', 'malloo'],
 'persons': [('persons', 'man', 'woman')],
 'w': ['wombat']}
```

Chapter 24: Linked lists

A linked list is a collection of nodes, each made up of a reference and a value. Nodes are strung together into a sequence using their references. Linked lists can be used to implement more complex data structures like lists, stacks, queues, and associative arrays.

Section 24.1: Single linked list example

This example implements a linked list with many of the same methods as that of the built-in list object.

```
class Node:
    def __init__(self, val):
        self.data = val
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self, val):
        self.data = val

    def setNext(self, val):
        self.next = val

class LinkedList:
    def __init__(self):
        self.head = None

    def isEmpty(self):
        """Check if the list is empty"""
        return self.head is None

    def add(self, item):
        """Add the item to the list"""
        new_node = Node(item)
        new_node.setNext(self.head)
        self.head = new_node

    def size(self):
        """Return the length/size of the list"""
        count = 0
        current = self.head
        while current is not None:
            count += 1
            current = current.getNext()
        return count

    def search(self, item):
        """Search for item in list. If found, return True. If not found, return False"""
        current = self.head
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                current = current.getNext()
```

```
        return found

    def remove(self, item):
        """Remove item from list. If item is not found in list, raise ValueError"""
        current = self.head
        previous = None
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                previous = current
                current = current.getNext()
        if found:
            if previous is None:
                self.head = current.getNext()
            else:
                previous.setNext(current.getNext())
        else:
            raise ValueError
            print 'Value not found.'

    def insert(self, position, item):
        """
        Insert item at position specified. If position specified is
        out of bounds, raise IndexError
        """
        if position > self.size() - 1:
            raise IndexError
            print "Index out of bounds."
        current = self.head
        previous = None
        pos = 0
        if position is 0:
            self.add(item)
        else:
            new_node = Node(item)
            while pos < position:
                pos += 1
                previous = current
                current = current.getNext()
            previous.setNext(new_node)
            new_node.setNext(current)

    def index(self, item):
        """
        Return the index where item is found.
        If item is not found, return None.
        """
        current = self.head
        pos = 0
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                current = current.getNext()
                pos += 1
        if found:
            pass
        else:
            pos = None
```

```
        return pos

    def pop(self, position = None):
        """
        If no argument is provided, return and remove the item at the head.
        If position is provided, return and remove the item at that position.
        If index is out of bounds, raise IndexError
        """
        if position > self.size():
            print 'Index out of bounds'
            raise IndexError

        current = self.head
        if position is None:
            ret = current.getData()
            self.head = current.getNext()
        else:
            pos = 0
            previous = None
            while pos < position:
                previous = current
                current = current.getNext()
                pos += 1
                ret = current.getData()
            previous.setNext(current.getNext())
        print ret
        return ret

    def append(self, item):
        """Append item to the end of the list"""
        current = self.head
        previous = None
        pos = 0
        length = self.size()
        while pos < length:
            previous = current
            current = current.getNext()
            pos += 1
        new_node = Node(item)
        if previous is None:
            new_node.setNext(current)
            self.head = new_node
        else:
            previous.setNext(new_node)

    def printList(self):
        """Print the list"""
        current = self.head
        while current is not None:
            print current.getData()
            current = current.getNext()
```

Usage functions much like that of the built-in list.

```
ll = LinkedList()
ll.add('l')
ll.add('H')
ll.insert(1,'e')
ll.append('l')
ll.append('o')
ll.printList()
```

```
H
e
l
l
o
```

Chapter 25: Linked List Node

Section 25.1: Write a simple Linked List Node in python

A linked list is either:

- the empty list, represented by None, or
- a node that contains a cargo object and a reference to a linked list.

```
#! /usr/bin/env python

class Node:
      def __init__(self, cargo=None, next=None):
          self.car = cargo
          self.cdr = next
      def __str__(self):
          return str(self.car)


      def display(lst):
          if lst:
             w("%s " % lst)
             display(lst.cdr)
          else:
             w("nil\n")
```

Chapter 26: Filter

Parameter	Details
function	*callable* that determines the condition or `None` then use the identity function for filtering (*positional-only*)
iterable	iterable that will be filtered (*positional-only*)

Section 26.1: Basic use of filter

To `filter` discards elements of a sequence based on some criteria:

```
names = ['Fred', 'Wilma', 'Barney']

def long_name(name):
    return len(name) > 5
```

Python 2.x `Version ≥ 2.0`

```
filter(long_name, names)
# Out: ['Barney']

[name for name in names if len(name) > 5] # equivalent list comprehension
# Out: ['Barney']


from itertools import ifilter
ifilter(long_name, names)       # as generator (similar to python 3.x filter builtin)
# Out: <itertools.ifilter at 0x4197e10>
list(ifilter(long_name, names)) # equivalent to filter with lists
# Out: ['Barney']

(name for name in names if len(name) > 5) # equivalent generator expression
# Out: <generator object <genexpr> at 0x0000000003FD5D38>
```

Python 2.x `Version ≥ 2.6`

```
# Besides the options for older python 2.x versions there is a future_builtin function:
from future_builtins import filter
filter(long_name, names)       # identical to itertools.ifilter
# Out: <itertools.ifilter at 0x3eb0ba8>
```

Python 3.x `Version ≥ 3.0`

```
filter(long_name, names)        # returns a generator
# Out: <filter at 0x1fc6e443470>
list(filter(long_name, names))  # cast to list
# Out: ['Barney']

(name for name in names if len(name) > 5) # equivalent generator expression
# Out: <generator object <genexpr> at 0x000001C6F49BF4C0>
```

Section 26.2: Filter without function

If the function parameter is `None`, then the identity function will be used:

```
list(filter(None, [1, 0, 2, [], '', 'a']))  # discards 0, [] and ''
# Out: [1, 2, 'a']
```

Python 2.x `Version ≥ 2.0.1`

```
[i for i in [1, 0, 2, [], '', 'a'] if i] # equivalent list comprehension
```

Python 3.x `Version ≥ 3.0.0`

```
(i for i in [1, 0, 2, [], '', 'a'] if i) # equivalent generator expression
```

Section 26.3: Filter as short-circuit check

`filter` (python 3.x) and `ifilter` (python 2.x) return a generator so they can be very handy when creating a short-circuit test like `or` or `and`:

Python 2.x `Version ≥ 2.0.1`

```
 # not recommended in real use but keeps the example short:
from itertools import ifilter as filter
```

Python 2.x `Version ≥ 2.6.1`

```
 from future_builtins import filter
```

To find the first element that is smaller than 100:

```
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
def find_something_smaller_than(name_value_tuple):
    print('Check {0}, {1}$'.format(*name_value_tuple)
    return name_value_tuple[1] < 100
next(filter(find_something_smaller_than, car_shop))
# Print: Check Toyota, 1000$
#        Check rectangular tire, 80$
# Out: ('rectangular tire', 80)
```

The `next`-function gives the next (in this case first) element of and is therefore the reason why it's short-circuit.

Section 26.4: Complementary function: filterfalse, ifilterfalse

There is a complementary function for `filter` in the `itertools`-module:

Python 2.x `Version ≥ 2.0.1`

```
 # not recommended in real use but keeps the example valid for python 2.x and python 3.x
from itertools import ifilterfalse as filterfalse
```

Python 3.x `Version ≥ 3.0.0`

```
from itertools import filterfalse
```

which works exactly like the *generator* `filter` but keeps only the elements that are `False`:

```
# Usage without function (None):
list(filterfalse(None, [1, 0, 2, [], '', 'a']))  # discards 1, 2, 'a'
# Out: [0, [], '']
```

```
# Usage with function
names = ['Fred', 'Wilma', 'Barney']

def long_name(name):
    return len(name) > 5

list(filterfalse(long_name, names))
# Out: ['Fred', 'Wilma']
```

```
# Short-circuit usage with next:
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
def find_something_smaller_than(name_value_tuple):
```

```
    print('Check {0}, {1}$'.format(*name_value_tuple)
    return name_value_tuple[1] < 100
next(filterfalse(find_something_smaller_than, car_shop))
# Print: Check Toyota, 1000$
# Out: ('Toyota', 1000)
```

```
# Using an equivalent generator:
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
generator = (car for car in car_shop if not car[1] < 100)
next(generator)
```

Chapter 27: Heapq

Section 27.1: Largest and smallest items in a collection

To find the largest items in a collection, `heapq` module has a function called `nlargest`, we pass it two arguments, the first one is the number of items that we want to retrieve, the second one is the collection name:

```
import heapq


numbers = [1, 4, 2, 100, 20, 50, 32, 200, 150, 8]
print(heapq.nlargest(4, numbers))  # [200, 150, 100, 50]
```

Similarly, to find the smallest items in a collection, we use `nsmallest` function:

```
print(heapq.nsmallest(4, numbers))  # [1, 2, 4, 8]
```

Both `nlargest` and `nsmallest` functions take an optional argument (key parameter) for complicated data structures. The following example shows the use of `age` property to retrieve the oldest and the youngest people from `people` dictionary:

```
people = [
    {'firstname': 'John', 'lastname': 'Doe', 'age': 30},
    {'firstname': 'Jane', 'lastname': 'Doe', 'age': 25},
    {'firstname': 'Janie', 'lastname': 'Doe', 'age': 10},
    {'firstname': 'Jane', 'lastname': 'Roe', 'age': 22},
    {'firstname': 'Johnny', 'lastname': 'Doe', 'age': 12},
    {'firstname': 'John', 'lastname': 'Roe', 'age': 45}
]

oldest = heapq.nlargest(2, people, key=lambda s: s['age'])
print(oldest)
# Output: [{'firstname': 'John', 'age': 45, 'lastname': 'Roe'}, {'firstname': 'John', 'age': 30,
'lastname': 'Doe'}]

youngest = heapq.nsmallest(2, people, key=lambda s: s['age'])
print(youngest)
# Output: [{'firstname': 'Janie', 'age': 10, 'lastname': 'Doe'}, {'firstname': 'Johnny', 'age': 12,
'lastname': 'Doe'}]
```

Section 27.2: Smallest item in a collection

The most interesting property of a `heap` is that its smallest element is always the first element: `heap[0]`

```
import heapq


numbers = [10, 4, 2, 100, 20, 50, 32, 200, 150, 8]

heapq.heapify(numbers)
print(numbers)
# Output: [2, 4, 10, 100, 8, 50, 32, 200, 150, 20]

heapq.heappop(numbers)  # 2
print(numbers)
# Output: [4, 8, 10, 100, 20, 50, 32, 200, 150]
```

```
heapq.heappop(numbers)  # 4
print(numbers)
# Output:  [8, 20, 10, 100, 150, 50, 32, 200]
```

Chapter 28: Tuple

A tuple is an immutable list of values. Tuples are one of Python's simplest and most common collection types, and can be created with the comma operator (`value = 1, 2, 3`).

Section 28.1: Tuple

Syntactically, a tuple is a comma-separated list of values:

```
t = 'a', 'b', 'c', 'd', 'e'
```

Although not necessary, it is common to enclose tuples in parentheses:

```
t = ('a', 'b', 'c', 'd', 'e')
```

Create an empty tuple with parentheses:

```
t0 = ()
type(t0)            # <type 'tuple'>
```

To create a tuple with a single element, you have to include a final comma:

```
t1 = 'a',
type(t1)            # <type 'tuple'>
```

Note that a single value in parentheses is not a tuple:

```
t2 = ('a')
type(t2)            # <type 'str'>
```

To create a singleton tuple it is necessary to have a trailing comma.

```
t2 = ('a',)
type(t2)            # <type 'tuple'>
```

Note that for singleton tuples it's recommended (see PEP8 on trailing commas) to use parentheses. Also, no white space after the trailing comma (see PEP8 on whitespaces)

```
t2 = ('a',)         # PEP8-compliant
t2 = 'a',           # this notation is not recommended by PEP8
t2 = ('a', )        # this notation is not recommended by PEP8
```

Another way to create a tuple is the built-in function `tuple`.

```
t = tuple('lupins')
print(t)            # ('l', 'u', 'p', 'i', 'n', 's')
t = tuple(range(3))
print(t)            # (0, 1, 2)
```

These examples are based on material from the book **Think Python** by Allen B. Downey.

Section 28.2: Tuples are immutable

One of the main differences between `list`s and `tuple`s in Python is that tuples are immutable, that is, one cannot add or modify items once the tuple is initialized. For example:

```
>>> t = (1, 4, 9)
>>> t[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Similarly, tuples don't have `.append` and `.extend` methods as `list` does. Using `+=` is possible, but it changes the binding of the variable, and not the tuple itself:

```
>>> t = (1, 2)
>>> q = t
>>> t += (3, 4)
>>> t
(1, 2, 3, 4)
>>> q
(1, 2)
```

Be careful when placing mutable objects, such as `lists`, inside tuples. This may lead to very confusing outcomes when changing them. For example:

```
>>> t = (1, 2, 3, [1, 2, 3])
(1, 2, 3, [1, 2, 3])
>>> t[3] += [4, 5]
```

Will ***both*** raise an error and change the contents of the list within the tuple:

```
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, 3, [1, 2, 3, 4, 5])
```

You can use the `+=` operator to "append" to a tuple - this works by creating a new tuple with the new element you "appended" and assign it to its current variable; the old tuple is not changed, but replaced!

This avoids converting to and from a list, but this is slow and is a bad practice, especially if you're going to append multiple times.

Section 28.3: Packing and Unpacking Tuples

Tuples in Python are values separated by commas. Enclosing parentheses for inputting tuples are optional, so the two assignments

```
a = 1, 2, 3   # a is the tuple (1, 2, 3)
```

and

```
a = (1, 2, 3) # a is the tuple (1, 2, 3)
```

are equivalent. The assignment `a = 1, 2, 3` is also called *packing* because it packs values together in a tuple.

Note that a one-value tuple is also a tuple. To tell Python that a variable is a tuple and not a single value you can use

a trailing comma

```
a = 1  # a is the value 1
a = 1, # a is the tuple (1,)
```

A comma is needed also if you use parentheses

```
a = (1,) # a is the tuple (1,)
a = (1)  # a is the value 1 and not a tuple
```

To unpack values from a tuple and do multiple assignments use

```
# unpacking AKA multiple assignment
x, y, z = (1, 2, 3)
# x == 1
# y == 2
# z == 3
```

The symbol `_` can be used as a disposable variable name if one only needs some elements of a tuple, acting as a placeholder:

```
a = 1, 2, 3, 4
_, x, y, _ = a
# x == 2
# y == 3
```

Single element tuples:

```
x, = 1,  # x is the value 1
x  = 1,  # x is the tuple (1,)
```

In Python 3 a target variable with a `*` prefix can be used as a *catch-all* variable (see Unpacking Iterables):

Python 3.x `Version ≥ 3.0`

```
first, *more, last = (1, 2, 3, 4, 5)
# first == 1
# more == [2, 3, 4]
# last == 5
```

Section 28.4: Built-in Tuple Functions

Tuples support the following build-in functions

Comparison

If elements are of the same type, python performs the comparison and returns the result. If elements are different types, it checks whether they are numbers.

- If numbers, perform comparison.
- If either element is a number, then the other element is returned.
- Otherwise, types are sorted alphabetically .

If we reached the end of one of the lists, the longer list is "larger." If both list are same it returns 0.

```
tuple1 = ('a', 'b', 'c', 'd', 'e')
tuple2 = ('1','2','3')
```

```
tuple3 = ('a', 'b', 'c', 'd', 'e')

cmp(tuple1, tuple2)
Out: 1

cmp(tuple2, tuple1)
Out: -1

cmp(tuple1, tuple3)
Out: 0
```

Tuple Length

The function `len` returns the total length of the tuple

```
len(tuple1)
Out: 5
```

Max of a tuple

The function `max` returns item from the tuple with the max value

```
max(tuple1)
Out: 'e'

max(tuple2)
Out: '3'
```

Min of a tuple

The function min returns the item from the tuple with the min value

```
min(tuple1)
Out: 'a'

min(tuple2)
Out: '1'
```

Convert a list into tuple

The built-in function `tuple` converts a list into a tuple.

```
list = [1,2,3,4,5]
tuple(list)
Out: (1, 2, 3, 4, 5)
```

Tuple concatenation

Use `+` to concatenate two tuples

```
tuple1 + tuple2
Out: ('a', 'b', 'c', 'd', 'e', '1', '2', '3')
```

Section 28.5: Tuple Are Element-wise Hashable and Equatable

```
hash( (1, 2) )  # ok
hash( ([], {"hello"})  # not ok, since lists and sets are not hashabe
```

Thus a tuple can be put inside a `set` or as a key in a `dict` only if each of its elements can.

```
{ (1, 2) } #  ok
```

```
{ ([], {"hello"}) ) # not ok
```

Section 28.6: Indexing Tuples

```
x = (1, 2, 3)
x[0]  # 1
x[1]  # 2
x[2]  # 3
x[3]  # IndexError: tuple index out of range
```

Indexing with negative numbers will start from the last element as -1:

```
x[-1] # 3
x[-2] # 2
x[-3] # 1
x[-4] # IndexError: tuple index out of range
```

Indexing a range of elements

```
print(x[:-1])   # (1, 2)
print(x[-1:])   # (3,)
print(x[1:3])   # (2, 3)
```

Section 28.7: Reversing Elements

Reverse elements within a tuple

```
colors = "red", "green", "blue"
rev = colors[::-1]
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

Or using reversed (reversed gives an iterable which is converted to a tuple):

```
rev = tuple(reversed(colors))
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

Chapter 29: Basic Input and Output

Section 29.1: Using the print function

Python 3.x `Version ≥ 3.0`

In Python 3, print functionality is in the form of a function:

```
print("This string will be displayed in the output")
# This string will be displayed in the output

print("You can print \n escape characters too.")
# You can print escape characters too.
```

Python 2.x `Version ≥ 2.3`

In Python 2, print was originally a statement, as shown below.

```
print "This string will be displayed in the output"
# This string will be displayed in the output

print "You can print \n escape characters too."
# You can print escape characters too.
```

Note: using `from __future__ import print_function` in Python 2 will allow users to use the `print()` function the same as Python 3 code. This is only available in Python 2.6 and above.

Section 29.2: Input from a File

Input can also be read from files. Files can be opened using the built-in function `open`. Using a `with <command> as <name>` syntax (called a 'Context Manager') makes using `open` and getting a handle for the file super easy:

```
with open('somefile.txt', 'r') as fileobj:
    # write code here using fileobj
```

This ensures that when code execution leaves the block the file is automatically closed.

Files can be opened in different modes. In the above example the file is opened as read-only. To open an existing file for reading only use `r`. If you want to read that file as bytes use `rb`. To append data to an existing file use `a`. Use `w` to create a file or overwrite any existing files of the same name. You can use `r+` to open a file for both reading and writing. The first argument of `open()` is the filename, the second is the mode. If mode is left blank, it will default to `r`.

```
# let's create an example file:
with open('shoppinglist.txt', 'w') as fileobj:
    fileobj.write('tomato\npasta\ngarlic')

with open('shoppinglist.txt', 'r') as fileobj:
    # this method makes a list where each line
    # of the file is an element in the list
    lines = fileobj.readlines()

print(lines)
# ['tomato\n', 'pasta\n', 'garlic']

with open('shoppinglist.txt', 'r') as fileobj:
```

```
    # here we read the whole content into one string:
    content = fileobj.read()
    # get a list of lines, just like int the previous example:
    lines = content.split('\n')

print(lines)
# ['tomato', 'pasta', 'garlic']
```

If the size of the file is tiny, it is safe to read the whole file contents into memory. If the file is very large it is often better to read line-by-line or by chunks, and process the input in the same loop. To do that:

```
with open('shoppinglist.txt', 'r') as fileobj:
    # this method reads line by line:
    lines = []
    for line in fileobj:
        lines.append(line.strip())
```

When reading files, be aware of the operating system-specific line-break characters. Although `for line in fileobj` automatically strips them off, it is always safe to call `strip()` on the lines read, as it is shown above.

Opened files (`fileobj` in the above examples) always point to a specific location in the file. When they are first opened the file handle points to the very beginning of the file, which is the position `0`. The file handle can display its current position with `tell`:

```
fileobj = open('shoppinglist.txt', 'r')
pos = fileobj.tell()
print('We are at %u.' % pos) # We are at 0.
```

Upon reading all the content, the file handler's position will be pointed at the end of the file:

```
content = fileobj.read()
end = fileobj.tell()
print('This file was %u characters long.' % end)
# This file was 22 characters long.
fileobj.close()
```

The file handler position can be set to whatever is needed:

```
fileobj = open('shoppinglist.txt', 'r')
fileobj.seek(7)
pos = fileobj.tell()
print('We are at character #%u.' % pos)
```

You can also read any length from the file content during a given call. To do this pass an argument for `read()`. When `read()` is called with no argument it will read until the end of the file. If you pass an argument it will read that number of bytes or characters, depending on the mode (`rb` and `r` respectively):

```
# reads the next 4 characters
# starting at the current position
next4 = fileobj.read(4)
# what we got?
print(next4) # 'cucu'
# where we are now?
pos = fileobj.tell()
print('We are at %u.' % pos) # We are at 11, as we was at 7, and read 4 chars.

fileobj.close()
```

To demonstrate the difference between characters and bytes:

```
with open('shoppinglist.txt', 'r') as fileobj:
    print(type(fileobj.read())) # <class 'str'>

with open('shoppinglist.txt', 'rb') as fileobj:
    print(type(fileobj.read())) # <class 'bytes'>
```

Section 29.3: Read from stdin

Python programs can read from unix pipelines. Here is a simple example how to read from `stdin`:

```
import sys

for line in sys.stdin:
    print(line)
```

Be aware that `sys.stdin` is a stream. It means that the for-loop will only terminate when the stream has ended.

You can now pipe the output of another program into your python program as follows:

```
$ cat myfile | python myprogram.py
```

In this example `cat myfile` can be any unix command that outputs to `stdout`.

Alternatively, using the fileinput module can come in handy:

```
import fileinput
for line in fileinput.input():
    process(line)
```

Section 29.4: Using input() and raw_input()

Python 2.x `Version ≥ 2.3`

`raw_input` will wait for the user to enter text and then return the result as a string.

```
foo = raw_input("Put a message here that asks the user for input")
```

In the above example `foo` will store whatever input the user provides.

Python 3.x `Version ≥ 3.0`

`input` will wait for the user to enter text and then return the result as a string.

```
foo = input("Put a message here that asks the user for input")
```

In the above example `foo` will store whatever input the user provides.

Section 29.5: Function to prompt user for a number

```
def input_number(msg, err_msg=None):
    while True:
        try:
```

```
            return float(raw_input(msg))
        except ValueError:
            if err_msg is not None:
                print(err_msg)


def input_number(msg, err_msg=None):
    while True:
        try:
            return float(input(msg))
        except ValueError:
            if err_msg is not None:
                print(err_msg)
```

And to use it:

```
user_number = input_number("input a number: ", "that's not a number!")
```

Or, if you do not want an "error message":

```
user_number = input_number("input a number: ")
```

Section 29.6: Printing a string without a newline at the end

Python 2.x `Version ≥ 2.3`

In Python 2.x, to continue a line with `print`, end the `print` statement with a comma. It will automatically add a space.

```
print "Hello,",
print "World!"
# Hello, World!
```

Python 3.x `Version ≥ 3.0`

In Python 3.x, the `print` function has an optional `end` parameter that is what it prints at the end of the given string. By default it's a newline character, so equivalent to this:

```
print("Hello, ", end="\n")
print("World!")
# Hello,
# World!
```

But you could pass in other strings

```
print("Hello, ", end="")
print("World!")
# Hello, World!

print("Hello, ", end="<br>")
print("World!")
# Hello, <br>World!

print("Hello, ", end="BREAK")
print("World!")
# Hello, BREAKWorld!
```

If you want more control over the output, you can use `sys.stdout.write`:

```python
import sys

sys.stdout.write("Hello, ")
sys.stdout.write("World!")
# Hello, World!
```

Chapter 30: Files & Folders I/O

Parameter	Details
filename	the path to your file or, if the file is in the working directory, the filename of your file
access_mode	a string value that determines how the file is opened
buffering	an integer value used for optional line buffering

When it comes to storing, reading, or communicating data, working with the files of an operating system is both necessary and easy with Python. Unlike other languages where file input and output requires complex reading and writing objects, Python simplifies the process only needing commands to open, read/write and close the file. This topic explains how Python can interface with files on the operating system.

