Contents

[1. Python 2.7 vs 