Section 30.1: File modes

There are different modes you can open a file with, specified by the `mode` parameter. These include:

- `'r'` - reading mode. The default. It allows you only to read the file, not to modify it. When using this mode the file must exist.
- `'w'` - writing mode. It will create a new file if it does not exist, otherwise will erase the file and allow you to write to it.
- `'a'` - append mode. It will write data to the end of the file. It does not erase the file, and the file must exist for this mode.
- `'rb'` - reading mode in binary. This is similar to `r` except that the reading is forced in binary mode. This is also a default choice.
- `'r+'` - reading mode plus writing mode at the same time. This allows you to read and write into files at the same time without having to use `r` and `w`.
- `'rb+'` - reading and writing mode in binary. The same as `r+` except the data is in binary
- `'wb'` - writing mode in binary. The same as `w` except the data is in binary.
- `'w+'` - writing and reading mode. The exact same as `r+` but if the file does not exist, a new one is made. Otherwise, the file is overwritten.
- `'wb+'` - writing and reading mode in binary mode. The same as `w+` but the data is in binary.
- `'ab'` - appending in binary mode. Similar to `a` except that the data is in binary.
- `'a+'` - appending and reading mode. Similar to `w+` as it will create a new file if the file does not exist. Otherwise, the file pointer is at the end of the file if it exists.
- `'ab+'` - appending and reading mode in binary. The same as `a+` except that the data is in binary.

```
with open(filename, 'r') as f:
    f.read()
with open(filename, 'w') as f:
    f.write(filedata)
with open(filename, 'a') as f:
    f.write('\\n' + newdata)
```

	r	r+	w	w+	a	a+
Read	✓	✓	✗	✓	✗	✓

Write	✗	✓	✓	✓	✓	✓
Creates file	✗	✗	✓	✓	✓	✓
Erases file	✗	✗	✓	✓	✗	✗
Initial position	Start	Start	Start	Start	End	End

Python 3 added a new mode for `exclusive creation` so that you will not accidentally truncate or overwrite and existing file.

- `'x'` - open for exclusive creation, will raise `FileExistsError` if the file already exists
- `'xb'` - open for exclusive creation writing mode in binary. The same as `x` except the data is in binary.
- `'x+'` - reading and writing mode. Similar to `w+` as it will create a new file if the file does not exist. Otherwise, will raise `FileExistsError`.
- `'xb+'` - writing and reading mode. The exact same as `x+` but the data is binary

	x	x+
Read	✗	✓
Write	✓	✓
Creates file	✓	✓
Erases file	✗	✗
Initial position	Start	Start

Allow one to write your file open code in a more pythonic manner:

Python 3.x `Version ≥ 3.3`

```
try:
    with open("fname", "r") as fout:
        # Work with your open file
except FileExistsError:
    # Your error handling goes here
```

In Python 2 you would have done something like

Python 2.x `Version ≥ 2.0`

```
import os.path
if os.path.isfile(fname):
    with open("fname", "w") as fout:
        # Work with your open file
else:
    # Your error handling goes here
```

Section 30.2: Reading a file line-by-line

The simplest way to iterate over a file line-by-line:

```
with open('myfile.txt', 'r') as fp:
    for line in fp:
        print(line)
```

`readline()` allows for more granular control over line-by-line iteration. The example below is equivalent to the one above:

```
with open('myfile.txt', 'r') as fp:
    while True:
        cur_line = fp.readline()
```

```
        # If the result is an empty string
        if cur_line == '':
            # We have reached the end of the file
            break
        print(cur_line)
```

Using the for loop iterator and readline() together is considered bad practice.

More commonly, the `readlines()` method is used to store an iterable collection of the file's lines:

```
with open("myfile.txt", "r") as fp:
    lines = fp.readlines()
for i in range(len(lines)):
    print("Line " + str(i) + ": " + line)
```

This would print the following:

> Line 0: hello
>
> Line 1: world

Section 30.3: Iterate files (recursively)

To iterate all files, including in sub directories, use os.walk:

```
import os
for root, folders, files in os.walk(root_dir):
    for filename in files:
        print root, filename
```

root_dir can be "." to start from current directory, or any other path to start from.

Python 3.x `Version ≥ 3.5`

If you also wish to get information about the file, you may use the more efficient method os.scandir like so:

```
for entry in os.scandir(path):
   if not entry.name.startswith('.') and entry.is_file():
       print(entry.name)
```

Section 30.4: Getting the full contents of a file

The preferred method of file i/o is to use the `with` keyword. This will ensure the file handle is closed once the reading or writing has been completed.

```
with open('myfile.txt') as in_file:
    content = in_file.read()

print(content)
```

or, to handle closing the file manually, you can forgo `with` and simply call `close` yourself:

```
in_file = open('myfile.txt', 'r')
content = in_file.read()
print(content)
```

```
in_file.close()
```

Keep in mind that without using a `with` statement, you might accidentally keep the file open in case an unexpected exception arises like so:

```
in_file = open('myfile.txt', 'r')
raise Exception("oops")
in_file.close()  # This will never be called
```

Section 30.5: Writing to a file

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1")
    f.write("Line 2")
    f.write("Line 3")
    f.write("Line 4")
```

If you open `myfile.txt`, you will see that its contents are:

> Line 1Line 2Line 3Line 4

Python doesn't automatically add line breaks, you need to do that manually:

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1\n")
    f.write("Line 2\n")
    f.write("Line 3\n")
    f.write("Line 4\n")
```

> Line 1
> Line 2
> Line 3
> Line 4

Do not use `os.linesep` as a line terminator when writing files opened in text mode (the default); use `\n` instead.

If you want to specify an encoding, you simply add the `encoding` parameter to the `open` function:

```
with open('my_file.txt', 'w', encoding='utf-8') as f:
    f.write('utf-8 text')
```

It is also possible to use the print statement to write to a file. The mechanics are different in Python 2 vs Python 3, but the concept is the same in that you can take the output that would have gone to the screen and send it to a file instead.

Python 3.x `Version ≥ 3.0`

```
with open('fred.txt', 'w') as outfile:
    s = "I'm Not Dead Yet!"
    print(s) # writes to stdout
    print(s, file = outfile) # writes to outfile

    #Note: it is possible to specify the file parameter AND write to the screen
    #by making sure file ends up with a None value either directly or via a variable
```

```
    myfile = None
    print(s, file = myfile) # writes to stdout
    print(s, file = None)   # writes to stdout
```

In Python 2 you would have done something like

Python 2.x Version ≥ 2.0

```
outfile = open('fred.txt', 'w')
s = "I'm Not Dead Yet!"
print s   # writes to stdout
print >> outfile, s   # writes to outfile
```

Unlike using the write function, the print function does automatically add line breaks.

Section 30.6: Check whether a file or path exists

Employ the EAFP coding style and `try` to open it.

```
import errno

try:
    with open(path) as f:
        # File exists
except IOError as e:
    # Raise the exception if it is not ENOENT (No such file or directory)
    if e.errno != errno.ENOENT:
        raise
    # No such file or directory
```

This will also avoid race-conditions if another process deleted the file between the check and when it is used. This race condition could happen in the following cases:

- Using the `os` module:

```
import os
os.path.isfile('/path/to/some/file.txt')
```

Python 3.x Version ≥ 3.4

- Using `pathlib`:

```
import pathlib
path = pathlib.Path('/path/to/some/file.txt')
if path.is_file():
    ...
```

To check whether a given path exists or not, you can follow the above EAFP procedure, or explicitly check the path:

```
import os
path = "/home/myFiles/directory1"

if os.path.exists(path):
    ## Do stuff
```

Section 30.7: Random File Access Using mmap

Using the `mmap` module allows the user to randomly access locations in a file by mapping the file into memory. This is an alternative to using normal file operations.

```
import mmap

with open('filename.ext', 'r') as fd:
    # 0: map the whole file
    mm = mmap.mmap(fd.fileno(), 0)

    # print characters at indices 5 through 10
    print mm[5:10]

    # print the line starting from mm's current position
    print mm.readline()

    # write a character to the 5th index
    mm[5] = 'a'

    # return mm's position to the beginning of the file
    mm.seek(0)

    # close the mmap object
    mm.close()
```

Section 30.8: Replacing text in a file

```
import fileinput

replacements = {'Search1': 'Replace1',
                'Search2': 'Replace2'}

for line in fileinput.input('filename.txt', inplace=True):
    for search_for in replacements:
        replace_with = replacements[search_for]
        line = line.replace(search_for, replace_with)
    print(line, end='')
```

Section 30.9: Checking if a file is empty

```
>>> import os
>>> os.stat(path_to_file).st_size == 0
```

or

```
>>> import os
>>> os.path.getsize(path_to_file) > 0
```

However, both will throw an exception if the file does not exist. To avoid having to catch such an error, do this:

```
import os
def is_empty_file(fpath):
    return os.path.isfile(fpath) and os.path.getsize(fpath) > 0
```

which will return a `bool` value.

Section 30.10: Read a file between a range of lines

So let's suppose you want to iterate only between some specific lines of a file

You can make use of `itertools` for that

```
import itertools

with open('myfile.txt', 'r') as f:
    for line in itertools.islice(f, 12, 30):
        # do something here
```

This will read through the lines 13 to 20 as in python indexing starts from 0. So line number 1 is indexed as 0

As can also read some extra lines by making use of the `next()` keyword here.

And when you are using the file object as an iterable, please don't use the `readline()` statement here as the two techniques of traversing a file are not to be mixed together

Section 30.11: Copy a directory tree

```
import shutil
source='//192.168.1.2/Daily Reports'
destination='D:\\Reports\\Today'
shutil.copytree(source, destination)
```

The destination directory **must not exist** already.

Section 30.12: Copying contents of one file to a different file

```
with open(input_file, 'r') as in_file, open(output_file, 'w') as out_file:
    for line in in_file:
        out_file.write(line)
```

- Using the `shutil` module:

```
import shutil
shutil.copyfile(src, dst)
```

Chapter 31: os.path

This module implements some useful functions on pathnames. The path parameters can be passed as either strings, or bytes. Applications are encouraged to represent file names as (Unicode) character strings.

Section 31.1: Join Paths

To join two or more path components together, firstly import os module of python and then use following:

```
import os
os.path.join('a', 'b', 'c')
```

The advantage of using os.path is that it allows code to remain compatible over all operating systems, as this uses the separator appropriate for the platform it's running on.

For example, the result of this command on Windows will be:

```
>>> os.path.join('a', 'b', 'c')
'a\b\c'
```

In an Unix OS:

```
>>> os.path.join('a', 'b', 'c')
'a/b/c'
```

Section 31.2: Path Component Manipulation

To split one component off of the path:

```
>>> p = os.path.join(os.getcwd(), 'foo.txt')
>>> p
'/Users/csaftoiu/tmp/foo.txt'
>>> os.path.dirname(p)
'/Users/csaftoiu/tmp'
>>> os.path.basename(p)
'foo.txt'
>>> os.path.split(os.getcwd())
('/Users/csaftoiu/tmp', 'foo.txt')
>>> os.path.splitext(os.path.basename(p))
('foo', '.txt')
```

Section 31.3: Get the parent directory

```
os.path.abspath(os.path.join(PATH_TO_GET_THE_PARENT, os.pardir))
```

Section 31.4: If the given path exists

to check if the given path exists

```
path = '/home/john/temp'
os.path.exists(path)
#this returns false if path doesn't exist or if the path is a broken symbolic link
```

Section 31.5: check if the given path is a directory, file, symbolic link, mount point etc

to check if the given path is a directory

```
dirname = '/home/john/python'
os.path.isdir(dirname)
```

to check if the given path is a file

```
filename = dirname + 'main.py'
os.path.isfile(filename)
```

to check if the given path is symbolic link

```
symlink = dirname + 'some_sym_link'
os.path.islink(symlink)
```

to check if the given path is a mount point

```
mount_path = '/home'
os.path.ismount(mount_path)
```

Section 31.6: Absolute Path from Relative Path

Use `os.path.abspath`:

```
>>> os.getcwd()
'/Users/csaftoiu/tmp'
>>> os.path.abspath('foo')
'/Users/csaftoiu/tmp/foo'
>>> os.path.abspath('../foo')
'/Users/csaftoiu/foo'
>>> os.path.abspath('/foo')
'/foo'
```

Chapter 32: Iterables and Iterators

Section 32.1: Iterator vs Iterable vs Generator

An **iterable** is an object that can return an **iterator**. Any object with state that has an `__iter__` method and returns an iterator is an iterable. It may also be an object *without* state that implements a `__getitem__` method. - The method can take indices (starting from zero) and raise an `IndexError` when the indices are no longer valid.

Python's `str` class is an example of a `__getitem__` iterable.

An **Iterator** is an object that produces the next value in a sequence when you call `next(*object*)` on some object. Moreover, any object with a `__next__` method is an iterator. An iterator raises `StopIteration` after exhausting the iterator and *cannot* be re-used at this point.

Iterable classes:

Iterable classes define an `__iter__` and a `__next__` method. Example of an iterable class:

```
class MyIterable:

    def __iter__(self):

         return self

    def __next__(self):
         #code

#Classic iterable object in older versions of python, __getitem__ is still supported...
class MySequence:

    def __getitem__(self, index):

         if (condition):
             raise IndexError
         return (item)

 #Can produce a plain `iterator` instance by using iter(MySequence())
```

Trying to instantiate the abstract class from the `collections` module to better see this.

Example:

Python 2.x `Version ≥ 2.3`

```
import collections
>>> collections.Iterator()
>>> TypeError: Cant instantiate abstract class Iterator with abstract methods next
```

Python 3.x `Version ≥ 3.0`

```
>>> TypeError: Cant instantiate abstract class Iterator with abstract methods __next__
```

Handle Python 3 compatibility for iterable classes in Python 2 by doing the following:

Python 2.x `Version ≥ 2.3`

```
class MyIterable(object): #or collections.Iterator, which I'd recommend....

     ....
```

```
    def __iter__(self):

        return self

    def next(self): #code

    __next__ = next
```

Both of these are now iterators and can be looped through:

```
ex1 = MyIterableClass()
ex2 = MySequence()

for (item) in (ex1): #code
for (item) in (ex2): #code
```

Generators are simple ways to create iterators. A generator *is* an iterator and an iterator is an iterable.

Section 32.2: Extract values one by one

Start with `iter()` built-in to get **iterator** over iterable and use `next()` to get elements one by one until `StopIteration` is raised signifying the end:

```
s = {1, 2}   # or list or generator or even iterator
i = iter(s)  # get iterator
a = next(i)  # a = 1
b = next(i)  # b = 2
c = next(i)  # raises StopIteration
```

Section 32.3: Iterating over entire iterable

```
s = {1, 2, 3}

# get every element in s
for a in s:
    print a  # prints 1, then 2, then 3

# copy into list
l1 = list(s)  # l1 = [1, 2, 3]

# use list comprehension
l2 = [a * 2 for a in s if a > 2]  # l2 = [6]
```

Section 32.4: Verify only one element in iterable

Use unpacking to extract the first element and ensure it's the only one:

```
a, = iterable

def foo():
    yield 1

a, = foo()  # a = 1

nums = [1, 2, 3]
a, = nums  # ValueError: too many values to unpack
```

Section 32.5: What can be iterable

Iterable can be anything for which items are received *one by one, forward only*. Built-in Python collections are iterable:

```
[1, 2, 3]     # list, iterate over items
(1, 2, 3)     # tuple
{1, 2, 3}     # set
{1: 2, 3: 4}  # dict, iterate over keys
```

Generators return iterables:

```
def foo():  # foo isn't iterable yet...
    yield 1

res = foo()  # ...but res already is
```

Section 32.6: Iterator isn't reentrant!

```
def gen():
    yield 1

iterable = gen()
for a in iterable:
    print a

# What was the first item of iterable? No way to get it now.
# Only to get a new iterator
gen()
```

Chapter 33: Functions

Parameter	Details
arg1, ..., *argN*	Regular arguments
**args*	Unnamed positional arguments
kw1, ..., *kwN*	Keyword-only arguments
***kwargs*	The rest of keyword arguments

Functions in Python provide organized, reusable and modular code to perform a set of specific actions. Functions simplify the coding process, prevent redundant logic, and make the code easier to follow. This topic describes the declaration and utilization of functions in Python.

Python has many *built-in functions* like `print()`, `input()`, `len()`. Besides built-ins you can also create your own functions to do more specific jobs—these are called *user-defined functions*.

Section 33.1: Defining and calling simple functions

Using the `def` statement is the most common way to define a function in python. This statement is a so called *single clause compound statement* with the following syntax:

```
def function_name(parameters):
    statement(s)
```

`function_name` is known as the *identifier* of the function. Since a function definition is an executable statement its execution *binds* the function name to the function object which can be called later on using the identifier.

`parameters` is an optional list of identifiers that get bound to the values supplied as arguments when the function is called. A function may have an arbitrary number of arguments which are separated by commas.

`statement(s)` – also known as the *function body* – are a nonempty sequence of statements executed each time the function is called. This means a function body cannot be empty, just like any *indented block*.

Here's an example of a simple function definition which purpose is to print `Hello` each time it's called:

```
def greet():
    print("Hello")
```

Now let's call the defined `greet()` function:

```
greet()
# Out: Hello
```

That's another example of a function definition which takes one single argument and displays the passed in value each time the function is called:

```
def greet_two(greeting):
    print(greeting)
```

After that the `greet_two()` function must be called with an argument:

```
greet_two("Howdy")
# Out: Howdy
```

Also you can give a default value to that function argument:

```
def greet_two(greeting="Howdy"):
    print(greeting)
```

Now you can call the function without giving a value:

```
greet_two()
# Out: Howdy
```

You'll notice that unlike many other languages, you do not need to explicitly declare a return type of the function. Python functions can return values of any type via the `return` keyword. One function can return any number of different types!

```
def many_types(x):
    if x < 0:
        return "Hello!"
    else:
        return 0

print(many_types(1))
print(many_types(-1))

# Output:
0
Hello!
```

As long as this is handled correctly by the caller, this is perfectly valid Python code.

A function that reaches the end of execution without a return statement will always return `None`:

```
def do_nothing():
    pass

print(do_nothing())
# Out: None
```

As mentioned previously a function definition must have a function body, a nonempty sequence of statements. Therefore the `pass` statement is used as function body, which is a null operation – when it is executed, nothing happens. It does what it means, it skips. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed.

Section 33.2: Defining a function with an arbitrary number of arguments

Arbitrary number of positional arguments:

Defining a function capable of taking an arbitrary number of arguments can be done by prefixing one of the arguments with a `*`

```
def func(*args):
    # args will be a tuple containing all values that are passed in
    for i in args:
        print(i)

func(1, 2, 3)  # Calling it with 3 arguments
```

```
# Out: 1
#      2
#      3

list_of_arg_values = [1, 2, 3]
func(*list_of_arg_values)  # Calling it with list of values, * expands the list
# Out: 1
#      2
#      3

func()  # Calling it without arguments
# No Output
```

You **can't** provide a default for `args`, for example `func(*args=[1, 2, 3])` will raise a syntax error (won't even compile).

You **can't** provide these by name when calling the function, for example `func(*args=[1, 2, 3])` will raise a `TypeError`.

But if you already have your arguments in an array (or any other `Iterable`), you **can** invoke your function like this: `func(*my_stuff)`.

These arguments (`*args`) can be accessed by index, for example `args[0]` will return the first argument

Arbitrary number of keyword arguments

You can take an arbitrary number of arguments with a name by defining an argument in the definition with **two** `*` in front of it:

```
def func(**kwargs):
    # kwargs will be a dictionary containing the names as keys and the values as values
    for name, value in kwargs.items():
        print(name, value)

func(value1=1, value2=2, value3=3)   # Calling it with 3 arguments
# Out: value1 1
#      value2 2
#      value3 3

func()                               # Calling it without arguments
# No Out put

my_dict = {'foo': 1, 'bar': 2}
func(**my_dict)                      # Calling it with a dictionary
# Out: foo 1
#      bar 2
```

You **can't** provide these **without** names, for example `func(1, 2, 3)` will raise a `TypeError`.

`kwargs` is a plain native python dictionary. For example, `args['value1']` will give the value for argument `value1`. Be sure to check beforehand that there is such an argument or a `KeyError` will be raised.

Warning

You can mix these with other optional and required arguments but the order inside the definition matters.

The **positional/keyword** arguments come first. (Required arguments).
Then comes the **arbitrary** `*arg` arguments. (Optional).

Then **keyword-only** arguments come next. (Required).
Finally the **arbitrary keyword** `**kwargs` come. (Optional).

```
#       |-positional-|-optional-|---keyword-only--|-optional-|
def func(arg1, arg2=10 , *args, kwarg1, kwarg2=2, **kwargs):
     pass
```

- `arg1` must be given, otherwise a `TypeError` is raised. It can be given as positional (`func(10)`) or keyword argument (`func(arg1=10)`).
- `kwarg1` must also be given, but it can only be provided as keyword-argument: `func(kwarg1=10)`.
- `arg2` and `kwarg2` are optional. If the value is to be changed the same rules as for `arg1` (either positional or keyword) and `kwarg1` (only keyword) apply.
- `*args` catches additional positional parameters. But note, that `arg1` and `arg2` must be provided as positional arguments to pass arguments to `*args`: `func(1, 1, 1, 1)`.
- `**kwargs` catches all additional keyword parameters. In this case any parameter that is not `arg1`, `arg2`, `kwarg1` or `kwarg2`. For example: `func(kwarg3=10)`.
- In Python 3, you can use `*` alone to indicate that all subsequent arguments must be specified as keywords. For instance the `math.isclose` function in Python 3.5 and higher is defined using `def math.isclose (a, b, *, rel_tol=1e-09, abs_tol=0.0)`, which means the first two arguments can be supplied positionally but the optional third and fourth parameters can only be supplied as keyword arguments.

Python 2.x doesn't support keyword-only parameters. This behavior can be emulated with `kwargs`:

```
def func(arg1, arg2=10, **kwargs):
    try:
        kwarg1 = kwargs.pop("kwarg1")
    except KeyError:
        raise TypeError("missing required keyword-only argument: 'kwarg1'")

    kwarg2 = kwargs.pop("kwarg2", 2)
    # function body ...
```

Note on Naming

The convention of naming optional positional arguments `args` and optional keyword arguments `kwargs` is just a convention you **can** use any names you like **but** it is useful to follow the convention so that others know what you are doing, *or even yourself later* so please do.

Note on Uniqueness

Any function can be defined with **none or one** `*args` and **none or one** `**kwargs` but not with more than one of each. Also `*args` **must** be the last positional argument and `**kwargs` must be the last parameter. Attempting to use more than one of either **will** result in a Syntax Error exception.

Note on Nesting Functions with Optional Arguments

It is possible to nest such functions and the usual convention is to remove the items that the code has already handled **but** if you are passing down the parameters you need to pass optional positional args with a `*` prefix and optional keyword args with a `**` prefix, otherwise args with be passed as a list or tuple and kwargs as a single dictionary. e.g.:

```
def fn(**kwargs):
    print(kwargs)
    f1(**kwargs)
```

```
def f1(**kwargs):
    print(len(kwargs))

fn(a=1, b=2)
# Out:
# {'a': 1, 'b': 2}
# 2
```

Section 33.3: Lambda (Inline/Anonymous) Functions

The `lambda` keyword creates an inline function that contains a single expression. The value of this expression is what the function returns when invoked.

Consider the function:

```
def greeting():
    return "Hello"
```

which, when called as:

```
print(greeting())
```

prints:

```
Hello
```

This can be written as a lambda function as follows:

```
greet_me = lambda: "Hello"
```

> See note at the bottom of this section regarding the assignment of lambdas to variables. Generally, don't do it.

This creates an inline function with the name `greet_me` that returns `Hello`. Note that you don't write `return` when creating a function with lambda. The value after `:` is automatically returned.

Once assigned to a variable, it can be used just like a regular function:

```
print(greet_me())
```

prints:

```
Hello
```

`lambda`s can take arguments, too:

```
strip_and_upper_case = lambda s: s.strip().upper()

strip_and_upper_case("  Hello   ")
```

returns the string:

```
HELLO
```

They can also take arbitrary number of arguments / keyword arguments, like normal functions.

```
greeting = lambda x, *args, **kwargs: print(x, args, kwargs)
greeting('hello', 'world', world='world')
```

prints:

```
hello ('world',) {'world': 'world'}
```

`lambda`s are commonly used for short functions that are convenient to define at the point where they are called (typically with `sorted`, `filter` and `map`).

For example, this line sorts a list of strings ignoring their case and ignoring whitespace at the beginning and at the end:

```
sorted( [" foo ", "    bAR", "BaZ    "], key=lambda s: s.strip().upper())
# Out:
# ['    bAR', 'BaZ    ', ' foo ']
```

Sort list just ignoring whitespaces:

```
sorted( [" foo ", "    bAR", "BaZ    "], key=lambda s: s.strip())
# Out:
# ['BaZ    ', '    bAR', ' foo ']
```

Examples with `map`:

```
sorted( map( lambda s: s.strip().upper(), [" foo ", "    bAR", "BaZ    "]))
# Out:
# ['BAR', 'BAZ', 'FOO']

sorted( map( lambda s: s.strip(), [" foo ", "    bAR", "BaZ    "]))
# Out:
# ['BaZ', 'bAR', 'foo']
```

Examples with numerical lists:

```
my_list = [3, -4, -2, 5, 1, 7]
sorted( my_list, key=lambda x: abs(x))
# Out:
# [1, -2, 3, -4, 5, 7]

list( filter( lambda x: x>0, my_list))
# Out:
# [3, 5, 1, 7]

list( map( lambda x: abs(x), my_list))
# Out:
[3, 4, 2, 5, 1, 7]
```

One can call other functions (with/without arguments) from inside a lambda function.

```
def foo(msg):
    print(msg)

greet = lambda x = "hello world": foo(x)
greet()
```

prints:

```
hello world
```

This is useful because `lambda` may contain only one expression and by using a subsidiary function one can run multiple statements.

NOTE

Bear in mind that PEP-8 (the official Python style guide) does not recommend assigning lambdas to variables (as we did in the first two examples):

> Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier.
>
> Yes:
>
> ```
> def f(x): return 2*x
> ```
>
> No:
>
> ```
> f = lambda x: 2*x
> ```
>
> The first form means that the name of the resulting function object is specifically `f` instead of the generic `<lambda>`. This is more useful for tracebacks and string representations in general. The use of the assignment statement eliminates the sole benefit a lambda expression can offer over an explicit `def` statement (i.e. that it can be embedded inside a larger expression).

Section 33.4: Defining a function with optional arguments

Optional arguments can be defined by assigning (using `=`) a default value to the argument-name:

```
def make(action='nothing'):
    return action
```

Calling this function is possible in 3 different ways:

```
make("fun")
# Out: fun

make(action="sleep")
# Out: sleep

# The argument is optional so the function will use the default value if the argument is
# not passed in.
make()
# Out: nothing
```

> **Warning**
>
> Mutable types (`list`, `dict`, `set`, etc.) should be treated with care when given as **default** attribute. Any mutation of the default argument will change it permanently. See Defining a function with optional mutable arguments.

Section 33.5: Defining a function with optional mutable arguments

There is a problem when using **optional arguments** with a **mutable default type** (described in Defining a function with optional arguments), which can potentially lead to unexpected behaviour.

Explanation

This problem arises because a function's default arguments are initialised **once**, at the point when the function is *defined*, and **not** (like many other languages) when the function is *called*. The default values are stored inside the function object's `__defaults__` member variable.

```
def f(a, b=42, c=[]):
    pass

print(f.__defaults__)
# Out: (42, [])
```

For **immutable** types (see Argument passing and mutability) this is not a problem because there is no way to mutate the variable; it can only ever be reassigned, leaving the original value unchanged. Hence, subsequent are guaranteed to have the same default value. However, for a **mutable** type, the original value can mutate, by making calls to its various member functions. Therefore, successive calls to the function are not guaranteed to have the initial default value.

```
def append(elem, to=[]):
    to.append(elem)      # This call to append() mutates the default variable "to"
    return to

append(1)
# Out: [1]

append(2)  # Appends it to the internally stored list
# Out: [1, 2]

append(3, [])  # Using a new created list gives the expected result
# Out: [3]

# Calling it again without argument will append to the internally stored list again
append(4)
# Out: [1, 2, 4]
```

> **Note:** Some IDEs like PyCharm will issue a warning when a mutable type is specified as a default attribute.

Solution

If you want to ensure that the default argument is always the one you specify in the function definition, then the solution is to **always** use an immutable type as your default argument.

A common idiom to achieve this when a mutable type is needed as the default, is to use `None` (immutable) as the default argument and then assign the actual default value to the argument variable if it is equal to `None`.

```
def append(elem, to=None):
    if to is None:
        to = []
```

```
    to.append(elem)
    return to
```

Section 33.6: Argument passing and mutability

First, some terminology:

- **argument (*actual* parameter):** the actual variable being passed to a function;
- **parameter (*formal* parameter):** the receiving variable that is used in a function.

In Python, arguments are passed by *assignment* (as opposed to other languages, where arguments can be passed by value/reference/pointer).

- Mutating a parameter will mutate the argument (if the argument's type is mutable).

```
def foo(x):         # here x is the parameter
    x[0] = 9        # This mutates the list labelled by both x and y
    print(x)

y = [4, 5, 6]
foo(y)              # call foo with y as argument
# Out: [9, 5, 6]   # list labelled by x has been mutated
print(y)
# Out: [9, 5, 6]   # list labelled by y has been mutated too
```

- Reassigning the parameter won't reassign the argument.

```
def foo(x):         # here x is the parameter, when we call foo(y) we assign y to x
    x[0] = 9        # This mutates the list labelled by both x and y
    x = [1, 2, 3]   # x is now labeling a different list (y is unaffected)
    x[2] = 8        # This mutates x's list, not y's list

y = [4, 5, 6]       # y is the argument, x is the parameter
foo(y)              # Pretend that we wrote "x = y", then go to line 1
y
# Out: [9, 5, 6]
```

In Python, we don't really assign values to variables, instead we *bind* (i.e. assign, attach) variables (considered as *names*) to objects.

- **Immutable:** Integers, strings, tuples, and so on. All operations make copies.
- **Mutable:** Lists, dictionaries, sets, and so on. Operations may or may not mutate.

```
x = [3, 1, 9]
y = x
x.append(5)     # Mutates the list labelled by x and y, both x and y are bound to [3, 1, 9]
x.sort()        # Mutates the list labelled by x and y (in-place sorting)
x = x + [4]     # Does not mutate the list (makes a copy for x only, not y)
z = x           # z is x ([1, 3, 9, 4])
x += [6]        # Mutates the list labelled by both x and z (uses the extend function).
x = sorted(x)   # Does not mutate the list (makes a copy for x only).
x
# Out: [1, 3, 4, 5, 6, 9]
y
# Out: [1, 3, 5, 9]
z
```

```
# Out: [1, 3, 5, 9, 4, 6]
```

Section 33.7: Returning values from functions

Functions can `return` a value that you can use directly:

```
def give_me_five():
    return 5

print(give_me_five())  # Print the returned value
# Out: 5
```

or save the value for later use:

```
num = give_me_five()
print(num)             # Print the saved returned value
# Out: 5
```

or use the value for any operations:

```
print(give_me_five() + 10)
# Out: 15
```

If `return` is encountered in the function the function will be exited immediately and subsequent operations will not be evaluated:

```
def give_me_another_five():
    return 5
    print('This statement will not be printed. Ever.')

print(give_me_another_five())
# Out: 5
```

You can also `return` multiple values (in the form of a tuple):

```
def give_me_two_fives():
    return 5, 5  # Returns two 5

first, second = give_me_two_fives()
print(first)
# Out: 5
print(second)
# Out: 5
```

A function with *no* `return` statement implicitly returns `None`. Similarly a function with a `return` statement, but no return value or variable returns `None`.

Section 33.8: Closure

Closures in Python are created by function calls. Here, the call to `makeInc` creates a binding for `x` that is referenced inside the function `inc`. Each call to `makeInc` creates a new instance of this function, but each instance has a link to a different binding of `x`.

```
def makeInc(x):
  def inc(y):
     # x is "attached" in the definition of inc
```

```
    return y + x

  return inc

incOne = makeInc(1)
incFive = makeInc(5)

incOne(5) # returns 6
incFive(5) # returns 10
```

Notice that while in a regular closure the enclosed function fully inherits all variables from its enclosing environment, in this construct the enclosed function has only read access to the inherited variables but cannot make assignments to them

```
def makeInc(x):
  def inc(y):
    # incrementing x is not allowed
    x += y
    return x

  return inc

incOne = makeInc(1)
incOne(5) # UnboundLocalError: local variable 'x' referenced before assignment
```

Python 3 offers the `nonlocal` statement (Nonlocal Variables) for realizing a full closure with nested functions.

Python 3.x `Version ≥ 3.0`

```
def makeInc(x):
  def inc(y):
    nonlocal x
    # now assigning a value to x is allowed
    x += y
    return x

  return inc

incOne = makeInc(1)
incOne(5) # returns 6
```

Section 33.9: Forcing the use of named parameters

All parameters specified after the first asterisk in the function signature are keyword-only.

```
def f(*a, b):
    pass

f(1, 2, 3)
# TypeError: f() missing 1 required keyword-only argument: 'b'
```

In Python 3 it's possible to put a single asterisk in the function signature to ensure that the remaining arguments may only be passed using keyword arguments.

```
def f(a, b, *, c):
    pass

f(1, 2, 3)
# TypeError: f() takes 2 positional arguments but 3 were given
```

```
f(1, 2, c=3)
# No error
```

Section 33.10: Nested functions

Functions in python are first-class objects. They can be defined in any scope

```
def fibonacci(n):
    def step(a,b):
        return b, a+b
    a, b = 0, 1
    for i in range(n):
        a, b = step(a, b)
    return a
```

Functions capture their enclosing scope can be passed around like any other sort of object

```
def make_adder(n):
    def adder(x):
        return n + x
    return adder
add5 = make_adder(5)
add6 = make_adder(6)
add5(10)
#Out: 15
add6(10)
#Out: 16

def repeatedly_apply(func, n, x):
    for i in range(n):
        x = func(x)
    return x

repeatedly_apply(add5, 5, 1)
#Out: 26
```

Section 33.11: Recursion limit

There is a limit to the depth of possible recursion, which depends on the Python implementation. When the limit is reached, a RuntimeError exception is raised:

```
def cursing(depth):
  try:
    cursing(depth + 1) # actually, re-cursing
  except RuntimeError as RE:
    print('I recursed {} times!'.format(depth))

cursing(0)
# Out: I recursed 1083 times!
```

It is possible to change the recursion depth limit by using `sys.setrecursionlimit(limit)` and check this limit by `sys.getrecursionlimit()`.

```
sys.setrecursionlimit(2000)
cursing(0)
# Out: I recursed 1997 times!
```

From Python 3.5, the exception is a `RecursionError`, which is derived from `RuntimeError`.

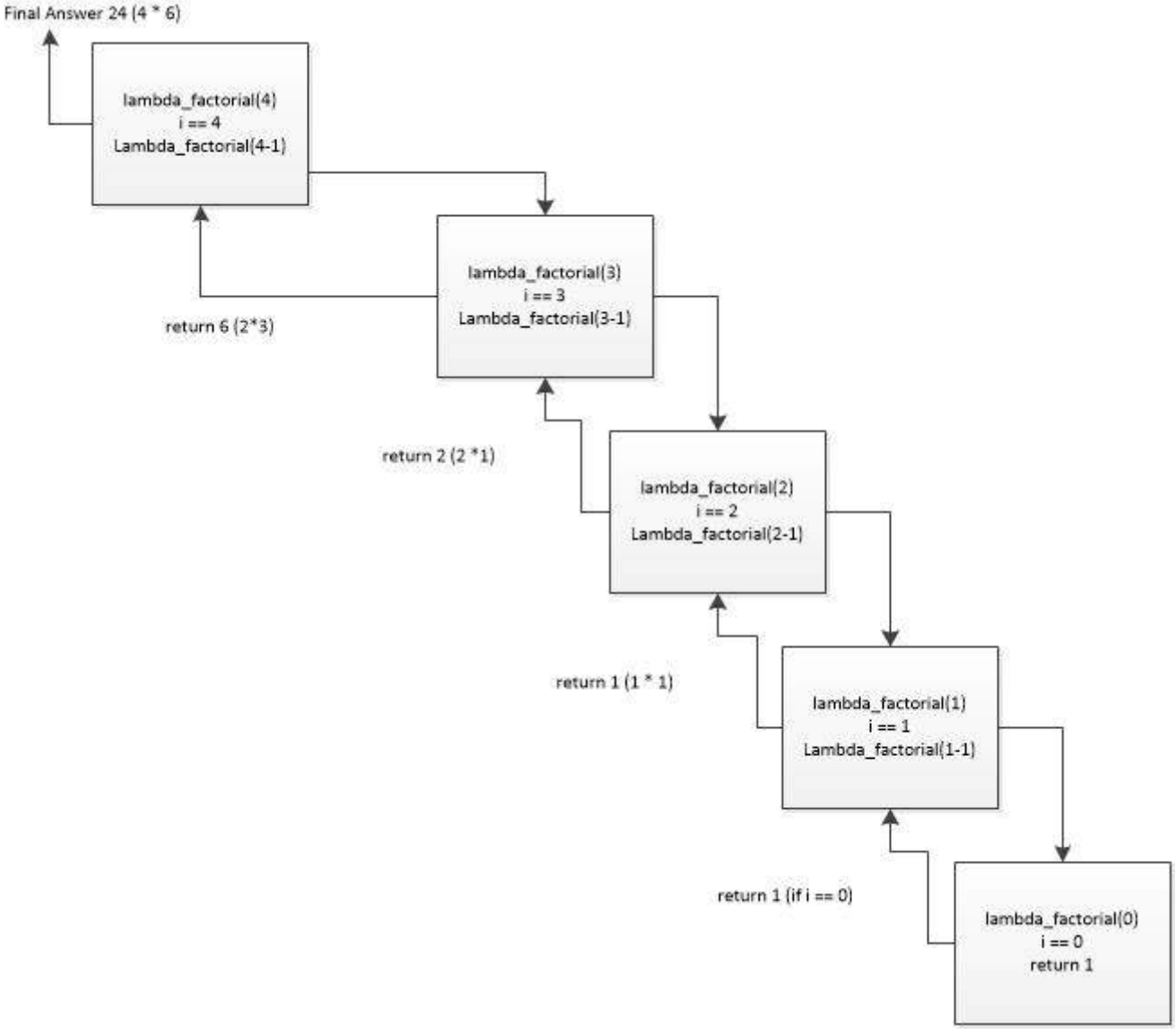
Section 33.12: Recursive Lambda using assigned variable

One method for creating recursive lambda functions involves assigning the function to a variable and then referencing that variable within the function itself. A common example of this is the recursive calculation of the factorial of a number - such as shown in the following code:

```
lambda_factorial = lambda i:1 if i==0 else i*lambda_factorial(i-1)
print(lambda_factorial(4)) # 4 * 3 * 2 * 1 = 12 * 2 = 24
```

Description of code

The lambda function, through its variable assignment, is passed a value (4) which it evaluates and returns 1 if it is 0 or else it returns the current value (`i`) * another calculation by the lambda function of the value - 1 (`i-1`). This continues until the passed value is decremented to 0 (`return 1`). A process which can be visualized as:



Section 33.13: Recursive functions

A recursive function is a function that calls itself in its definition. For example the mathematical function, factorial, defined by `factorial(n) = n*(n-1)*(n-2)*...*3*2*1`. can be programmed as

```
def factorial(n):
    #n here should be an integer
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

the outputs here are:

```
factorial(0)
#out 1
factorial(1)
#out 1
factorial(2)
#out 2
factorial(3)
#out 6
```

as expected. Notice that this function is recursive because the second `return factorial(n-1)`, where the function calls itself in its definition.

Some recursive functions can be implemented using lambda, the factorial function using lambda would be something like this:

```
factorial = lambda n: 1 if n == 0 else n*factorial(n-1)
```

The function outputs the same as above.

Section 33.14: Defining a function with arguments

Arguments are defined in parentheses after the function name:

```
def divide(dividend, divisor):  # The names of the function and its arguments
    # The arguments are available by name in the body of the function
    print(dividend / divisor)
```

The function name and its list of arguments are called the *signature* of the function. Each named argument is effectively a local variable of the function.

When calling the function, give values for the arguments by listing them in order

```
divide(10, 2)
# output: 5
```

or specify them in any order using the names from the function definition:

```
divide(divisor=2, dividend=10)
# output: 5
```

Section 33.15: Iterable and dictionary unpacking

Functions allow you to specify these types of parameters: positional, named, variable positional, Keyword args (kwargs). Here is a clear and concise use of each type.

```
def unpacking(a, b, c=45, d=60, *args, **kwargs):
```

```
    print(a, b, c, d, args, kwargs)

>>> unpacking(1, 2)
1 2 45 60 () {}
>>> unpacking(1, 2, 3, 4)
1 2 3 4 () {}
>>> unpacking(1, 2, c=3, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, c=3)
1 2 3 4 () {}


>>> pair = (3,)
>>> unpacking(1, 2, *pair, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, *pair)
1 2 3 4 () {}
>>> unpacking(1, 2, *pair, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
>>> unpacking(1, 2, c=3, *pair)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'

>>> args_list = [3]
>>> unpacking(1, 2, *args_list, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, *args_list)
1 2 3 4 () {}
>>> unpacking(1, 2, c=3, *args_list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
>>> unpacking(1, 2, *args_list, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'


>>> pair = (3, 4)
>>> unpacking(1, 2, *pair)
1 2 3 4 () {}
>>> unpacking(1, 2, 3, 4, *pair)
1 2 3 4 (3, 4) {}
>>> unpacking(1, 2, d=4, *pair)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, *pair, d=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'



>>> args_list = [3, 4]
>>> unpacking(1, 2, *args_list)
1 2 3 4 () {}
>>> unpacking(1, 2, 3, 4, *args_list)
1 2 3 4 (3, 4) {}
```

```
>>> unpacking(1, 2, d=4, *args_list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, *args_list, d=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'


>>> arg_dict = {'c':3, 'd':4}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {}
>>> arg_dict = {'d':4, 'c':3}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {}
>>> arg_dict = {'c':3, 'd':4, 'not_a_parameter': 75}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {'not_a_parameter': 75}


>>> unpacking(1, 2, *pair, **arg_dict)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, 3, 4, **arg_dict)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'

# Positional arguments take priority over any other form of argument passing
>>> unpacking(1, 2, **arg_dict, c=3)
1 2 3 4 () {'not_a_parameter': 75}
>>> unpacking(1, 2, 3, **arg_dict, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
```

Section 33.16: Defining a function with multiple arguments

One can give a function as many arguments as one wants, the only fixed rules are that each argument name must be unique and that optional arguments must be after the not-optional ones:

```
def func(value1, value2, optionalvalue=10):
    return '{0} {1} {2}'.format(value1, value2, optionalvalue1)
```

When calling the function you can either give each keyword without the name but then the order matters:

```
print(func(1, 'a', 100))
# Out: 1 a 100

print(func('abc', 14))
# abc 14 10
```

Or combine giving the arguments with name and without. Then the ones with name must follow those without but the order of the ones with name doesn't matter:

```
print(func('This', optionalvalue='StackOverflow Documentation', value2='is'))
# Out: This is StackOverflow Documentation
```

Chapter 34: Defining functions with list arguments

Section 34.1: Function and Call

Lists as arguments are just another variable:

```
def func(myList):
    for item in myList:
        print(item)
```

and can be passed in the function call itself:

```
func([1,2,3,5,7])

1
2
3
5
7
```

Or as a variable:

```
aList = ['a','b','c','d']
func(aList)

a
b
c
d
```

Chapter 35: Functional Programming in Python

Functional programming decomposes a problem into a set of functions. Ideally, functions only take inputs and produce outputs, and don't have any internal state that affects the output produced for a given input.below are functional techniques common to many languages: such as lambda, map, reduce.

Section 35.1: Lambda Function

An anonymous, inlined function defined with lambda. The parameters of the lambda are defined to the left of the colon. The function body is defined to the right of the colon. The result of running the function body is (implicitly) returned.

```
s=lambda x:x*x
s(2)     =>4
```

Section 35.2: Map Function

Map takes a function and a collection of items. It makes a new, empty collection, runs the function on each item in the original collection and inserts each return value into the new collection. It returns the new collection.

This is a simple map that takes a list of names and returns a list of the lengths of those names:

```
name_lengths = map(len, ["Mary", "Isla", "Sam"])
print(name_lengths)     =>[4, 4, 3]
```

Section 35.3: Reduce Function

Reduce takes a function and a collection of items. It returns a value that is created by combining the items.

This is a simple reduce. It returns the sum of all the items in the collection.

```
total = reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])
print(total)     =>10
```

Section 35.4: Filter Function

Filter takes a function and a collection. It returns a collection of every item for which the function returned True.

```
arr=[1,2,3,4,5,6]
[i for i in filter(lambda x:x>4,arr)]     # outputs[5,6]
```

Chapter 36: Partial functions

Param	details
x	the number to be raised
y	the exponent
raise	the function to be specialized

As you probably know if you came from OOP school, specializing an abstract class and use it is a practice you should keep in mind when writing your code.

What if you could define an abstract function and specialize it in order to create different versions of it? Thinks it as a sort of *function Inheritance* where you bind specific params to make them reliable for a specific scenario.

Section 36.1: Raise the power

Let's suppose we want raise *x* to a number *y*.

You'd write this as:

```
def raise_power(x, y):
    return x**y
```

What if your *y* value can assume a finite set of values?

Let's suppose *y* can be one of *[3,4,5]* and let's say you don't want offer end user the possibility to use such function since it is very computationally intensive. In fact you would check if provided *y* assumes a valid value and rewrite your function as:

```
def raise(x, y):
    if y in (3,4,5):
        return x**y
    raise NumberNotInRangeException("You should provide a valid exponent")
```

Messy? Let's use the abstract form and specialize it to all three cases: let's implement them **partially**.

```
from functors import partial
raise_to_three = partial(raise, y=3)
raise_to_four = partial(raise, y=4)
raise_to_five = partial(raise, y=5)
```

What happens here? We fixed the *y* params and we defined three different functions.

No need to use the abstract function defined above (you could make it *private*) but you could use **partial applied** functions to deal with raising a number to a fixed value.

Chapter 37: Decorators

Parameter	Details
f	The function to be decorated (wrapped)

Decorator functions are software design patterns. They dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the decorated function. When used correctly, decorators can become powerful tools in the development process. This topic covers implementation and applications of decorator functions in Python.

Section 37.1: Decorator function

Decorators augment the behavior of other functions or methods. Any function that takes a function as a parameter and returns an augmented function can be used as a **decorator**.

```
# This simplest decorator does nothing to the function being decorated. Such
# minimal decorators can occasionally be used as a kind of code markers.
def super_secret_function(f):
    return f

@super_secret_function
def my_function():
    print("This is my secret function.")
```

The `@`-notation is syntactic sugar that is equivalent to the following:

```
my_function = super_secret_function(my_function)
```

It is important to bear this in mind in order to understand how the decorators work. This "unsugared" syntax makes it clear why the decorator function takes a function as an argument, and why it should return another function. It also demonstrates what would happen if you *don't* return a function:

```
def disabled(f):
    """
    This function returns nothing, and hence removes the decorated function
    from the local scope.
    """
    pass

@disabled
def my_function():
    print("This function can no longer be called...")

my_function()
# TypeError: 'NoneType' object is not callable
```

Thus, we usually define a *new function* inside the decorator and return it. This new function would first do something that it needs to do, then call the original function, and finally process the return value. Consider this simple decorator function that prints the arguments that the original function receives, then calls it.

```
#This is the decorator
def print_args(func):
    def inner_func(*args, **kwargs):
        print(args)
        print(kwargs)
```

```
        return func(*args, **kwargs) #Call the original function with its arguments.
    return inner_func

@print_args
def multiply(num_a, num_b):
    return num_a * num_b

print(multiply(3, 5))
#Output:
# (3,5) - This is actually the 'args' that the function receives.
# {} - This is the 'kwargs', empty because we didn't specify keyword arguments.
# 15 - The result of the function.
```

Section 37.2: Decorator class

As mentioned in the introduction, a decorator is a function that can be applied to another function to augment its behavior. The syntactic sugar is equivalent to the following: `my_func = decorator(my_func)`. But what if the `decorator` was instead a class? The syntax would still work, except that now `my_func` gets replaced with an instance of the `decorator` class. If this class implements the `__call__()` magic method, then it would still be possible to use `my_func` as if it was a function:

```
class Decorator(object):
    """Simple decorator class."""

    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print('Before the function call.')
        res = self.func(*args, **kwargs)
        print('After the function call.')
        return res

@Decorator
def testfunc():
    print('Inside the function.')

testfunc()
# Before the function call.
# Inside the function.
# After the function call.
```

Note that a function decorated with a class decorator will no longer be considered a "function" from type-checking perspective:

```
import types
isinstance(testfunc, types.FunctionType)
# False
type(testfunc)
# <class '__main__.Decorator'>
```

Decorating Methods

For decorating methods you need to define an additional `__get__`-method:

```
from types import MethodType

class Decorator(object):
    def __init__(self, func):
```

```
        self.func = func

    def __call__(self, *args, **kwargs):
        print('Inside the decorator.')
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        # Return a Method if it is called on an instance
        return self if instance is None else MethodType(self, instance)

class Test(object):
    @Decorator
    def __init__(self):
        pass

a = Test()
```

> Inside the decorator.

Warning!

Class Decorators only produce one instance for a specific function so decorating a method with a class decorator will share the same decorator between all instances of that class:

```
from types import MethodType

class CountCallsDecorator(object):
    def __init__(self, func):
        self.func = func
        self.ncalls = 0    # Number of calls of this method

    def __call__(self, *args, **kwargs):
        self.ncalls += 1   # Increment the calls counter
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        return self if instance is None else MethodType(self, instance)

class Test(object):
    def __init__(self):
        pass

    @CountCallsDecorator
    def do_something(self):
        return 'something was done'

a = Test()
a.do_something()
a.do_something.ncalls   # 1
b = Test()
b.do_something()
b.do_something.ncalls   # 2
```

Section 37.3: Decorator with arguments (decorator factory)

A decorator takes just one argument: the function to be decorated. There is no way to pass other arguments.

But additional arguments are often desired. The trick is then to make a function which takes arbitrary arguments

and returns a decorator.

Decorator functions

```
def decoratorfactory(message):
    def decorator(func):
        def wrapped_func(*args, **kwargs):
            print('The decorator wants to tell you: {}'.format(message))
            return func(*args, **kwargs)
        return wrapped_func
    return decorator

@decoratorfactory('Hello World')
def test():
    pass

test()
```

> The decorator wants to tell you: Hello World

Important Note:

With such decorator factories you **must** call the decorator with a pair of parentheses:

```
@decoratorfactory # Without parentheses
def test():
    pass

test()
```

> TypeError: decorator() missing 1 required positional argument: 'func'

Decorator classes

```
def decoratorfactory(*decorator_args, **decorator_kwargs):

    class Decorator(object):
        def __init__(self, func):
            self.func = func

        def __call__(self, *args, **kwargs):
            print('Inside the decorator with arguments {}'.format(decorator_args))
            return self.func(*args, **kwargs)

    return Decorator

@decoratorfactory(10)
def test():
    pass

test()
```

> Inside the decorator with arguments (10,)

Section 37.4: Making a decorator look like the decorated function

Decorators normally strip function metadata as they aren't the same. This can cause problems when using meta-programming to dynamically access function metadata. Metadata also includes function's docstrings and its name. `functools.wraps` makes the decorated function look like the original function by copying several attributes to the wrapper function.

```
from functools import wraps
```

The two methods of wrapping a decorator are achieving the same thing in hiding that the original function has been decorated. There is no reason to prefer the function version to the class version unless you're already using one over the other.

As a function

```
def decorator(func):
    # Copies the docstring, name, annotations and module to the decorator
    @wraps(func)
    def wrapped_func(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapped_func

@decorator
def test():
    pass

test.__name__
```

> 'test'

As a class

```
class Decorator(object):
    def __init__(self, func):
        # Copies name, module, annotations and docstring to the instance.
        self._wrapped = wraps(func)(self)

    def __call__(self, *args, **kwargs):
        return self._wrapped(*args, **kwargs)

@Decorator
def test():
    """Docstring of test."""
    pass

test.__doc__
```

> 'Docstring of test.'

Section 37.5: Using a decorator to time a function

```
import time
def timer(func):
    def inner(*args, **kwargs):
```

```
        t1 = time.time()
        f = func(*args, **kwargs)
        t2 = time.time()
        print 'Runtime took {0} seconds'.format(t2-t1)
        return f
    return inner

@timer
def example_function():
    #do stuff


example_function()
```

Section 37.6: Create singleton class with a decorator

A singleton is a pattern that restricts the instantiation of a class to one instance/object. Using a decorator, we can define a class as a singleton by forcing the class to either return an existing instance of the class or create a new instance (if it doesn't exist).

```
def singleton(cls):
    instance = [None]
    def wrapper(*args, **kwargs):
        if instance[0] is None:
            instance[0] = cls(*args, **kwargs)
        return instance[0]

    return wrapper
```

This decorator can be added to any class declaration and will make sure that at most one instance of the class is created. Any subsequent calls will return the already existing class instance.

```
@singleton
class SomeSingletonClass:
    x = 2
    def __init__(self):
        print("Created!")

instance = SomeSingletonClass()  # prints: Created!
instance = SomeSingletonClass()  # doesn't print anything
print(instance.x)                # 2

instance.x = 3
print(SomeSingletonClass().x)    # 3
```

So it doesn't matter whether you refer to the class instance via your local variable or whether you create another "instance", you always get the same object.

Chapter 38: Classes

Python offers itself not only as a popular scripting language, but also supports the object-oriented programming paradigm. Classes describe data and provide methods to manipulate that data, all encompassed under a single object. Furthermore, classes allow for abstraction by separating concrete implementation details from abstract representations of data.

Code utilizing classes is generally easier to read, understand, and maintain.

Section 38.1: Introduction to classes

A class, functions as a template that defines the basic characteristics of a particular object. Here's an example:

```
class Person(object):
    """A simple class."""                          # docstring
    species = "Homo Sapiens"                       # class attribute

    def __init__(self, name):                      # special method
        """This is the initializer. It's a special
        method (see below).
        """
        self.name = name                           # instance attribute

    def __str__(self):                             # special method
        """This method is run when Python tries
        to cast the object to a string. Return
        this string when using print(), etc.
        """
        return self.name

    def rename(self, renamed):                     # regular method
        """Reassign and print the name attribute."""
        self.name = renamed
        print("Now my name is {}".format(self.name))
```

There are a few things to note when looking at the above example.

1. The class is made up of *attributes* (data) and *methods* (functions).
2. Attributes and methods are simply defined as normal variables and functions.
3. As noted in the corresponding docstring, the `__init__()` method is called the *initializer*. It's equivalent to the constructor in other object oriented languages, and is the method that is first run when you create a new object, or new instance of the class.
4. Attributes that apply to the whole class are defined first, and are called *class attributes*.
5. Attributes that apply to a specific instance of a class (an object) are called *instance attributes*. They are generally defined inside `__init__()`; this is not necessary, but it is recommended (since attributes defined outside of `__init__()` run the risk of being accessed before they are defined).
6. Every method, included in the class definition passes the object in question as its first parameter. The word `self` is used for this parameter (usage of `self` is actually by convention, as the word `self` has no inherent meaning in Python, but this is one of Python's most respected conventions, and you should always follow it).
7. Those used to object-oriented programming in other languages may be surprised by a few things. One is that Python has no real concept of `private` elements, so everything, by default, imitates the behavior of the C++/Java `public` keyword. For more information, see the "Private Class Members" example on this page.
8. Some of the class's methods have the following form: `__functionname__(self, other_stuff)`. All such methods are called "magic methods" and are an important part of classes in Python. For instance, operator overloading in Python is implemented with magic methods. For more information, see the relevant

documentation.

Now let's make a few instances of our `Person` class!

```
>>> # Instances
>>> kelly = Person("Kelly")
>>> joseph = Person("Joseph")
>>> john_doe = Person("John Doe")
```

We currently have three `Person` objects, `kelly`, `joseph`, and `john_doe`.

We can access the attributes of the class from each instance using the dot operator `.` Note again the difference between class and instance attributes:

```
>>> # Attributes
>>> kelly.species
'Homo Sapiens'
>>> john_doe.species
'Homo Sapiens'
>>> joseph.species
'Homo Sapiens'
>>> kelly.name
'Kelly'
>>> joseph.name
'Joseph'
```

We can execute the methods of the class using the same dot operator `.`:

```
>>> # Methods
>>> john_doe.__str__()
'John Doe'
>>>  print(john_doe)
'John Doe'
>>>  john_doe.rename("John")
'Now my name is John'
```

Section 38.2: Bound, unbound, and static methods

The idea of bound and unbound methods was removed in Python 3. In Python 3 when you declare a method within a class, you are using a `def` keyword, thus creating a function object. This is a regular function, and the surrounding class works as its namespace. In the following example we declare method `f` within class `A`, and it becomes a function `A.f`:

Python 3.x `Version ≥ 3.0`

```
class A(object):
    def f(self, x):
        return 2 * x
A.f
# <function A.f at ...>  (in Python 3.x)
```

In Python 2 the behavior was different: function objects within the class were implicitly replaced with objects of type `instancemethod`, which were called *unbound methods* because they were not bound to any particular class instance. It was possible to access the underlying function using `.__func__` property.

Python 2.x `Version ≥ 2.3`

```
A.f
```

```
# <unbound method A.f>   (in Python 2.x)
A.f.__class__
# <type 'instancemethod'>
A.f.__func__
# <function f at ...>
```

The latter behaviors are confirmed by inspection - methods are recognized as functions in Python 3, while the distinction is upheld in Python 2.

Python 3.x `Version ≥ 3.0`

```
import inspect

inspect.isfunction(A.f)
# True
inspect.ismethod(A.f)
# False
```

Python 2.x `Version ≥ 2.3`

```
import inspect

inspect.isfunction(A.f)
# False
inspect.ismethod(A.f)
# True
```

In both versions of Python function/method `A.f` can be called directly, provided that you pass an instance of class `A` as the first argument.

```
A.f(1, 7)
# Python 2: TypeError: unbound method f() must be called with
#                      A instance as first argument (got int instance instead)
# Python 3: 14
a = A()
A.f(a, 20)
# Python 2 & 3: 40
```

Now suppose `a` is an instance of class `A`, what is `a.f` then? Well, intuitively this should be the same method `f` of class `A`, only it should somehow "know" that it was applied to the object `a` – in Python this is called method *bound* to `a`.

The nitty-gritty details are as follows: writing `a.f` invokes the magic `__getattribute__` method of `a`, which first checks whether `a` has an attribute named `f` (it doesn't), then checks the class `A` whether it contains a method with such a name (it does), and creates a new object `m` of type `method` which has the reference to the original `A.f` in `m.__func__`, and a reference to the object `a` in `m.__self__`. When this object is called as a function, it simply does the following: `m(...) => m.__func__(m.__self__, ...)`. Thus this object is called a **bound method** because when invoked it knows to supply the object it was bound to as the first argument. (These things work same way in Python 2 and 3).

```
a = A()
a.f
# <bound method A.f of <__main__.A object at ...>>
a.f(2)
# 4

# Note: the bound method object a.f is recreated *every time* you call it:
a.f is a.f  # False
# As a performance optimization you can store the bound method in the object's
# __dict__, in which case the method object will remain fixed:
a.f = a.f
```

```
a.f is a.f  # True
```

Finally, Python has **class methods** and **static methods** – special kinds of methods. Class methods work the same way as regular methods, except that when invoked on an object they bind to the *class* of the object instead of to the object. Thus `m.__self__ = type(a)`. When you call such bound method, it passes the class of `a` as the first argument. Static methods are even simpler: they don't bind anything at all, and simply return the underlying function without any transformations.

```
class D(object):
    multiplier = 2

    @classmethod
    def f(cls, x):
        return cls.multiplier * x

    @staticmethod
    def g(name):
        print("Hello, %s" % name)

D.f
# <bound method type.f of <class '__main__.D'>>
D.f(12)
# 24
D.g
# <function D.g at ...>
D.g("world")
# Hello, world
```

Note that class methods are bound to the class even when accessed on the instance:

```
d = D()
d.multiplier = 1337
(D.multiplier, d.multiplier)
# (2, 1337)
d.f
# <bound method D.f of <class '__main__.D'>>
d.f(10)
# 20
```

It is worth noting that at the lowest level, functions, methods, staticmethods, etc. are actually descriptors that invoke `__get__`, `__set__` and optionally `__del__` special methods. For more details on classmethods and staticmethods:

- What is the difference between @staticmethod and @classmethod in Python?
- Meaning of @classmethod and @staticmethod for beginner?

Section 38.3: Basic inheritance

Inheritance in Python is based on similar ideas used in other object oriented languages like Java, C++ etc. A new class can be derived from an existing class as follows.

```
class BaseClass(object):
    pass

class DerivedClass(BaseClass):
    pass
```

The `BaseClass` is the already existing (*parent*) class, and the `DerivedClass` is the new (*child*) class that inherits (or *subclasses*) attributes from `BaseClass`. **Note**: As of Python 2.2, all classes implicitly inherit from the `object` class, which is the base class for all built-in types.

We define a parent `Rectangle` class in the example below, which implicitly inherits from `object`:

```
class Rectangle():
    def __init__(self, w, h):
        self.w = w
        self.h = h

    def area(self):
        return self.w * self.h

    def perimeter(self):
        return 2 * (self.w + self.h)
```

The `Rectangle` class can be used as a base class for defining a `Square` class, as a square is a special case of rectangle.

```
class Square(Rectangle):
    def __init__(self, s):
        # call parent constructor, w and h are both s
        super(Square, self).__init__(s, s)
        self.s = s
```

The `Square` class will automatically inherit all attributes of the `Rectangle` class as well as the object class. `super()` is used to call the `__init__()` method of `Rectangle` class, essentially calling any overridden method of the base class. **Note**: in Python 3, `super()` does not require arguments.

Derived class objects can access and modify the attributes of its base classes:

```
r.area()
# Output: 12
r.perimeter()
# Output: 14

s.area()
# Output: 4
s.perimeter()
# Output: 8
```

Built-in functions that work with inheritance

`issubclass(DerivedClass, BaseClass)`: returns `True` if `DerivedClass` is a subclass of the `BaseClass`

`isinstance(s, Class)`: returns `True` if s is an instance of `Class` or any of the derived classes of `Class`

```
# subclass check
issubclass(Square, Rectangle)
# Output: True

# instantiate
r = Rectangle(3, 4)
s = Square(2)

isinstance(r, Rectangle)
# Output: True
isinstance(r, Square)
```

```
# Output: False
# A rectangle is not a square

isinstance(s, Rectangle)
# Output: True
# A square is a rectangle
isinstance(s, Square)
# Output: True
```

Section 38.4: Monkey Patching

In this case, "monkey patching" means adding a new variable or method to a class after it's been defined. For instance, say we defined class `A` as

```
class A(object):
    def __init__(self, num):
        self.num = num

    def __add__(self, other):
        return A(self.num + other.num)
```

But now we want to add another function later in the code. Suppose this function is as follows.

```
def get_num(self):
    return self.num
```

But how do we add this as a method in `A`? That's simple we just essentially place that function into `A` with an assignment statement.

```
A.get_num = get_num
```

Why does this work? Because functions are objects just like any other object, and methods are functions that belong to the class.

The function `get_num` shall be available to all existing (already created) as well to the new instances of `A`

These additions are available on all instances of that class (or its subclasses) automatically. For example:

```
foo = A(42)

A.get_num = get_num

bar = A(6);

foo.get_num() # 42

bar.get_num() # 6
```

Note that, unlike some other languages, this technique does not work for certain built-in types, and it is not considered good style.

Section 38.5: New-style vs. old-style classes

Python 2.x `Version ≥ 2.2.0`

New-style classes were introduced in Python 2.2 to unify *classes* and *types*. They inherit from the top-level `object`

type. *A new-style class is a user-defined type*, and is very similar to built-in types.

```
# new-style class
class New(object):
    pass

# new-style instance
new = New()

new.__class__
# <class '__main__.New'>
type(new)
# <class '__main__.New'>
issubclass(New, object)
# True
```

Old-style classes do **not** inherit from `object`. Old-style instances are always implemented with a built-in `instance` type.

```
# old-style class
class Old:
    pass

# old-style instance
old = Old()

old.__class__
# <class __main__.Old at ...>
type(old)
# <type 'instance'>
issubclass(Old, object)
# False
```

Python 3.x `Version ≥ 3.0.0`

In Python 3, old-style classes were removed.

New-style classes in Python 3 implicitly inherit from `object`, so there is no need to specify `MyClass(object)` anymore.

```
class MyClass:
    pass

my_inst = MyClass()

type(my_inst)
# <class '__main__.MyClass'>
my_inst.__class__
# <class '__main__.MyClass'>
issubclass(MyClass, object)
# True
```

Section 38.6: Class methods: alternate initializers

Class methods present alternate ways to build instances of classes. To illustrate, let's look at an example.

Let's suppose we have a relatively simple `Person` class:

```
class Person(object):
```

```
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.full_name = first_name + " " + last_name

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

It might be handy to have a way to build instances of this class specifying a full name instead of first and last name separately. One way to do this would be to have `last_name` be an optional parameter, and assuming that if it isn't given, we passed the full name in:

```
class Person(object):

    def __init__(self, first_name, age, last_name=None):
        if last_name is None:
            self.first_name, self.last_name = first_name.split(" ", 2)
        else:
            self.first_name = first_name
            self.last_name = last_name

        self.full_name = self.first_name + " " + self.last_name
        self.age = age

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

However, there are two main problems with this bit of code:

1. The parameters `first_name` and `last_name` are now misleading, since you can enter a full name for `first_name`. Also, if there are more cases and/or more parameters that have this kind of flexibility, the if/elif/else branching can get annoying fast.
2. Not quite as important, but still worth pointing out: what if `last_name` is `None`, but `first_name` doesn't split into two or more things via spaces? We have yet another layer of input validation and/or exception handling...

Enter class methods. Rather than having a single initializer, we will create a separate initializer, called `from_full_name`, and decorate it with the (built-in) `classmethod` decorator.

```
class Person(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.full_name = first_name + " " + last_name

    @classmethod
    def from_full_name(cls, name, age):
        if " " not in name:
            raise ValueError
        first_name, last_name = name.split(" ", 2)
        return cls(first_name, last_name, age)

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

Notice `cls` instead of `self` as the first argument to `from_full_name`. Class methods are applied to the overall class, *not* an instance of a given class (which is what `self` usually denotes). So, if `cls` is our `Person` class, then the returned value from the `from_full_name` class method is `Person(first_name, last_name, age)`, which uses `Person`'s `__init__` to create an instance of the `Person` class. In particular, if we were to make a subclass `Employee` of `Person`, then `from_full_name` would work in the `Employee` class as well.

To show that this works as expected, let's create instances of `Person` in more than one way without the branching in `__init__`:

```
In [2]: bob = Person("Bob", "Bobberson", 42)

In [3]: alice = Person.from_full_name("Alice Henderson", 31)

In [4]: bob.greet()
Hello, my name is Bob Bobberson.

In [5]: alice.greet()
Hello, my name is Alice Henderson.
```

Other references:

- Python @classmethod and @staticmethod for beginner?
- https://docs.python.org/2/library/functions.html#classmethod
- https://docs.python.org/3.5/library/functions.html#classmethod

Section 38.7: Multiple Inheritance

Python uses the C3 linearization algorithm to determine the order in which to resolve class attributes, including methods. This is known as the Method Resolution Order (MRO).

Here's a simple example:

```
class Foo(object):
    foo = 'attr foo of Foo'


class Bar(object):
    foo = 'attr foo of Bar' # we won't see this.
    bar = 'attr bar of Bar'

class FooBar(Foo, Bar):
    foobar = 'attr foobar of FooBar'
```

Now if we instantiate FooBar, if we look up the foo attribute, we see that Foo's attribute is found first

```
fb = FooBar()
```

and

```
>>> fb.foo
'attr foo of Foo'
```

Here's the MRO of FooBar:

```
>>> FooBar.mro()
[<class '__main__.FooBar'>, <class '__main__.Foo'>, <class '__main__.Bar'>, <type 'object'>]
```

It can be simply stated that Python's MRO algorithm is

1. Depth first (e.g. `FooBar` then `Foo`) unless
2. a shared parent (`object`) is blocked by a child (`Bar`) and
3. no circular relationships allowed.

That is, for example, Bar cannot inherit from FooBar while FooBar inherits from Bar.

For a comprehensive example in Python, see the wikipedia entry.

Another powerful feature in inheritance is `super`. super can fetch parent classes features.

```
class Foo(object):
    def foo_method(self):
        print "foo Method"

class Bar(object):
    def bar_method(self):
        print "bar Method"

class FooBar(Foo, Bar):
    def foo_method(self):
        super(FooBar, self).foo_method()
```

Multiple inheritance with init method of class, when every class has own init method then we try for multiple inheritance then only init method get called of class which is inherit first.

for below example only Foo class **init** method getting called **Bar** class init not getting called

```
    class Foo(object):
        def __init__(self):
            print "foo init"

    class Bar(object):
        def __init__(self):
            print "bar init"

    class FooBar(Foo, Bar):
        def __init__(self):
            print "foobar init"
            super(FooBar, self).__init__()

    a = FooBar()
```

Output:

```
    foobar init
    foo init
```

But it doesn't mean that **Bar** class is not inherit. Instance of final **FooBar** class is also instance of **Bar** class and **Foo** class.

```
print isinstance(a,FooBar)
print isinstance(a,Foo)
```

```
print isinstance(a,Bar)
```

Output:

```
True
True
True
```

Section 38.8: Properties

Python classes support **properties**, which look like regular object variables, but with the possibility of attaching custom behavior and documentation.

```
class MyClass(object):

    def __init__(self):
       self._my_string = ""

    @property
    def string(self):
        """A profoundly important string."""
        return self._my_string

    @string.setter
    def string(self, new_value):
        assert isinstance(new_value, str), \
               "Give me a string, not a %r!" % type(new_value)
        self._my_string = new_value

    @string.deleter
    def x(self):
        self._my_string = None
```

The object's of class `MyClass` will *appear* to have a property `.string`, however it's behavior is now tightly controlled:

```
mc = MyClass()
mc.string = "String!"
print(mc.string)
del mc.string
```

As well as the useful syntax as above, the property syntax allows for validation, or other augmentations to be added to those attributes. This could be especially useful with public APIs - where a level of help should be given to the user.

Another common use of properties is to enable the class to present 'virtual attributes' - attributes which aren't actually stored but are computed only when requested.

```
class Character(object):
    def __init__(name, max_hp):
        self._name = name
        self._hp = max_hp
        self._max_hp = max_hp

    # Make hp read only by not providing a set method
    @property
    def hp(self):
        return self._hp
```

```
    # Make name read only by not providing a set method
    @property
    def name(self):
        return self.name

    def take_damage(self, damage):
        self.hp -= damage
        self.hp = 0 if self.hp <0 else self.hp

    @property
    def is_alive(self):
        return self.hp != 0

    @property
    def is_wounded(self):
        return self.hp < self.max_hp if self.hp > 0 else False

    @property
    def is_dead(self):
        return not self.is_alive

bilbo = Character('Bilbo Baggins', 100)
bilbo.hp
# out : 100
bilbo.hp = 200
# out : AttributeError: can't set attribute
# hp attribute is read only.

bilbo.is_alive
# out : True
bilbo.is_wounded
# out : False
bilbo.is_dead
# out : False

bilbo.take_damage( 50 )

bilbo.hp
# out : 50

bilbo.is_alive
# out : True
bilbo.is_wounded
# out : True
bilbo.is_dead
# out : False

bilbo.take_damage( 50 )
bilbo.hp
# out : 0

bilbo.is_alive
# out : False
bilbo.is_wounded
# out : False
bilbo.is_dead
# out : True
```

Section 38.9: Default values for instance variables

If the variable contains a value of an immutable type (e.g. a string) then it is okay to assign a default value like this

```
class Rectangle(object):
    def __init__(self, width, height, color='blue'):
        self.width = width
        self.height = height
        self.color = color

    def area(self):
        return self.width  * self.height

# Create some instances of the class
default_rectangle = Rectangle(2, 3)
print(default_rectangle.color) # blue

red_rectangle = Rectangle(2, 3, 'red')
print(red_rectangle.color) # red
```

One needs to be careful when initializing mutable objects such as lists in the constructor. Consider the following example:

```
class Rectangle2D(object):
    def __init__(self, width, height, pos=[0,0], color='blue'):
        self.width = width
        self.height = height
        self.pos = pos
        self.color = color

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.pos[0] = 4
r1.pos # [4, 0]
r2.pos # [4, 0] r2's pos has changed as well
```

This behavior is caused by the fact that in Python default parameters are bound at function execution and not at function declaration. To get a default instance variable that's not shared among instances, one should use a construct like this:

```
class Rectangle2D(object):
    def __init__(self, width, height, pos=None, color='blue'):
        self.width = width
        self.height = height
        self.pos = pos or [0, 0] # default value is [0, 0]
        self.color = color

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.pos[0] = 4
r1.pos # [4, 0]
r2.pos # [0, 0] r2's pos hasn't changed
```

See also Mutable Default Arguments and "Least Astonishment" and the Mutable Default Argument.

Section 38.10: Class and instance variables

Instance variables are unique for each instance, while class variables are shared by all instances.

```
class C:
    x = 2  # class variable
```

```
    def __init__(self, y):
        self.y = y  # instance variable

C.x
# 2
C.y
# AttributeError: type object 'C' has no attribute 'y'

c1 = C(3)
c1.x
# 2
c1.y
# 3

c2 = C(4)
c2.x
# 2
c2.y
# 4
```

Class variables can be accessed on instances of this class, but assigning to the class attribute will create an instance variable which shadows the class variable

```
c2.x = 4
c2.x
# 4
C.x
# 2
```

Note that *mutating* class variables from instances can lead to some unexpected consequences.

```
class D:
    x = []
    def __init__(self, item):
        self.x.append(item)  # note that this is not an assignment!

d1 = D(1)
d2 = D(2)

d1.x
# [1, 2]
d2.x
# [1, 2]
D.x
# [1, 2]
```

Section 38.11: Class composition

Class composition allows explicit relations between objects. In this example, people live in cities that belong to countries. Composition allows people to access the number of all people living in their country:

```
class Country(object):
    def __init__(self):
        self.cities=[]

    def addCity(self,city):
        self.cities.append(city)
```

```
class City(object):
    def __init__(self, numPeople):
        self.people = []
        self.numPeople = numPeople


    def addPerson(self, person):
        self.people.append(person)

    def join_country(self,country):
        self.country = country
        country.addCity(self)

        for i in range(self.numPeople):
                person(i).join_city(self)


class Person(object):
    def __init__(self, ID):
        self.ID=ID

    def join_city(self, city):
        self.city = city
        city.addPerson(self)

    def people_in_my_country(self):
        x= sum([len(c.people) for c in self.city.country.cities])
        return x

US=Country()
NYC=City(10).join_country(US)
SF=City(5).join_country(US)

print(US.cities[0].people[0].people_in_my_country())

# 15
```

Section 38.12: Listing All Class Members

The `dir()` function can be used to get a list of the members of a class:

```
dir(Class)
```

For example:

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

It is common to look only for "non-magic" members. This can be done using a simple comprehension that lists members with names not starting with `__`:

```
>>> [m for m in dir(list) if not m.startswith('__')]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

Caveats:

Classes can define a `__dir__()` method. If that method exists calling `dir()` will call `__dir__()`, otherwise Python will try to create a list of members of the class. This means that the dir function can have unexpected results. Two quotes of importance from the official python documentation:

> If the object does not provide **dir**(), the function tries its best to gather information from the object's **dict** attribute, if defined, and from its type object. The resulting list is not necessarily complete, and may be inaccurate when the object has a custom **getattr**().

> **Note:** Because dir() is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class.

Section 38.13: Singleton class

A singleton is a pattern that restricts the instantiation of a class to one instance/object. For more info on python singleton design patterns, see here.

```
class Singleton:
    def __new__(cls):
        try:
            it = cls.__it__
        except AttributeError:
            it = cls.__it__ = object.__new__(cls)
        return it

    def __repr__(self):
        return '<{}>'.format(self.__class__.__name__.upper())

    def __eq__(self, other):
         return other is self
```

Another method is to decorate your class. Following the example from this answer create a Singleton class:

```
class Singleton:
    """
    A non-thread-safe helper class to ease implementing singletons.
    This should be used as a decorator -- not a metaclass -- to the
    class that should be a singleton.

    The decorated class can define one `__init__` function that
    takes only the `self` argument. Other than that, there are
    no restrictions that apply to the decorated class.

    To get the singleton instance, use the `Instance` method. Trying
    to use `__call__` will result in a `TypeError` being raised.

    Limitations: The decorated class cannot be inherited from.

    """

    def __init__(self, decorated):
```

```
        self._decorated = decorated

    def Instance(self):
        """
        Returns the singleton instance. Upon its first call, it creates a
        new instance of the decorated class and calls its `__init__` method.
        On all subsequent calls, the already created instance is returned.

        """
        try:
            return self._instance
        except AttributeError:
            self._instance = self._decorated()
            return self._instance

    def __call__(self):
        raise TypeError('Singletons must be accessed through `Instance()`.')

    def __instancecheck__(self, inst):
        return isinstance(inst, self._decorated)
```

To use you can use the `Instance` method

```
@Singleton
class Single:
    def __init__(self):
        self.name=None
        self.val=0
    def getName(self):
        print(self.name)

x=Single.Instance()
y=Single.Instance()
x.name='I\'m single'
x.getName() # outputs I'm single
y.getName() # outputs I'm single
```

Section 38.14: Descriptors and Dotted Lookups

Descriptors are objects that are (usually) attributes of classes and that have any of `__get__`, `__set__`, or `__delete__` special methods.

Data Descriptors have any of `__set__`, or `__delete__`

These can control the dotted lookup on an instance, and are used to implement functions, `staticmethod`, `classmethod`, and `property`. A dotted lookup (e.g. instance `foo` of class `Foo` looking up attribute `bar` - i.e. `foo.bar`) uses the following algorithm:

1. `bar` is looked up in the class, `Foo`. If it is there and it is a **Data Descriptor**, then the data descriptor is used. That's how `property` is able to control access to data in an instance, and instances cannot override this. If a **Data Descriptor** is not there, then
2. `bar` is looked up in the instance `__dict__`. This is why we can override or block methods being called from an instance with a dotted lookup. If `bar` exists in the instance, it is used. If not, we then
3. look in the class `Foo` for `bar`. If it is a **Descriptor**, then the descriptor protocol is used. This is how functions (in this context, unbound methods), `classmethod`, and `staticmethod` are implemented. Else it simply returns the object there, or there is an `AttributeError`

Chapter 39: Metaclasses

Metaclasses allow you to deeply modify the behaviour of Python classes (in terms of how they're defined, instantiated, accessed, and more) by replacing the `type` metaclass that new classes use by default.

Section 39.1: Basic Metaclasses

When `type` is called with three arguments it behaves as the (meta)class it is, and creates a new instance, ie. it produces a new class/type.

```
Dummy = type('OtherDummy', (), dict(x=1))
Dummy.__class__               # <type 'type'>
Dummy().__class__.__class__   # <type 'type'>
```

It is possible to subclass `type` to create an custom metaclass.

```
class mytype(type):
    def __init__(cls, name, bases, dict):
        # call the base initializer
        type.__init__(cls, name, bases, dict)

        # perform custom initialization...
        cls.__custom_attribute__ = 2
```

Now, we have a new custom `mytype` metaclass which can be used to create classes in the same manner as `type`.

```
MyDummy = mytype('MyDummy', (), dict(x=2))
MyDummy.__class__               # <class '__main__.mytype'>
MyDummy().__class__.__class__   # <class '__main__.mytype'>
MyDummy.__custom_attribute__    # 2
```

When we create a new class using the `class` keyword the metaclass is by default chosen based on upon the baseclasses.

```
>>> class Foo(object):
...     pass

>>> type(Foo)
type
```

In the above example the only baseclass is `object` so our metaclass will be the type of `object`, which is `type`. It is possible override the default, however it depends on whether we use Python 2 or Python 3:

Python 2.x `Version ≤ 2.7`

A special class-level attribute `__metaclass__` can be used to specify the metaclass.

```
class MyDummy(object):
    __metaclass__ = mytype
type(MyDummy)  # <class '__main__.mytype'>
```

Python 3.x `Version ≥ 3.0`

A special `metaclass` keyword argument specify the metaclass.

```
class MyDummy(metaclass=mytype):
```

```
    pass
type(MyDummy)  # <class '__main__.mytype'>
```

Any keyword arguments (except `metaclass`) in the class declaration will be passed to the metaclass. Thus `class MyDummy(metaclass=mytype, x=2)` will pass `x=2` as a keyword argument to the `mytype` constructor.

Read this in-depth description of python meta-classes for more details.

Section 39.2: Singletons using metaclasses

A singleton is a pattern that restricts the instantiation of a class to one instance/object. For more info on python singleton design patterns, see here.

```
class SingletonType(type):
    def __call__(cls, *args, **kwargs):
        try:
            return cls.__instance
        except AttributeError:
            cls.__instance = super(SingletonType, cls).__call__(*args, **kwargs)
            return cls.__instance
```

Python 2.x `Version ≤ 2.7`

```
class MySingleton(object):
    __metaclass__ = SingletonType
```

Python 3.x `Version ≥ 3.0`

```
class MySingleton(metaclass=SingletonType):
    pass
```

```
MySingleton() is MySingleton()  # True, only one instantiation occurs
```

Section 39.3: Using a metaclass

Metaclass syntax

Python 2.x `Version ≤ 2.7`

```
class MyClass(object):
    __metaclass__ = SomeMetaclass
```

Python 3.x `Version ≥ 3.0`

```
class MyClass(metaclass=SomeMetaclass):
    pass
```

Python 2 and 3 compatibility with `six`

```
import six

class MyClass(six.with_metaclass(SomeMetaclass)):
    pass
```

Section 39.4: Introduction to Metaclasses

What is a metaclass?

In Python, everything is an object: integers, strings, lists, even functions and classes themselves are objects. And every object is an instance of a class.

To check the class of an object x, one can call `type(x)`, so:

```
>>> type(5)
<type 'int'>
>>> type(str)
<type 'type'>
>>> type([1, 2, 3])
<type 'list'>

>>> class C(object):
...     pass
...
>>> type(C)
<type 'type'>
```

Most classes in python are instances of `type`. `type` itself is also a class. Such classes whose instances are also classes are called metaclasses.

The Simplest Metaclass

OK, so there is already one metaclass in Python: `type`. Can we create another one?

```
class SimplestMetaclass(type):
    pass

class MyClass(object):
    __metaclass__ = SimplestMetaclass
```

That does not add any functionality, but it is a new metaclass, see that MyClass is now an instance of SimplestMetaclass:

```
>>> type(MyClass)
<class '__main__.SimplestMetaclass'>
```

A Metaclass which does Something

A metaclass which does something usually overrides `type`'s `__new__`, to modify some properties of the class to be created, before calling the original `__new__` which creates the class:

```
class AnotherMetaclass(type):
    def __new__(cls, name, parents, dct):
        # cls is this class
        # name is the name of the class to be created
        # parents is the list of the class's parent classes
        # dct is the list of class's attributes (methods, static variables)

        # here all of the attributes can be modified before creating the class, e.g.

        dct['x'] = 8  # now the class will have a static variable x = 8

        # return value is the new class. super will take care of that
        return super(AnotherMetaclass, cls).__new__(cls, name, parents, dct)
```

Section 39.5: Custom functionality with metaclasses

Functionality in metaclasses can be changed so that whenever a class is built, a string is printed to standard output, or an exception is thrown. This metaclass will print the name of the class being built.

```
class VerboseMetaclass(type):
```

```
    def __new__(cls, class_name, class_parents, class_dict):
        print("Creating class ", class_name)
        new_class = super().__new__(cls, class_name, class_parents, class_dict)
        return new_class
```

You can use the metaclass like so:

```
class Spam(metaclass=VerboseMetaclass):
    def eggs(self):
        print("[insert example string here]")
s = Spam()
s.eggs()
```

The standard output will be:

```
Creating class Spam
[insert example string here]
```

Section 39.6: The default metaclass

You may have heard that everything in Python is an object. It is true, and all objects have a class:

```
>>> type(1)
int
```

The literal 1 is an instance of `int`. Let's declare a class:

```
>>> class Foo(object):
...     pass
...
```

Now let's instantiate it:

```
>>> bar = Foo()
```

What is the class of `bar`?

```
>>> type(bar)
Foo
```

Nice, `bar` is an instance of `Foo`. But what is the class of `Foo` itself?

```
>>> type(Foo)
type
```

Ok, `Foo` itself is an instance of `type`. How about `type` itself?

```
>>> type(type)
type
```

So what is a metaclass? For now let's pretend it is just a fancy name for the class of a class. Takeaways:

- Everything is an object in Python, so everything has a class
- The class of a class is called a metaclass
- The default metaclass is `type`, and by far it is the most common metaclass

But why should you know about metaclasses? Well, Python itself is quite "hackable", and the concept of metaclass is important if you are doing advanced stuff like meta-programming or if you want to control how your classes are initialized.

Chapter 40: String Formatting

When storing and transforming data for humans to see, string formatting can become very important. Python offers a wide variety of string formatting methods which are outlined in this topic.

Section 40.1: Basics of String Formatting

```
foo = 1
bar = 'bar'
baz = 3.14
```

You can use `str.format` to format output. Bracket pairs are replaced with arguments in the order in which the arguments are passed:

```
print('{}, {} and {}'.format(foo, bar, baz))
# Out: "1, bar and 3.14"
```

Indexes can also be specified inside the brackets. The numbers correspond to indexes of the arguments passed to the `str.format` function (0-based).

```
print('{0}, {1}, {2}, and {1}'.format(foo, bar, baz))
# Out: "1, bar, 3.14, and bar"
print('{0}, {1}, {2}, and {3}'.format(foo, bar, baz))
# Out: index out of range error
```

Named arguments can be also used:

```
print("X value is: {x_val}. Y value is: {y_val}.".format(x_val=2, y_val=3))
# Out: "X value is: 2. Y value is: 3."
```

Object attributes can be referenced when passed into `str.format`:

```
class AssignValue(object):
    def __init__(self, value):
        self.value = value
my_value = AssignValue(6)
print('My value is: {0.value}'.format(my_value))  # "0" is optional
# Out: "My value is: 6"
```

Dictionary keys can be used as well:

```
my_dict = {'key': 6, 'other_key': 7}
print("My other key is: {0[other_key]}".format(my_dict))  # "0" is optional
# Out: "My other key is: 7"
```

Same applies to list and tuple indices:

```
my_list = ['zero', 'one', 'two']
print("2nd element is: {0[2]}".format(my_list))  # "0" is optional
# Out: "2nd element is: two"
```

> Note: In addition to `str.format`, Python also provides the modulo operator `%`--also known as the *string formatting* or *interpolation operator* (see PEP 3101)--for formatting strings. `str.format` is a successor of `%`

> and it offers greater flexibility, for instance by making it easier to carry out multiple substitutions.

In addition to argument indexes, you can also include a *format specification* inside the curly brackets. This is an expression that follows special rules and must be preceded by a colon (`:`). See the docs for a full description of format specification. An example of format specification is the alignment directive `:~^20` (`^` stands for center alignment, total width 20, fill with ~ character):

```
'{:~^20}'.format('centered')
# Out: '~~~~~~centered~~~~~~'
```

`format` allows behaviour not possible with `%`, for example repetition of arguments:

```
t = (12, 45, 22222, 103, 6)
print '{0} {2} {1} {2} {3} {2} {4} {2}'.format(*t)
# Out: 12 22222 45 22222 103 22222 6 22222
```

As `format` is a function, it can be used as an argument in other functions:

```
number_list = [12,45,78]
print map('the number is {}'.format, number_list)
# Out: ['the number is 12', 'the number is 45', 'the number is 78']


from datetime import datetime,timedelta

once_upon_a_time = datetime(2010, 7, 1, 12, 0, 0)
delta = timedelta(days=13, hours=8,  minutes=20)

gen = (once_upon_a_time + x * delta for x in xrange(5))

print '\n'.join(map('{:%Y-%m-%d %H:%M:%S}'.format, gen))
#Out: 2010-07-01 12:00:00
#     2010-07-14 20:20:00
#     2010-07-28 04:40:00
#     2010-08-10 13:00:00
#     2010-08-23 21:20:00
```

Section 40.2: Alignment and padding

Python 2.x `Version ≥ 2.6`

The `format()` method can be used to change the alignment of the string. You have to do it with a format expression of the form `:[fill_char][align_operator][width]` where `align_operator` is one of:

- `<` forces the field to be left-aligned within `width`.
- `>` forces the field to be right-aligned within `width`.
- `^` forces the field to be centered within `width`.
- `=` forces the padding to be placed after the sign (numeric types only).

`fill_char` (if omitted default is whitespace) is the character used for the padding.

```
'{:~<9s}, World'.format('Hello')
# 'Hello~~~~, World'

'{:~>9s}, World'.format('Hello')
# '~~~~Hello, World'
```

```
'{:~^9s}'.format('Hello')
# '~~Hello~~'

'{:0=6d}'.format(-123)
# '-00123'
```

Note: you could achieve the same results using the string functions `ljust()`, `rjust()`, `center()`, `zfill()`, however these functions are deprecated since version 2.5.

Section 40.3: Format literals (f-string)

Literal format strings were introduced in PEP 498 (Python3.6 and upwards), allowing you to prepend `f` to the beginning of a string literal to effectively apply `.format` to it with all variables in the current scope.

```
>>> foo = 'bar'
>>> f'Foo is {foo}'
'Foo is bar'
```

This works with more advanced format strings too, including alignment and dot notation.

```
>>> f'{foo:^7s}'
'  bar  '
```

Note: The `f''` does not denote a particular type like `b''` for `bytes` or `u''` for `unicode` in python2. The formatting is immediately applied, resulting in a normal string.

The format strings can also be *nested*:

```
>>> price = 478.23
>>> f"{f'${price:0.2f}':*>20s}"
'**************$478.23'
```

The expressions in an f-string are evaluated in left-to-right order. This is detectable only if the expressions have side effects:

```
>>> def fn(l, incr):
...     result = l[0]
...     l[0] += incr
...     return result
...
>>> lst = [0]
>>> f'{fn(lst,2)} {fn(lst,3)}'
'0 2'
>>> f'{fn(lst,2)} {fn(lst,3)}'
'5 7'
>>> lst
[10]
```

Section 40.4: Float formatting

```
>>> '{0:.0f}'.format(42.12345)
'42'

>>> '{0:.1f}'.format(42.12345)
'42.1'

>>> '{0:.3f}'.format(42.12345)
```

```
'42.123'

>>> '{0:.5f}'.format(42.12345)
'42.12345'

>>> '{0:.7f}'.format(42.12345)
'42.1234500'
```

Same hold for other way of referencing:

```
>>> '{:.3f}'.format(42.12345)
'42.123'

>>> '{answer:.3f}'.format(answer=42.12345)
'42.123'
```

Floating point numbers can also be formatted in scientific notation or as percentages:

```
>>> '{0:.3e}'.format(42.12345)
'4.212e+01'

>>> '{0:.0%}'.format(42.12345)
'4212%'
```

You can also combine the `{0}` and `{name}` notations. This is especially useful when you want to round all variables to a pre-specified number of decimals *with 1 declaration*:

```
>>> s = 'Hello'
>>> a, b, c = 1.12345, 2.34567, 34.5678
>>> digits = 2

>>> '{0}! {1:.{n}f}, {2:.{n}f}, {3:.{n}f}'.format(s, a, b, c, n=digits)
'Hello! 1.12, 2.35, 34.57'
```

Section 40.5: Named placeholders

Format strings may contain named placeholders that are interpolated using keyword arguments to `format`.

Using a dictionary (Python 2.x)

```
>>> data = {'first': 'Hodor', 'last': 'Hodor!'}
>>> '{first} {last}'.format(**data)
'Hodor Hodor!'
```

Using a dictionary (Python 3.2+)

```
>>> '{first} {last}'.format_map(data)
'Hodor Hodor!'
```

`str.format_map` allows to use dictionaries without having to unpack them first. Also the class of `data` (which might be a custom type) is used instead of a newly filled `dict`.

Without a dictionary:

```
>>> '{first} {last}'.format(first='Hodor', last='Hodor!')
'Hodor Hodor!'
```

Section 40.6: String formatting with datetime

Any class can configure its own string formatting syntax through the `__format__` method. A type in the standard Python library that makes handy use of this is the `datetime` type, where one can use `strftime`-like formatting codes directly within `str.format`:

```
>>> from datetime import datetime
>>> 'North America: {dt:%m/%d/%Y}.  ISO: {dt:%Y-%m-%d}.'.format(dt=datetime.now())
'North America: 07/21/2016.  ISO: 2016-07-21.'
```

A full list of list of datetime formatters can be found in the official documentation.

Section 40.7: Formatting Numerical Values

The `.format()` method can interpret a number in different formats, such as:

```
>>> '{:c}'.format(65)    # Unicode character
'A'

>>> '{:d}'.format(0x0a)  # base 10
'10'

>>> '{:n}'.format(0x0a)  # base 10 using current locale for separators
'10'
```

Format integers to different bases (hex, oct, binary)

```
>>> '{0:x}'.format(10) # base 16, lowercase - Hexadecimal
'a'

>>> '{0:X}'.format(10) # base 16, uppercase - Hexadecimal
'A'

>>> '{:o}'.format(10) # base 8 - Octal
'12'

>>> '{:b}'.format(10) # base 2 - Binary
'1010'

>>> '{0:#b}, {0:#o}, {0:#x}'.format(42) # With prefix
'0b101010, 0o52, 0x2a'

>>> '8 bit: {0:08b}; Three bytes: {0:06x}'.format(42) # Add zero padding
'8 bit: 00101010; Three bytes: 00002a'
```

Use formatting to convert an RGB float tuple to a color hex string:

```
>>> r, g, b = (1.0, 0.4, 0.0)
>>> '#{:02X}{:02X}{:02X}'.format(int(255 * r), int(255 * g), int(255 * b))
'#FF6600'
```

Only integers can be converted:

```
>>> '{:x}'.format(42.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Unknown format code 'x' for object of type 'float'
```

Section 40.8: Nested formatting

Some formats can take additional parameters, such as the width of the formatted string, or the alignment:

```
>>> '{:.>10}'.format('foo')
'.......foo'
```

Those can also be provided as parameters to `format` by nesting more `{}` inside the `{}`:

```
>>> '{:.>{}}'.format('foo', 10)
'.......foo'
'{:{}{}{}}'.format('foo', '*', '^', 15)
'******foo******'
```

In the latter example, the format string `'{:{}{}{}}'` is modified to `'{:*^15}'` (i.e. "center and pad with * to total length of 15") before applying it to the actual string `'foo'` to be formatted that way.

This can be useful in cases when parameters are not known beforehand, for instances when aligning tabular data:

```
>>> data = ["a", "bbbbbbb", "ccc"]
>>> m = max(map(len, data))
>>> for d in data:
...     print('{:>{}}'.format(d, m))
      a
bbbbbbb
    ccc
```

Section 40.9: Format using Getitem and Getattr

Any data structure that supports `__getitem__` can have their nested structure formatted:

```
person = {'first': 'Arthur', 'last': 'Dent'}
'{p[first]} {p[last]}'.format(p=person)
# 'Arthur Dent'
```

Object attributes can be accessed using `getattr()`:

```
class Person(object):
    first = 'Zaphod'
    last = 'Beeblebrox'

'{p.first} {p.last}'.format(p=Person())
# 'Zaphod Beeblebrox'
```

Section 40.10: Padding and truncating strings, combined

Say you want to print variables in a 3 character column.

Note: doubling { and } escapes them.

```
s = """

pad
{{:3}}             :{a:3}:

truncate
```

```
{{:.3}}            :{e:.3}:

combined
{{:>3.3}}          :{a:>3.3}:
{{:3.3}}           :{a:3.3}:
{{:3.3}}           :{c:3.3}:
{{:3.3}}           :{e:3.3}:
"""

print (s.format(a="1"*1, c="3"*3, e="5"*5))
```

Output:

```
pad
{:3}               :1  :

truncate
{:.3}              :555:

combined
{:>3.3}            :  1:
{:3.3}             :1  :
{:3.3}             :333:
{:3.3}             :555:
```

Section 40.11: Custom formatting for a class

Note:

> Everything below applies to the `str.format` method, as well as the `format` function. In the text below, the two are interchangeable.

For every value which is passed to the `format` function, Python looks for a `__format__` method for that argument. Your own custom class can therefore have their own `__format__` method to determine how the `format` function will display and format your class and it's attributes.

This is different than the `__str__` method, as in the `__format__` method you can take into account the formatting language, including alignment, field width etc, and even (if you wish) implement your own format specifiers, and your own formatting language extensions.[1]

```
object.__format__(self, format_spec)
```

For example:

```
# Example in Python 2 - but can be easily applied to Python 3

class Example(object):
    def __init__(self,a,b,c):
        self.a, self.b, self.c = a,b,c

    def __format__(self, format_spec):
        """ Implement special semantics for the 's' format specifier """
        # Reject anything that isn't an s
        if format_spec[-1] != 's':
            raise ValueError('{} format specifier not understood for this object',
format_spec[:-1])
```

```
        # Output in this example will be (<a>,<b>,<c>)
        raw = "(" + ",".join([str(self.a), str(self.b), str(self.c)]) + ")"
        # Honor the format language by using the inbuilt string format
        # Since we know the original format_spec ends in an 's'
        # we can take advantage of the str.format method with a
        # string argument we constructed above
        return "{r:{f}}".format( r=raw, f=format_spec )

inst = Example(1,2,3)
print "{0:>20s}".format( inst )
# out :              (1,2,3)
# Note how the right align and field width of 20 has been honored.
```

Note:

> If your custom class does not have a custom `__format__` method and an instance of the class is passed to the `format` function, **Python2** will always use the return value of the `__str__` method or `__repr__` method to determine what to print (and if neither exist then the default `repr` will be used), and you will need to use the `s` format specifier to format this. With **Python3**, to pass your custom class to the `format` function, you will need define `__format__` method on your custom class.

Chapter 41: String Methods

Section 41.1: Changing the capitalization of a string

Python's string type provides many functions that act on the capitalization of a string. These include:

- `str.casefold`
- `str.upper`
- `str.lower`
- `str.capitalize`
- `str.title`
- `str.swapcase`

With unicode strings (the default in Python 3), these operations are **not** 1:1 mappings or reversible. Most of these operations are intended for display purposes, rather than normalization.

Python 3.x `Version ≥ 3.3`

`str.casefold()`

`str.casefold` creates a lowercase string that is suitable for case insensitive comparisons. This is more aggressive than `str.lower` and may modify strings that are already in lowercase or cause strings to grow in length, and is not intended for display purposes.

```
"XßΣ".casefold()
# 'xssσ'

"XßΣ".lower()
# 'xßς'
```

The transformations that take place under casefolding are defined by the Unicode Consortium in the CaseFolding.txt file on their website.

`str.upper()`

`str.upper` takes every character in a string and converts it to its uppercase equivalent, for example:

```
"This is a 'string'.".upper()
# "THIS IS A 'STRING'."
```

`str.lower()`

`str.lower` does the opposite; it takes every character in a string and converts it to its lowercase equivalent:

```
"This IS a 'string'.".lower()
# "this is a 'string'."
```

`str.capitalize()`

`str.capitalize` returns a capitalized version of the string, that is, it makes the first character have upper case and the rest lower:

```
"this Is A 'String'.".capitalize() # Capitalizes the first character and lowercases all others
```

```
# "This is a 'string'."
```

`str.title()`

`str.title` returns the title cased version of the string, that is, every letter in the beginning of a word is made upper case and all others are made lower case:

```
"this Is a 'String'".title()
# "This Is A 'String'"
```

`str.swapcase()`

`str.swapcase` returns a new string object in which all lower case characters are swapped to upper case and all upper case characters to lower:

```
"this iS A STRiNG".swapcase() #Swaps case of each character
# "THIS Is a strIng"
```

Usage as `str` class methods

It is worth noting that these methods may be called either on string objects (as shown above) or as a class method of the `str` class (with an explicit call to `str.upper`, etc.)

```
str.upper("This is a 'string'")
# "THIS IS A 'STRING'"
```

This is most useful when applying one of these methods to many strings at once in say, a *`map`* function.

```
map(str.upper,["These","are","some","'strings'"])
# ['THESE', 'ARE', 'SOME', "'STRINGS'"]
```

Section 41.2: str.translate: Translating characters in a string

Python supports a `translate` method on the `str` type which allows you to specify the translation table (used for replacements) as well as any characters which should be deleted in the process.

str.translate(table[, deletechars])

Parameter	Description
`table`	It is a lookup table that defines the mapping from one character to another.
`deletechars`	A list of characters which are to be removed from the string.

The `maketrans` method (`str.maketrans` in Python 3 and `string.maketrans` in Python 2) allows you to generate a translation table.

```
>>> translation_table = str.maketrans("aeiou", "12345")
>>> my_string = "This is a string!"
>>> translated = my_string.translate(translation_table)
'Th3s 3s 1 str3ng!'
```

The `translate` method returns a string which is a translated copy of the original string.

You can set the `table` argument to `None` if you only need to delete characters.

```
>>> 'this syntax is very useful'.translate(None, 'aeiou')
```

```
'ths syntx s vry sfl'
```

Section 41.3: str.format and f-strings: Format values into a string

Python provides string interpolation and formatting functionality through the `str.format` function, introduced in version 2.6 and f-strings introduced in version 3.6.

Given the following variables:

```
i = 10
f = 1.5
s = "foo"
l = ['a', 1, 2]
d = {'a': 1, 2: 'foo'}
```

The following statements are all equivalent

```
"10 1.5 foo ['a', 1, 2] {'a': 1, 2: 'foo'}"
```

```
>>> "{} {} {} {} {}".format(i, f, s, l, d)

>>> str.format("{} {} {} {} {}", i, f, s, l, d)

>>> "{0} {1} {2} {3} {4}".format(i, f, s, l, d)

>>> "{0:d} {1:0.1f} {2} {3!r} {4!r}".format(i, f, s, l, d)

>>> "{i:d} {f:0.1f} {s} {l!r} {d!r}".format(i=i, f=f, s=s, l=l, d=d)
```

```
>>> f"{i} {f} {s} {l} {d}"

>>> f"{i:d} {f:0.1f} {s} {l!r} {d!r}"
```

For reference, Python also supports C-style qualifiers for string formatting. The examples below are equivalent to those above, but the `str.format` versions are preferred due to benefits in flexibility, consistency of notation, and extensibility:

```
"%d %0.1f %s %r %r" % (i, f, s, l, d)

"%(i)d %(f)0.1f %(s)s %(l)r %(d)r" % dict(i=i, f=f, s=s, l=l, d=d)
```

The braces uses for interpolation in `str.format` can also be numbered to reduce duplication when formatting strings. For example, the following are equivalent:

```
"I am from Australia. I love cupcakes from Australia!"
```

```
>>> "I am from {}. I love cupcakes from {}!".format("Australia", "Australia")

>>> "I am from {0}. I love cupcakes from {0}!".format("Australia")
```

While the official python documentation is, as usual, thorough enough, pyformat.info has a great set of examples with detailed explanations.

Additionally, the { and } characters can be escaped by using double brackets:

```
"{'a': 5, 'b': 6}"
```

```
>>> "{{'{}': {}, '{}': {}}}".format("a", 5, "b", 6)

>>> f"{{'{'a'}': {5}, '{'b'}': {6}}}"
```

See String Formatting for additional information. `str.format()` was proposed in PEP 3101 and f-strings in PEP 498.

Section 41.4: String module's useful constants

Python's `string` module provides constants for string related operations. To use them, import the `string` module:

```
>>> import string
```

`string.ascii_letters`:

Concatenation of `ascii_lowercase` and `ascii_uppercase`:

```
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

`string.ascii_lowercase`:

Contains all lower case ASCII characters:

```
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

`string.ascii_uppercase`:

Contains all upper case ASCII characters:

```
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

`string.digits`:

Contains all decimal digit characters:

```
>>> string.digits
'0123456789'
```

`string.hexdigits`:

Contains all hex digit characters:

```
>>> string.hexdigits
'0123456789abcdefABCDEF'
```

`string.octaldigits`:

Contains all octal digit characters:

```
>>> string.octaldigits
'01234567'
```

`string.punctuation`:

Contains all characters which are considered punctuation in the `C` locale:

```
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

`string.whitespace`:

Contains all ASCII characters considered whitespace:

```
>>> string.whitespace
' \t\n\r\x0b\x0c'
```

In script mode, `print(string.whitespace)` will print the actual characters, use `str` to get the string returned above.

`string.printable`:

Contains all characters which are considered printable; a combination of `string.digits`, `string.ascii_letters`, `string.punctuation`, and `string.whitespace`.

```
>>> string.printable
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~
\t\n\r\x0b\x0c'
```

Section 41.5: Stripping unwanted leading/trailing characters from a string

Three methods are provided that offer the ability to strip leading and trailing characters from a string: `str.strip`, `str.rstrip` and `str.lstrip`. All three methods have the same signature and all three return a new string object with unwanted characters removed.

`str.strip([chars])`

`str.strip` acts on a given string and removes (strips) any leading or trailing characters contained in the argument `chars`; if `chars` is not supplied or is `None`, all white space characters are removed by default. For example:

```
>>> "    a line with leading and trailing space     ".strip()
'a line with leading and trailing space'
```

If `chars` is supplied, all characters contained in it are removed from the string, which is returned. For example:

```
>>> ">>> a Python prompt".strip('> ')  # strips '>' character and space character
'a Python prompt'
```

`str.rstrip([chars])` and `str.lstrip([chars])`

These methods have similar semantics and arguments with `str.strip()`, their difference lies in the direction from which they start. `str.rstrip()` starts from the end of the string while `str.lstrip()` splits from the start of the string.

For example, using `str.rstrip`:

```
>>> "     spacious string      ".rstrip()
'     spacious string'
```

While, using `str.lstrip`:

```
>>> "     spacious string      ".rstrip()
'spacious string      '
```

Section 41.6: Reversing a string

A string can reversed using the built-in `reversed()` function, which takes a string and returns an iterator in reverse order.

```
>>> reversed('hello')
<reversed object at 0x0000000000000000>
>>> [char for char in reversed('hello')]
['o', 'l', 'l', 'e', 'h']
```

`reversed()` can be wrapped in a call to `''.join()` to make a string from the iterator.

```
>>> ''.join(reversed('hello'))
'olleh'
```

While using `reversed()` might be more readable to uninitiated Python users, using extended slicing with a step of `-1` is faster and more concise. Here , try to implement it as function:

```
>>> def reversed_string(main_string):
...     return main_string[::-1]
...
>>> reversed_string('hello')
'olleh'
```

Section 41.7: Split a string based on a delimiter into a list of strings

`str.split(sep=None, maxsplit=-1)`

`str.split` takes a string and returns a list of substrings of the original string. The behavior differs depending on whether the `sep` argument is provided or omitted.

If `sep` isn't provided, or is `None`, then the splitting takes place wherever there is whitespace. However, leading and trailing whitespace is ignored, and multiple consecutive whitespace characters are treated the same as a single whitespace character:

```
>>> "This is a sentence.".split()
['This', 'is', 'a', 'sentence.']

>>> " This is    a sentence.  ".split()
['This', 'is', 'a', 'sentence.']

>>> "            ".split()
[]
```

The `sep` parameter can be used to define a delimiter string. The original string is split where the delimiter string occurs, and the delimiter itself is discarded. Multiple consecutive delimiters are *not* treated the same as a single occurrence, but rather cause empty strings to be created.

```
>>> "This is a sentence.".split(' ')
['This', 'is', 'a', 'sentence.']

>>> "Earth,Stars,Sun,Moon".split(',')
['Earth', 'Stars', 'Sun', 'Moon']

>>> " This is    a sentence.  ".split(' ')
['', 'This', 'is', '', '', '', 'a', 'sentence.', '', '']

>>> "This is a sentence.".split('e')
['This is a s', 'nt', 'nc', '.']

>>> "This is a sentence.".split('en')
['This is a s', 't', 'ce.']
```

The default is to split on *every* occurrence of the delimiter, however the `maxsplit` parameter limits the number of splittings that occur. The default value of `-1` means no limit:

```
>>> "This is a sentence.".split('e', maxsplit=0)
['This is a sentence.']

>>> "This is a sentence.".split('e', maxsplit=1)
['This is a s', 'ntence.']

>>> "This is a sentence.".split('e', maxsplit=2)
['This is a s', 'nt', 'nce.']

>>> "This is a sentence.".split('e', maxsplit=-1)
['This is a s', 'nt', 'nc', '.']
```

```
str.rsplit(sep=None, maxsplit=-1)
```

`str.rsplit` ("right split") differs from `str.split` ("left split") when `maxsplit` is specified. The splitting starts at the end of the string rather than at the beginning:

```
>>> "This is a sentence.".rsplit('e', maxsplit=1)
['This is a sentenc', '.']

>>> "This is a sentence.".rsplit('e', maxsplit=2)
['This is a sent', 'nc', '.']
```

Note: Python specifies the maximum number of *splits* performed, while most other programming languages specify the maximum number of *substrings* created. This may create confusion when porting or comparing code.

Section 41.8: Replace all occurrences of one substring with another substring

Python's `str` type also has a method for replacing occurrences of one sub-string with another sub-string in a given string. For more demanding cases, one can use `re.sub`.

```
str.replace(old, new[, count]):
```

`str.replace` takes two arguments `old` and `new` containing the `old` sub-string which is to be replaced by the `new` sub-string. The optional argument `count` specifies the number of replacements to be made:

For example, in order to replace `'foo'` with `'spam'` in the following string, we can call `str.replace` with `old = 'foo'` and `new = 'spam'`:

```
>>> "Make sure to foo your sentence.".replace('foo', 'spam')
"Make sure to spam your sentence."
```

If the given string contains multiple examples that match the `old` argument, **all** occurrences are replaced with the value supplied in `new`:

```
>>> "It can foo multiple examples of foo if you want.".replace('foo', 'spam')
"It can spam multiple examples of spam if you want."
```

unless, of course, we supply a value for `count`. In this case `count` occurrences are going to get replaced:

```
>>> """It can foo multiple examples of foo if you want, \
... or you can limit the foo with the third argument.""".replace('foo', 'spam', 1)
'It can spam multiple examples of foo if you want, or you can limit the foo with the third
argument.'
```

Section 41.9: Testing what a string is composed of

Python's `str` type also features a number of methods that can be used to evaluate the contents of a string. These are `str.isalpha`, `str.isdigit`, `str.isalnum`, `str.isspace`. Capitalization can be tested with `str.isupper`, `str.islower` and `str.istitle`.

`str.isalpha`

`str.isalpha` takes no arguments and returns `True` if the all characters in a given string are alphabetic, for example:

```
>>> "Hello World".isalpha()  # contains a space
False
>>> "Hello2World".isalpha()  # contains a number
False
>>> "HelloWorld!".isalpha()  # contains punctuation
False
>>> "HelloWorld".isalpha()
True
```

As an edge case, the empty string evaluates to `False` when used with `"".isalpha()`.

`str.isupper`, `str.islower`, `str.istitle`

These methods test the capitalization in a given string.

`str.isupper` is a method that returns `True` if all characters in a given string are uppercase and `False` otherwise.

```
>>> "HeLLO WORLD".isupper()
False
>>> "HELLO WORLD".isupper()
True
>>> "".isupper()
```

```
False
```

Conversely, `str.islower` is a method that returns `True` if all characters in a given string are lowercase and `False` otherwise.

```
>>> "Hello world".islower()
False
>>> "hello world".islower()
True
>>> "".islower()
False
```

`str.istitle` returns `True` if the given string is title cased; that is, every word begins with an uppercase character followed by lowercase characters.

```
>>> "hello world".istitle()
False
>>> "Hello world".istitle()
False
>>> "Hello World".istitle()
True
>>> "".istitle()
False
```

`str.isdecimal`, `str.isdigit`, `str.isnumeric`

`str.isdecimal` returns whether the string is a sequence of decimal digits, suitable for representing a decimal number.

`str.isdigit` includes digits not in a form suitable for representing a decimal number, such as superscript digits.

`str.isnumeric` includes any number values, even if not digits, such as values outside the range 0-9.

```
           isdecimal    isdigit   isnumeric

12345       True        True       True
?2??5       True        True       True
?²³?????     False       True       True
??         False       False      True
Five        False       False      False
```

Bytestrings (`bytes` in Python 3, `str` in Python 2), only support `isdigit`, which only checks for basic ASCII digits.

As with `str.isalpha`, the empty string evaluates to `False`.

`str.isalnum`

This is a combination of `str.isalpha` and `str.isnumeric`, specifically it evaluates to `True` if all characters in the given string are **alphanumeric**, that is, they consist of alphabetic *or* numeric characters:

```
>>> "Hello2World".isalnum()
True
>>> "HelloWorld".isalnum()
True
>>> "2016".isalnum()
True
```

```
>>> "Hello World".isalnum()  # contains whitespace
False
```

`str.isspace`

Evaluates to `True` if the string contains only whitespace characters.

```
>>> "\t\r\n".isspace()
True
>>> " ".isspace()
True
```

Sometimes a string looks “empty” but we don't know whether it's because it contains just whitespace or no character at all

```
>>> "".isspace()
False
```

To cover this case we need an additional test

```
>>> my_str = ''
>>> my_str.isspace()
False
>>> my_str.isspace() or not my_str
True
```

But the shortest way to test if a string is empty or just contains whitespace characters is to use `strip`(with no arguments it removes all leading and trailing whitespace characters)

```
>>> not my_str.strip()
True
```

Section 41.10: String Contains

Python makes it extremely intuitive to check if a string contains a given substring. Just use the `in` operator:

```
>>> "foo" in "foo.baz.bar"
True
```

Note: testing an empty string will always result in `True`:

```
>>> "" in "test"
True
```

Section 41.11: Join a list of strings into one string

A string can be used as a separator to join a list of strings together into a single string using the `join()` method. For example you can create a string where each element in a list is separated by a space.

```
>>> " ".join(["once","upon","a","time"])
"once upon a time"
```

The following example separates the string elements with three hyphens.

```
>>> "---".join(["once", "upon", "a", "time"])
```

```
"once---upon---a---time"
```

Section 41.12: Counting number of times a substring appears in a string

One method is available for counting the number of occurrences of a sub-string in another string, `str.count`.

```
str.count(sub[, start[, end]])
```

`str.count` returns an `int` indicating the number of non-overlapping occurrences of the sub-string `sub` in another string. The optional arguments `start` and `end` indicate the beginning and the end in which the search will take place. By default `start = 0` and `end = len(str)` meaning the whole string will be searched:

```
>>> s = "She sells seashells by the seashore."
>>> s.count("sh")
2
>>> s.count("se")
3
>>> s.count("sea")
2
>>> s.count("seashells")
1
```

By specifying a different value for `start`, `end` we can get a more localized search and count, for example, if `start` is equal to 13 the call to:

```
>>> s.count("sea", start)
1
```

is equivalent to:

```
>>> t = s[start:]
>>> t.count("sea")
1
```

Section 41.13: Case insensitive string comparisons

Comparing string in a case insensitive way seems like something that's trivial, but it's not. This section only considers unicode strings (the default in Python 3). Note that Python 2 may have subtle weaknesses relative to Python 3 - the later's unicode handling is much more complete.

The first thing to note it that case-removing conversions in unicode aren't trivial. There is text for which `text.lower() != text.upper().lower()`, such as `"ß"`:

```
>>> "ß".lower()
'ß'

>>> "ß".upper().lower()
'ss'
```

But let's say you wanted to caselessly compare `"BUSSE"` and `"Buße"`. You probably also want to compare `"BUSSE"` and `"BU□E"` equal - that's the newer capital form. The recommended way is to use `casefold`:

Python 3.x `Version ≥ 3.3`

```
>>> help(str.casefold)
"""
Help on method_descriptor:

casefold(...)
      S.casefold() -> str

     Return a version of S suitable for caseless comparisons.
"""
```

Do not just use `lower`. If `casefold` is not available, doing `.upper().lower()` helps (but only somewhat).

Then you should consider accents. If your font renderer is good, you probably think `"ê" == "ê"` - but it doesn't:

```
>>> "ê" == "ê"
False
```

This is because they are actually

```
>>> import unicodedata

>>> [unicodedata.name(char) for char in "ê"]
['LATIN SMALL LETTER E WITH CIRCUMFLEX']

>>> [unicodedata.name(char) for char in "ê"]
['LATIN SMALL LETTER E', 'COMBINING CIRCUMFLEX ACCENT']
```

The simplest way to deal with this is `unicodedata.normalize`. You probably want to use **NFKD** normalization, but feel free to check the documentation. Then one does

```
>>> unicodedata.normalize("NFKD", "ê") == unicodedata.normalize("NFKD", "ê")
True
```

To finish up, here this is expressed in functions:

```
import unicodedata

def normalize_caseless(text):
    return unicodedata.normalize("NFKD", text.casefold())

def caseless_equal(left, right):
    return normalize_caseless(left) == normalize_caseless(right)
```

Section 41.14: Justify strings

Python provides functions for justifying strings, enabling text padding to make aligning various strings much easier.

Below is an example of `str.ljust` and `str.rjust`:

```
interstates_lengths = {
    5: (1381, 2222),
    19: (63, 102),
    40: (2555, 4112),
    93: (189,305),
}
for road, length in interstates_lengths.items():
    miles,kms = length
```

```
    print('{} -> {} mi. ({} km.)'.format(str(road).rjust(4), str(miles).ljust(4),
str(kms).ljust(4)))
```

```
  40 -> 2555 mi. (4112 km.)
  19 -> 63   mi. (102  km.)
   5 -> 1381 mi. (2222 km.)
  93 -> 189  mi. (305  km.)
```

`ljust` and `rjust` are very similar. Both have a `width` parameter and an optional `fillchar` parameter. Any string created by these functions is at least as long as the `width` parameter that was passed into the function. If the string is longer than `width` alread, it is not truncated. The `fillchar` argument, which defaults to the space character `' '` must be a single character, not a multicharacter string.

The `ljust` function pads the end of the string it is called on with the `fillchar` until it is `width` characters long. The `rjust` function pads the beginning of the string in a similar fashion. Therefore, the `l` and `r` in the names of these functions refer to the side that the original string, *not the `fillchar`*, is positioned in the output string.

Section 41.15: Test the starting and ending characters of a string

In order to test the beginning and ending of a given string in Python, one can use the methods `str.startswith()` and `str.endswith()`.

`str.startswith(prefix[, start[, end]])`

As its name implies, `str.startswith` is used to test whether a given string starts with the given characters in `prefix`.

```
>>> s = "This is a test string"
>>> s.startswith("T")
True
>>> s.startswith("Thi")
True
>>> s.startswith("thi")
False
```

The optional arguments `start` and `end` specify the start and end points from which the testing will start and finish. In the following example, by specifying a start value of `2` our string will be searched from position `2` and afterwards:

```
>>> s.startswith("is", 2)
True
```

This yields `True` since `s[2] == 'i'` and `s[3] == 's'`.

You can also use a `tuple` to check if it starts with any of a set of strings

```
>>> s.startswith(('This', 'That'))
True
>>> s.startswith(('ab', 'bc'))
False
```

`str.endswith(prefix[, start[, end]])`

`str.endswith` is exactly similar to `str.startswith` with the only difference being that it searches for ending characters and not starting characters. For example, to test if a string ends in a full stop, one could write:

```
>>> s = "this ends in a full stop."
>>> s.endswith('.')
True
>>> s.endswith('!')
False
```

as with `startswith` more than one characters can used as the ending sequence:

```
>>> s.endswith('stop.')
True
>>> s.endswith('Stop.')
False
```

You can also use a `tuple` to check if it ends with any of a set of strings

```
>>> s.endswith(('.', 'something'))
True
>>> s.endswith(('ab', 'bc'))
False
```

Section 41.16: Conversion between str or bytes data and unicode characters

The contents of files and network messages may represent encoded characters. They often need to be converted to unicode for proper display.

In Python 2, you may need to convert str data to Unicode characters. The default (`''`, `""`, etc.) is an ASCII string, with any values outside of ASCII range displayed as escaped values. Unicode strings are `u''` (or `u""`, etc.).

Python 2.x `Version ≥ 2.3`

```
# You get "© abc" encoded in UTF-8 from a file, network, or other data source

s = '\xc2\xa9 abc'  # s is a byte array, not a string of characters
                    # Doesn't know the original was UTF-8
                    # Default form of string literals in Python 2
s[0]                # '\xc2' - meaningless byte (without context such as an encoding)
type(s)             # str - even though it's not a useful one w/o having a known encoding

u = s.decode('utf-8')  # u'\xa9 abc'
                       # Now we have a Unicode string, which can be read as UTF-8 and printed
properly
                       # In Python 2, Unicode string literals need a leading u
                       # str.decode converts a string which may contain escaped bytes to a Unicode
string
u[0]                # u'\xa9' - Unicode Character 'COPYRIGHT SIGN' (U+00A9) '©'
type(u)             # unicode

u.encode('utf-8')   # '\xc2\xa9 abc'
                    # unicode.encode produces a string with escaped bytes for non-ASCII characters
```

In Python 3 you may need to convert arrays of bytes (referred to as a 'byte literal') to strings of Unicode characters. The default is now a Unicode string, and bytestring literals must now be entered as `b''`, `b""`, etc. A byte literal will return `True` to `isinstance(some_val, byte)`, assuming `some_val` to be a string that might be encoded as bytes.

Python 3.x `Version ≥ 3.0`

```
# You get from file or network "© abc" encoded in UTF-8
```

```
s = b'\xc2\xa9 abc' # s is a byte array, not characters
                    # In Python 3, the default string literal is Unicode; byte array literals need a
leading b
s[0]                # b'\xc2' - meaningless byte (without context such as an encoding)
type(s)             # bytes - now that byte arrays are explicit, Python can show that.

u = s.decode('utf-8')  # '© abc' on a Unicode terminal
                       # bytes.decode converts a byte array to a string (which will, in Python 3, be
Unicode)
u[0]                # '\u00a9' - Unicode Character 'COPYRIGHT SIGN' (U+00A9) '©'
type(u)             # str
                    # The default string literal in Python 3 is UTF-8 Unicode

u.encode('utf-8')   # b'\xc2\xa9 abc'
                    # str.encode produces a byte array, showing ASCII-range bytes as unescaped
characters.
```

Chapter 42: Using loops within functions

In Python function will be returned as soon as execution hits "return" statement.

Section 42.1: Return statement inside loop in a function

In this example, function will return as soon as value var has 1

```
def func(params):
    for value in params:
        print ('Got value {}'.format(value))

        if value == 1:
            # Returns from function as soon as value is 1
            print (">>>> Got 1")
            return

        print ("Still looping")

    return "Couldn't find 1"

func([5, 3, 1, 2, 8, 9])
```

output

```
Got value 5
Still looping
Got value 3
Still looping
Got value 1
>>>> Got 1
```

Chapter 43: Importing modules

Section 43.1: Importing a module

Use the `import` statement:

```
>>> import random
>>> print(random.randint(1, 10))
4
```

`import module` will import a module and then allow you to reference its objects -- values, functions and classes, for example -- using the `module.name` syntax. In the above example, the `random` module is imported, which contains the `randint` function. So by importing `random` you can call `randint` with `random.randint`.

You can import a module and assign it to a different name:

```
>>> import random as rn
>>> print(rn.randint(1, 10))
4
```

If your python file `main.py` is in the same folder as `custom.py`. You can import it like this:

```
import custom
```

It is also possible to import a function from a module:

```
>>> from math import sin
>>> sin(1)
0.8414709848078965
```

To import specific functions deeper down into a module, the dot operator may be used **only** on the left side of the `import` keyword:

```
from urllib.request import urlopen
```

In python, we have two ways to call function from top level. One is `import` and another is `from`. We should use `import` when we have a possibility of name collision. Suppose we have `hello.py` file and `world.py` files having same function named `function`. Then `import` statement will work good.

```
from hello import function
from world import function

function() #world's function will be invoked. Not hello's
```

In general `import` will provide you a namespace.

```
import hello
import world

hello.function() # exclusively hello's function will be invoked
world.function() # exclusively world's function will be invoked
```

But if you are sure enough, in your whole project there is no way having same function name you should use `from` statement

Multiple imports can be made on the same line:

```
>>> # Multiple modules
>>> import time, sockets, random
>>> # Multiple functions
>>> from math import sin, cos, tan
>>> # Multiple constants
>>> from math import pi, e

>>> print(pi)
3.141592653589793
>>> print(cos(45))
0.5253219888177297
>>> print(time.time())
1482807222.7240417
```

The keywords and syntax shown above can also be used in combinations:

```
>>> from urllib.request import urlopen as geturl, pathname2url as path2url, getproxies
>>> from math import factorial as fact, gamma, atan as arctan
>>> import random.randint, time, sys

>>> print(time.time())
1482807222.7240417
>>> print(arctan(60))
1.554131203080956
>>> filepath = "/dogs/jumping poodle (december).png"
>>> print(path2url(filepath))
/dogs/jumping%20poodle%20%28december%29.png
```

Section 43.2: The __all__ special variable

Modules can have a special variable named `__all__` to restrict what variables are imported when using `from mymodule import *`.

Given the following module:

```
# mymodule.py

__all__ = ['imported_by_star']

imported_by_star = 42
not_imported_by_star = 21
```

Only `imported_by_star` is imported when using `from mymodule import *`:

```
>>> from mymodule import *
>>> imported_by_star
42
>>> not_imported_by_star
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'not_imported_by_star' is not defined
```

However, `not_imported_by_star` can be imported explicitly:

```
>>> from mymodule import not_imported_by_star
>>> not_imported_by_star
```

```
21
```

Section 43.3: Import modules from an arbitrary filesystem location

If you want to import a module that doesn't already exist as a built-in module in the Python Standard Library nor as a side-package, you can do this by adding the path to the directory where your module is found to `sys.path`. This may be useful where multiple python environments exist on a host.

```
import sys
sys.path.append("/path/to/directory/containing/your/module")
import mymodule
```

It is important that you append the path to the *directory* in which `mymodule` is found, not the path to the module itself.

Section 43.4: Importing all names from a module

```
from module_name import *
```

for example:

```
from math import *
sqrt(2)    # instead of math.sqrt(2)
ceil(2.7)  # instead of math.ceil(2.7)
```

This will import all names defined in the `math` module into the global namespace, other than names that begin with an underscore (which indicates that the writer feels that it is for internal use only).

Warning: If a function with the same name was already defined or imported, it will be **overwritten**. Almost always importing only specific names `from math import sqrt, ceil` is the **recommended way**:

```
def sqrt(num):
    print("I don't know what's the square root of {}.".format(num))

sqrt(4)
# Output: I don't know what's the square root of 4.

from math import *
sqrt(4)
# Output: 2.0
```

Starred imports are only allowed at the module level. Attempts to perform them in class or function definitions result in a `SyntaxError`.

```
def f():
    from math import *
```

and

```
class A:
    from math import *
```

both fail with:

```
SyntaxError: import * only allowed at module level
```

Section 43.5: Programmatic importing

Python 2.x `Version ≥ 2.7`

To import a module through a function call, use the `importlib` module (included in Python starting in version 2.7):

```
import importlib
random = importlib.import_module("random")
```

The `importlib.import_module()` function will also import the submodule of a package directly:

```
collections_abc = importlib.import_module("collections.abc")
```

For older versions of Python, use the `imp` module.

Python 2.x `Version ≤ 2.7`

Use the functions `imp.find_module` and `imp.load_module` to perform a programmatic import.

Taken from standard library documentation

```
import imp, sys
def import_module(name):
    fp, pathname, description = imp.find_module(name)
    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        if fp:
            fp.close()
```

Do **NOT** use `__import__()` to programmatically import modules! There are subtle details involving `sys.modules`, the `fromlist` argument, etc. that are easy to overlook which `importlib.import_module()` handles for you.

Section 43.6: PEP8 rules for Imports

Some recommended PEP8 style guidelines for imports:

1. Imports should be on separate lines:

```
from math import sqrt, ceil      # Not recommended
from math import sqrt            # Recommended
from math import ceil
```

2. Order imports as follows at the top of the module:

   - Standard library imports
   - Related third party imports
   - Local application/library specific imports

3. Wildcard imports should be avoided as it leads to confusion in names in the current namespace. If you do `from module import *`, it can be unclear if a specific name in your code comes from `module` or not. This is

doubly true if you have multiple `from module import *`-type statements.

4. Avoid using relative imports; use explicit imports instead.

Section 43.7: Importing specific names from a module

Instead of importing the complete module you can import only specified names:

```
from random import randint # Syntax "from MODULENAME import NAME1[, NAME2[, ...]]"
print(randint(1, 10))      # Out: 5
```

`from random` is needed, because the python interpreter has to know from which resource it should import a function or class and `import randint` specifies the function or class itself.

Another example below (similar to the one above):

```
from math import pi
print(pi)                  # Out: 3.14159265359
```

The following example will raise an error, because we haven't imported a module:

```
random.randrange(1, 10)    # works only if "import random" has been run before
```

Outputs:

```
NameError: name 'random' is not defined
```

The python interpreter does not understand what you mean with `random`. It needs to be declared by adding `import random` to the example:

```
import random
random.randrange(1, 10)
```

Section 43.8: Importing submodules

```
from module.submodule import function
```

This imports `function` from `module.submodule`.

Section 43.9: Re-importing a module

When using the interactive interpreter, you might want to reload a module. This can be useful if you're editing a module and want to import the newest version, or if you've monkey-patched an element of an existing module and want to revert your changes.

Note that you **can't** just `import` the module again to revert:

```
import math
math.pi = 3
print(math.pi)    # 3
import math
print(math.pi)    # 3
```

This is because the interpreter registers every module you import. And when you try to reimport a module, the interpreter sees it in the register and does nothing. So the hard way to reimport is to use `import` after removing the corresponding item from the register:

```
print(math.pi)    # 3
import sys
if 'math' in sys.modules:  # Is the ``math`` module in the register?
    del sys.modules['math']  # If so, remove it.
import math
print(math.pi)    # 3.141592653589793
```

But there is more a straightforward and simple way.

Python 2

Use the `reload` function:

Python 2.x `Version ≥ 2.3`

```
import math
math.pi = 3
print(math.pi)    # 3
reload(math)
print(math.pi)    # 3.141592653589793
```

Python 3

The `reload` function has moved to `importlib`:

Python 3.x `Version ≥ 3.0`

```
import math
math.pi = 3
print(math.pi)    # 3
from importlib import reload
reload(math)
print(math.pi)    # 3.141592653589793
```

Section 43.10: __import__() function

The `__import__()` function can be used to import modules where the name is only known at runtime

```
if user_input == "os":
    os = __import__("os")

# equivalent to import os
```

This function can also be used to specify the file path to a module

```
mod = __import__(r"C:/path/to/file/anywhere/on/computer/module.py")
```

Chapter 44: Difference between Module and Package

Section 44.1: Modules

A module is a single Python file that can be imported. Using a module looks like this:

`module.py`

```
def hi():
    print("Hello world!")
```

`my_script.py`

```
import module
module.hi()
```

in an interpreter

```
>>> from module import hi
>>> hi()
# Hello world!
```

Section 44.2: Packages

A package is made up of multiple Python files (or modules), and can even include libraries written in C or C++. Instead of being a single file, it is an entire folder structure which might look like this:

Folder `package`

- `__init__.py`
- `dog.py`
- `hi.py`

`__init__.py`

```
from package.dog import woof
from package.hi import hi
```

`dog.py`

```
def woof():
    print("WOOF!!!")
```

`hi.py`

```
def hi():
    print("Hello world!")
```

All Python packages must contain an `__init__.py` file. When you import a package in your script (`import package`), the `__init__.py` script will be run, giving you access to the all of the functions in the package. In this case, it allows you to use the `package.hi` and `package.woof` functions.

Chapter 45: Math Module

Section 45.1: Rounding: round, floor, ceil, trunc

In addition to the built-in `round` function, the `math` module provides the `floor`, `ceil`, and `trunc` functions.

```
x = 1.55
y = -1.55

# round to the nearest integer
round(x)       #  2
round(y)       # -2

# the second argument gives how many decimal places to round to (defaults to 0)
round(x, 1)    #  1.6
round(y, 1)    # -1.6

# math is a module so import it first, then use it.
import math

# get the largest integer less than x
math.floor(x)  #  1
math.floor(y)  # -2

# get the smallest integer greater than x
math.ceil(x)   #  2
math.ceil(y)   # -1

# drop fractional part of x
math.trunc(x)  #  1, equivalent to math.floor for positive numbers
math.trunc(y)  # -1, equivalent to math.ceil for negative numbers
```

Python 2.x `Version ≤ 2.7`

`floor`, `ceil`, `trunc`, and `round` always return a `float`.

```
round(1.3)  # 1.0
```

`round` always breaks ties away from zero.

```
round(0.5)  # 1.0
round(1.5)  # 2.0
```

Python 3.x `Version ≥ 3.0`

`floor`, `ceil`, and `trunc` always return an `Integral` value, while `round` returns an `Integral` value if called with one argument.

```
round(1.3)      # 1
round(1.33, 1)  # 1.3
```

`round` breaks ties towards the nearest even number. This corrects the bias towards larger numbers when performing a large number of calculations.

```
round(0.5)  # 0
round(1.5)  # 2
```

Warning!

As with any floating-point representation, some fractions *cannot be represented exactly*. This can lead to some unexpected rounding behavior.

```
round(2.675, 2)  # 2.67, not 2.68!
```

Warning about the floor, trunc, and integer division of negative numbers

Python (and C++ and Java) round away from zero for negative numbers. Consider:

```
>>> math.floor(-1.7)
-2.0
>>> -5 // 2
-3
```

Section 45.2: Trigonometry

Calculating the length of the hypotenuse

```
math.hypot(2, 4) # Just a shorthand for SquareRoot(2**2 + 4**2)
# Out: 4.47213595499958
```

Converting degrees to/from radians

All `math` functions expect **radians** so you need to convert degrees to radians:

```
math.radians(45)              # Convert 45 degrees to radians
# Out: 0.7853981633974483
```

All results of the inverse trigonometric functions return the result in radians, so you may need to convert it back to degrees:

```
math.degrees(math.asin(1))    # Convert the result of asin to degrees
# Out: 90.0
```

Sine, cosine, tangent and inverse functions

```
# Sine and arc sine
math.sin(math.pi / 2)
# Out: 1.0
math.sin(math.radians(90))   # Sine of 90 degrees
# Out: 1.0

math.asin(1)
# Out: 1.5707963267948966    # "= pi / 2"
math.asin(1) / math.pi
# Out: 0.5

# Cosine and arc cosine:
math.cos(math.pi / 2)
# Out: 6.123233995736766e-17
# Almost zero but not exactly because "pi" is a float with limited precision!

math.acos(1)
# Out: 0.0

# Tangent and arc tangent:
math.tan(math.pi/2)
# Out: 1.633123935319537e+16
# Very large but not exactly "Inf" because "pi" is a float with limited precision
```

Python 3.x `Version ≥ 3.5`

```
math.atan(math.inf)
```

```
# Out: 1.5707963267948966 # This is just "pi / 2"
```

```
math.atan(float('inf'))
# Out: 1.5707963267948966 # This is just "pi / 2"
```

Apart from the `math.atan` there is also a two-argument `math.atan2` function, which computes the correct quadrant and avoids pitfalls of division by zero:

```
math.atan2(1, 2)   # Equivalent to "math.atan(1/2)"
# Out: 0.4636476090008061 # ≈ 26.57 degrees, 1st quadrant

math.atan2(-1, -2) # Not equal to "math.atan(-1/-2)" == "math.atan(1/2)"
# Out: -2.677945044588987 # ≈ -153.43 degrees (or 206.57 degrees), 3rd quadrant

math.atan2(1, 0)   # math.atan(1/0) would raise ZeroDivisionError
# Out: 1.5707963267948966 # This is just "pi / 2"
```

Hyperbolic sine, cosine and tangent

```
# Hyperbolic sine function
math.sinh(math.pi) # = 11.548739357257746
math.asinh(1)      # = 0.8813735870195429

# Hyperbolic cosine function
math.cosh(math.pi) # = 11.591953275521519
math.acosh(1)      # = 0.0

# Hyperbolic tangent function
math.tanh(math.pi) # = 0.99627207622075
math.atanh(0.5)    # = 0.5493061443340549
```

Section 45.3: Pow for faster exponentiation

Using the timeit module from the command line:

```
> python -m timeit 'for x in xrange(50000): b = x**3'
10 loops, best of 3: 51.2 msec per loop
> python -m timeit 'from math import pow' 'for x in xrange(50000): b = pow(x,3)'
100 loops, best of 3: 9.15 msec per loop
```

The built-in `**` operator often comes in handy, but if performance is of the essence, use math.pow. Be sure to note, however, that pow returns floats, even if the arguments are integers:

```
> from math import pow
> pow(5,5)
3125.0
```

Section 45.4: Infinity and NaN ("not a number")

In all versions of Python, we can represent infinity and NaN ("not a number") as follows:

```
pos_inf = float('inf')     # positive infinity
neg_inf = float('-inf')    # negative infinity
not_a_num = float('nan')   # NaN ("not a number")
```

In Python 3.5 and higher, we can also use the defined constants `math.inf` and `math.nan`:

Python 3.x `Version ≥ 3.5`

```
pos_inf = math.inf
neg_inf = -math.inf
not_a_num = math.nan
```

The string representations display as `inf` and `-inf` and `nan`:

```
pos_inf, neg_inf, not_a_num
# Out: (inf, -inf, nan)
```

We can test for either positive or negative infinity with the `isinf` method:

```
math.isinf(pos_inf)
# Out: True

math.isinf(neg_inf)
# Out: True
```

We can test specifically for positive infinity or for negative infinity by direct comparison:

```
pos_inf == float('inf')    # or  == math.inf in Python 3.5+
# Out: True

neg_inf == float('-inf')   # or  == -math.inf in Python 3.5+
# Out: True

neg_inf == pos_inf
# Out: False
```

Python 3.2 and higher also allows checking for finiteness:

Python 3.x `Version ≥ 3.2`

```
math.isfinite(pos_inf)
# Out: False

math.isfinite(0.0)
# Out: True
```

Comparison operators work as expected for positive and negative infinity:

```
import sys

sys.float_info.max
# Out: 1.7976931348623157e+308  (this is system-dependent)

pos_inf > sys.float_info.max
# Out: True

neg_inf < -sys.float_info.max
# Out: True
```

But if an arithmetic expression produces a value larger than the maximum that can be represented as a `float`, it will become infinity:

```
pos_inf == sys.float_info.max * 1.0000001
# Out: True

neg_inf == -sys.float_info.max * 1.0000001
```

```
# Out: True
```

However division by zero does not give a result of infinity (or negative infinity where appropriate), rather it raises a `ZeroDivisionError` exception.

```
try:
    x = 1.0 / 0.0
    print(x)
except ZeroDivisionError:
    print("Division by zero")

# Out: Division by zero
```

Arithmetic operations on infinity just give infinite results, or sometimes NaN:

```
-5.0 * pos_inf == neg_inf
# Out: True

-5.0 * neg_inf == pos_inf
# Out: True

pos_inf * neg_inf == neg_inf
# Out: True

0.0 * pos_inf
# Out: nan

0.0 * neg_inf
# Out: nan

pos_inf / pos_inf
# Out: nan
```

NaN is never equal to anything, not even itself. We can test for it is with the `isnan` method:

```
not_a_num == not_a_num
# Out: False

math.isnan(not_a_num)
Out: True
```

NaN always compares as "not equal", but never less than or greater than:

```
not_a_num != 5.0   # or any random value
# Out: True

not_a_num > 5.0   or   not_a_num < 5.0   or   not_a_num == 5.0
# Out: False
```

Arithmetic operations on NaN always give NaN. This includes multiplication by -1: there is no "negative NaN".

```
5.0 * not_a_num
# Out: nan

float('-nan')
# Out: nan
```

Python 3.x `Version ≥ 3.5`

```
-math.nan
```

```
# Out: nan
```

There is one subtle difference between the old `float` versions of NaN and infinity and the Python 3.5+ `math` library constants:

Python 3.x `Version ≥ 3.5`

```
math.inf is math.inf, math.nan is math.nan
# Out: (True, True)

float('inf') is float('inf'), float('nan') is float('nan')
# Out: (False, False)
```

Section 45.5: Logarithms

`math.log(x)` gives the natural (base `e`) logarithm of `x`.

```
math.log(math.e)  # 1.0
math.log(1)       # 0.0
math.log(100)     # 4.605170185988092
```

`math.log` can lose precision with numbers close to 1, due to the limitations of floating-point numbers. In order to accurately calculate logs close to 1, use `math.log1p`, which evaluates the natural logarithm of 1 plus the argument:

```
math.log(1 + 1e-20)  # 0.0
math.log1p(1e-20)    # 1e-20
```

`math.log10` can be used for logs base 10:

```
math.log10(10)  # 1.0
```

Python 2.x `Version ≥ 2.3.0`

When used with two arguments, `math.log(x, base)` gives the logarithm of `x` in the given `base` (i.e. `log(x) / log(base)`).

```
math.log(100, 10) # 2.0
math.log(27, 3)   # 3.0
math.log(1, 10)   # 0.0
```

Section 45.6: Constants

`math` modules includes two commonly used mathematical constants.

- `math.pi` - The mathematical constant pi
- `math.e` - The mathematical constant *e* (base of natural logarithm)

```
>>> from math import pi, e
>>> pi
3.141592653589793
>>> e
2.718281828459045
>>>
```

Python 3.5 and higher have constants for infinity and NaN ("not a number"). The older syntax of passing a string to `float()` still works.

Python 3.x Version ≥ 3.5

```
math.inf == float('inf')
# Out: True

-math.inf == float('-inf')
# Out: True

# NaN never compares equal to anything, even itself
math.nan == float('nan')
# Out: False
```

Section 45.7: Imaginary Numbers

Imaginary numbers in Python are represented by a "j" or "J" trailing the target number.

```
1j         # Equivalent to the square root of -1.
1j * 1j    # = (-1+0j)
```

Section 45.8: Copying signs

In Python 2.6 and higher, `math.copysign(x, y)` returns `x` with the sign of `y`. The returned value is always a `float`.

Python 2.x Version ≥ 2.6

```
math.copysign(-2, 3)    # 2.0
math.copysign(3, -3)    # -3.0
math.copysign(4, 14.2)  # 4.0
math.copysign(1, -0.0)  # -1.0, on a platform which supports signed zero
```

Section 45.9: Complex numbers and the cmath module

The `cmath` module is similar to the `math` module, but defines functions appropriately for the complex plane.

First of all, complex numbers are a numeric type that is part of the Python language itself rather than being provided by a library class. Thus we don't need to `import cmath` for ordinary arithmetic expressions.

Note that we use `j` (or `J`) and not `i`.

```
z = 1 + 3j
```

We must use `1j` since `j` would be the name of a variable rather than a numeric literal.

```
1j * 1j
Out: (-1+0j)

1j ** 1j
# Out: (0.20787957635076193+0j)     # "i to the i"  ==  math.e ** -(math.pi/2)
```

We have the `real` part and the `imag` (imaginary) part, as well as the complex `conjugate`:

```
# real part and imaginary part are both float type
z.real, z.imag
# Out: (1.0, 3.0)

z.conjugate()
# Out: (1-3j)    # z.conjugate() == z.real - z.imag * 1j
```

The built-in functions `abs` and `complex` are also part of the language itself and don't require any import:

```
abs(1 + 1j)
# Out: 1.4142135623730951     # square root of 2

complex(1)
# Out: (1+0j)

complex(imag=1)
# Out: (1j)

complex(1, 1)
# Out: (1+1j)
```

The `complex` function can take a string, but it can't have spaces:

```
complex('1+1j')
# Out: (1+1j)

complex('1 + 1j')
# Exception: ValueError: complex() arg is a malformed string
```

But for most functions we do need the module, for instance `sqrt`:

```
import cmath

cmath.sqrt(-1)
# Out: 1j
```

Naturally the behavior of `sqrt` is different for complex numbers and real numbers. In non-complex `math` the square root of a negative number raises an exception:

```
import math

math.sqrt(-1)
# Exception: ValueError: math domain error
```

Functions are provided to convert to and from polar coordinates:

```
cmath.polar(1 + 1j)
# Out: (1.4142135623730951, 0.7853981633974483)    # == (sqrt(1 + 1), atan2(1, 1))

abs(1 + 1j), cmath.phase(1 + 1j)
# Out: (1.4142135623730951, 0.7853981633974483)    # same as previous calculation

cmath.rect(math.sqrt(2), math.atan(1))
# Out: (1.0000000000000002+1.0000000000000002j)
```

The mathematical field of complex analysis is beyond the scope of this example, but many functions in the complex plane have a "branch cut", usually along the real axis or the imaginary axis. Most modern platforms support "signed zero" as specified in IEEE 754, which provides continuity of those functions on both sides of the branch cut. The following example is from the Python documentation:

```
cmath.phase(complex(-1.0, 0.0))
# Out: 3.141592653589793

cmath.phase(complex(-1.0, -0.0))
```

```
# Out: -3.141592653589793
```

The `cmath` module also provides many functions with direct counterparts from the `math` module.

In addition to `sqrt`, there are complex versions of `exp`, `log`, `log10`, the trigonometric functions and their inverses (`sin`, `cos`, `tan`, `asin`, `acos`, `atan`), and the hyperbolic functions and their inverses (`sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`). Note however there is no complex counterpart of `math.atan2`, the two-argument form of arctangent.

```
cmath.log(1+1j)
# Out: (0.34657359027997264+0.7853981633974483j)

cmath.exp(1j * cmath.pi)
# Out: (-1+1.2246467991473532e-16j)   # e to the i pi == -1, within rounding error
```

The constants `pi` and `e` are provided. Note these are `float` and not `complex`.

```
type(cmath.pi)
# Out: <class 'float'>
```

The `cmath` module also provides complex versions of `isinf`, and (for Python 3.2+) `isfinite`. See "Infinity and NaN". A complex number is considered infinite if either its real part or its imaginary part is infinite.

```
cmath.isinf(complex(float('inf'), 0.0))
# Out: True
```

Likewise, the `cmath` module provides a complex version of `isnan`. See "Infinity and NaN". A complex number is considered "not a number" if either its real part or its imaginary part is "not a number".

```
cmath.isnan(0.0, float('nan'))
# Out: True
```

Note there is no `cmath` counterpart of the `math.inf` and `math.nan` constants (from Python 3.5 and higher)

Python 3.x `Version ≥ 3.5`

```
cmath.isinf(complex(0.0, math.inf))
# Out: True

cmath.isnan(complex(math.nan, 0.0))
# Out: True

cmath.inf
# Exception: AttributeError: module 'cmath' has no attribute 'inf'
```

In Python 3.5 and higher, there is an `isclose` method in both `cmath` and `math` modules.

Python 3.x `Version ≥ 3.5`

```
z = cmath.rect(*cmath.polar(1+1j))

z
# Out: (1.0000000000000002+1.0000000000000002j)

cmath.isclose(z, 1+1j)
# True
```

Chapter 46: Complex math

Section 46.1: Advanced complex arithmetic

The module `cmath` includes additional functions to use complex numbers.

```
import cmath
```

This module can calculate the phase of a complex number, in radians:

```
z = 2+3j # A complex number
cmath.phase(z) # 0.982793723247329
```

It allows the conversion between the cartesian (rectangular) and polar representations of complex numbers:

```
cmath.polar(z) # (3.605551275463989, 0.982793723247329)
cmath.rect(2, cmath.pi/2) # (0+2j)
```

The module contains the complex version of

- Exponential and logarithmic functions (as usual, `log` is the natural logarithm and `log10` the decimal logarithm):

```
cmath.exp(z) # (-7.315110094901103+1.0427436562359045j)
cmath.log(z) # (1.2824746787307684+0.982793723247329j)
cmath.log10(-100) # (2+1.3643763538418412j)
```

- Square roots:

```
cmath.sqrt(z) # (1.6741492280355401+0.8959774761298381j)
```

- Trigonometric functions and their inverses:

```
cmath.sin(z)  # (9.15449914691143-4.168906959966565j)
cmath.cos(z)  # (-4.189625690968807-9.109227893755337j)
cmath.tan(z)  # (-0.003764025641504249+1.00323862735361j)
cmath.asin(z) # (0.5706527843210994+1.9833870299165355j)
cmath.acos(z) # (1.0001435424737972-1.9833870299165355j)
cmath.atan(z) # (1.409921049596575+0.22907268296853878j)
cmath.sin(z)**2 + cmath.cos(z)**2 # (1+0j)
```

- Hyperbolic functions and their inverses:

```
cmath.sinh(z)  # (-3.59056458998578+0.5309210862485197j)
cmath.cosh(z)  # (-3.7245455049153224+0.5118225699873846j)
cmath.tanh(z)  # (0.965385879022133-0.009884375038322495j)
cmath.asinh(z) # (0.5706527843210994+1.9833870299165355j)
cmath.acosh(z) # (1.9833870299165355+1.0001435424737972j)
cmath.atanh(z) # (0.1469466662255297+1.3389725222944935j)
cmath.cosh(z)**2 - cmath.sin(z)**2  # (1+0j)
cmath.cosh((0+1j)*z) - cmath.cos(z) # 0j
```

Section 46.2: Basic complex arithmetic

Python has built-in support for complex arithmetic. The imaginary unit is denoted by `j`:

```
z = 2+3j # A complex number
w = 1-7j # Another complex number
```

Complex numbers can be summed, subtracted, multiplied, divided and exponentiated:

```
z + w # (3-4j)
z - w # (1+10j)
z * w # (23-11j)
z / w # (-0.38+0.34j)
z**3  # (-46+9j)
```

Python can also extract the real and imaginary parts of complex numbers, and calculate their absolute value and conjugate:

```
z.real # 2.0
z.imag # 3.0
abs(z) # 3.605551275463989
z.conjugate() # (2-3j)
```

Chapter 47: Collections module

The built-in `collections` package provides several specialized, flexible collection types that are both high-performance and provide alternatives to the general collection types of `dict`, `list`, `tuple` and `set`. The module also defines abstract base classes describing different types of collection functionality (such as `MutableSet` and `ItemsView`).

Section 47.1: collections.Counter

Counter is a dict sub class that allows you to easily count objects. It has utility methods for working with the frequencies of the objects that you are counting.

```
import collections
counts = collections.Counter([1,2,3])
```

the above code creates an object, counts, which has the frequencies of all the elements passed to the constructor. This example has the value `Counter({1: 1, 2: 1, 3: 1})`

Constructor examples

Letter Counter

```
>>> collections.Counter('Happy Birthday')
Counter({'a': 2, 'p': 2, 'y': 2, 'i': 1, 'r': 1, 'B': 1, ' ': 1, 'H': 1, 'd': 1, 'h': 1, 't': 1})
```

Word Counter

```
>>> collections.Counter('I am Sam Sam I am That Sam-I-am That Sam-I-am! I do not like that Sam-I-am'.split())
Counter({'I': 3, 'Sam': 2, 'Sam-I-am': 2, 'That': 2, 'am': 2, 'do': 1, 'Sam-I-am!': 1, 'that': 1, 'not': 1, 'like': 1})
```

Recipes

```
>>> c = collections.Counter({'a': 4, 'b': 2, 'c': -2, 'd': 0})
```

Get count of individual element

```
>>> c['a']
4
```

Set count of individual element

```
>>> c['c'] = -3
>>> c
Counter({'a': 4, 'b': 2, 'd': 0, 'c': -3})
```

Get total number of elements in counter (4 + 2 + 0 - 3)

```
>>> sum(c.itervalues())  # negative numbers are counted!
3
```

Get elements (only those with positive counter are kept)

```
>>> list(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

Remove keys with 0 or negative value

```
>>> c - collections.Counter()
Counter({'a': 4, 'b': 2})
```

Remove everything

```
>>> c.clear()
>>> c
Counter()
```

Add remove individual elements

```
>>> c.update({'a': 3, 'b':3})
>>> c.update({'a': 2, 'c':2})  # adds to existing, sets if they don't exist
>>> c
Counter({'a': 5, 'b': 3, 'c': 2})
>>> c.subtract({'a': 3, 'b': 3, 'c': 3})  # subtracts (negative values are allowed)
>>> c
Counter({'a': 2, 'b': 0, 'c': -1})
```

Section 47.2: collections.OrderedDict

The order of keys in Python dictionaries is arbitrary: they are not governed by the order in which you add them.

For example:

```
>>> d = {'foo': 5, 'bar': 6}
>>> print(d)
{'foo': 5, 'bar': 6}
>>> d['baz'] = 7
>>> print(a)
{'baz': 7, 'foo': 5, 'bar': 6}
>>> d['foobar'] = 8
>>> print(a)
{'baz': 7, 'foo': 5, 'bar': 6, 'foobar': 8}
```

(The arbitrary ordering implied above means that you may get different results with the above code to that shown here.)

The order in which the keys appear is the order which they would be iterated over, e.g. using a `for` loop.

The `collections.OrderedDict` class provides dictionary objects that retain the order of keys. `OrderedDicts` can be created as shown below with a series of ordered items (here, a list of tuple key-value pairs):

```
>>> from collections import OrderedDict
>>> d = OrderedDict([('foo', 5), ('bar', 6)])
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6)])
>>> d['baz'] = 7
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6), ('baz', 7)])
>>> d['foobar'] = 8
```

```
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

Or we can create an empty `OrderedDict` and then add items:

```
>>> o = OrderedDict()
>>> o['key1'] = "value1"
>>> o['key2'] = "value2"
>>> print(o)
OrderedDict([('key1', 'value1'), ('key2', 'value2')])
```

Iterating through an `OrderedDict` allows key access in the order they were added.

What happens if we assign a new value to an existing key?

```
>>> d['foo'] = 4
>>> print(d)
OrderedDict([('foo', 4), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

The key retains its original place in the `OrderedDict`.

Section 47.3: collections.defaultdict

collections.defaultdict(default_factory) returns a subclass of `dict` that has a default value for missing keys. The argument should be a function that returns the default value when called with no arguments. If there is nothing passed, it defaults to `None`.

```
>>> state_capitals = collections.defaultdict(str)
>>> state_capitals
defaultdict(<class 'str'>, {})
```

returns a reference to a defaultdict that will create a string object with its default_factory method.

A typical usage of `defaultdict` is to use one of the builtin types such as `str`, `int`, `list` or `dict` as the default_factory, since these return empty types when called with no arguments:

```
>>> str()
''
>>> int()
0
>>> list
[]
```

Calling the defaultdict with a key that does not exist does not produce an error as it would in a normal dictionary.

```
>>> state_capitals['Alaska']
''
>>> state_capitals
defaultdict(<class 'str'>, {'Alaska': ''})
```

Another example with `int`:

```
>>> fruit_counts = defaultdict(int)
>>> fruit_counts['apple'] += 2  # No errors should occur
>>> fruit_counts
default_dict(int, {'apple': 2})
>>> fruit_counts['banana']  # No errors should occur
```

```
0
>>> fruit_counts  # A new key is created
default_dict(int, {'apple': 2, 'banana': 0})
```

Normal dictionary methods work with the default dictionary

```
>>> state_capitals['Alabama'] = 'Montgomery'
>>> state_capitals
defaultdict(<class 'str'>, {'Alabama': 'Montgomery', 'Alaska': ''})
```

Using `list` as the default_factory will create a list for each new key.

```
>>> s = [('NC', 'Raleigh'), ('VA', 'Richmond'), ('WA', 'Seattle'), ('NC', 'Asheville')]
>>> dd = collections.defaultdict(list)
>>> for k, v in s:
...     dd[k].append(v)
>>> dd
defaultdict(<class 'list'>,
    {'VA': ['Richmond'],
     'NC': ['Raleigh', 'Asheville'],
     'WA': ['Seattle']})
```

Section 47.4: collections.namedtuple

Define a new type `Person` using `namedtuple` like this:

```
Person = namedtuple('Person', ['age', 'height', 'name'])
```

The second argument is the list of attributes that the tuple will have. You can list these attributes also as either space or comma separated string:

```
Person = namedtuple('Person', 'age, height, name')
```

or

```
Person = namedtuple('Person', 'age height name')
```

Once defined, a named tuple can be instantiated by calling the object with the necessary parameters, e.g.:

```
dave = Person(30, 178, 'Dave')
```

Named arguments can also be used:

```
jack = Person(age=30, height=178, name='Jack S.')
```

Now you can access the attributes of the namedtuple:

```
print(jack.age)  # 30
print(jack.name)  # 'Jack S.'
```

The first argument to the namedtuple constructor (in our example `'Person'`) is the `typename`. It is typical to use the same word for the constructor and the typename, but they can be different:

```
Human = namedtuple('Person',  'age, height, name')
dave = Human(30, 178, 'Dave')
```

```
print(dave)  # yields: Person(age=30, height=178, name='Dave')
```

Section 47.5: collections.deque

Returns a new `deque` object initialized left-to-right (using append()) with data from iterable. If iterable is not specified, the new `deque` is empty.

Deques are a generalization of stacks and queues (the name is pronounced “deck” and is short for “double-ended queue”). Deques support thread-safe, memory efficient appends and pops from either side of the `deque` with approximately the same O(1) performance in either direction.

Though list objects support similar operations, they are optimized for fast fixed-length operations and incur O(n) memory movement costs for pop(0) and insert(0, v) operations which change both the size and position of the underlying data representation.

New in version 2.4.

If `maxlen` is not specified or is `None`, deques may grow to an arbitrary length. Otherwise, the `deque` is bounded to the specified maximum length. Once a bounded length `deque` is full, when new items are added, a corresponding number of items are discarded from the opposite end. Bounded length deques provide functionality similar to the tail filter in Unix. They are also useful for tracking transactions and other pools of data where only the most recent activity is of interest.

Changed in version 2.6: Added maxlen parameter.

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with three items
>>> for elem in d:                   # iterate over the deque's elements
...     print elem.upper()
G
H
I

>>> d.append('j')                    # add a new entry to the right side
>>> d.appendleft('f')                # add a new entry to the left side
>>> d                                # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # return and remove the rightmost item
'j'
>>> d.popleft()                      # return and remove the leftmost item
'f'
>>> list(d)                          # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                             # peek at leftmost item
'g'
>>> d[-1]                            # peek at rightmost item
'i'

>>> list(reversed(d))                # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                         # search the deque
True
>>> d.extend('jkl')                  # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                      # right rotation
>>> d
```

```
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                    # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))              # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                       # empty the deque
>>> d.pop()                         # cannot pop from an empty deque
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')             # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

Source: https://docs.python.org/2/library/collections.html

Section 47.6: collections.ChainMap

`ChainMap` is new in **version 3.3**

Returns a new `ChainMap` object given a number of `maps`. This object groups multiple dicts or other mappings together to create a single, updateable view.

`ChainMaps` are useful managing nested contexts and overlays. An example in the python world is found in the implementation of the `Context` class in Django's template engine. It is useful for quickly linking a number of mappings so that the result can be treated as a single unit. It is often much faster than creating a new dictionary and running multiple `update()` calls.

Anytime one has a chain of lookup values there can be a case for `ChainMap`. An example includes having both user specified values and a dictionary of default values. Another example is the `POST` and `GET` parameter maps found in web use, e.g. Django or Flask. Through the use of `ChainMap` one returns a combined view of two distinct dictionaries.

The `maps` parameter list is ordered from first-searched to last-searched. Lookups search the underlying mappings successively until a key is found. In contrast, writes, updates, and deletions only operate on the first mapping.

```
import collections

# define two dictionaries with at least some keys overlapping.
dict1 = {'apple': 1, 'banana': 2}
dict2 = {'coconut': 1, 'date': 1, 'apple': 3}

# create two ChainMaps with different ordering of those dicts.
combined_dict = collections.ChainMap(dict1, dict2)
reverse_ordered_dict = collections.ChainMap(dict2, dict1)
```

Note the impact of order on which value is found first in the subsequent lookup

```
for k, v in combined_dict.items():
    print(k, v)

date 1
apple 1
banana 2
```

```
coconut 1

for k, v in reverse_ordered_dict.items():
    print(k, v)

date 1
apple 3
banana 2
coconut 1
```

Chapter 48: Operator module

Section 48.1: Itemgetter

Grouping the key-value pairs of a dictionary by the value with `itemgetter`:

```
from itertools import groupby
from operator import itemgetter
adict = {'a': 1, 'b': 5, 'c': 1}

dict((i, dict(v)) for i, v in groupby(adict.items(), itemgetter(1)))
# Output: {1: {'a': 1, 'c': 1}, 5: {'b': 5}}
```

which is equivalent (but faster) to a `lambda` function like this:

```
dict((i, dict(v)) for i, v in groupby(adict.items(), lambda x: x[1]))
```

Or sorting a list of tuples by the second element first the first element as secondary:

```
alist_of_tuples = [(5,2), (1,3), (2,2)]
sorted(alist_of_tuples, key=itemgetter(1,0))
# Output: [(2, 2), (5, 2), (1, 3)]
```

Section 48.2: Operators as alternative to an infix operator

For every infix operator, e.g. `+` there is an `operator`-function (`operator.add` for `+`):

```
1 + 1
# Output: 2
from operator import add
add(1, 1)
# Output: 2
```

even though the main documentation states that for the arithmetic operators only numerical input is allowed it *is* possible:

```
from operator import mul
mul('a', 10)
# Output: 'aaaaaaaaaa'
mul([3], 3)
# Output: [3, 3, 3]
```

See also: mapping from operation to operator function in the official Python documentation.

Section 48.3: Methodcaller

Instead of this `lambda`-function that calls the method explicitly:

```
alist = ['wolf', 'sheep', 'duck']
list(filter(lambda x: x.startswith('d'), alist))     # Keep only elements that start with 'd'
# Output: ['duck']
```

one could use a operator-function that does the same:

```
from operator import methodcaller
```

```
list(filter(methodcaller('startswith', 'd'), alist)) # Does the same but is faster.
# Output: ['duck']
```

Chapter 49: JSON Module

Section 49.1: Storing data in a file

The following snippet encodes the data stored in `d` into JSON and stores it in a file (replace `filename` with the actual name of the file).

```
import json

d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}

with open(filename, 'w') as f:
    json.dump(d, f)
```

Section 49.2: Retrieving data from a file

The following snippet opens a JSON encoded file (replace `filename` with the actual name of the file) and returns the object that is stored in the file.

```
import json

with open(filename, 'r') as f:
    d = json.load(f)
```

Section 49.3: Formatting JSON output

Let's say we have the following data:

```
>>> data = {"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

Just dumping this as JSON does not do anything special here:

```
>>> print(json.dumps(data))
{"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

Setting indentation to get prettier output

If we want pretty printing, we can set an `indent` size:

```
>>> print(json.dumps(data, indent=2))
{
  "cats": [
    {
      "name": "Tubbs",
      "color": "white"
    },
    {
      "name": "Pepper",
      "color": "black"
    }
  ]
}
```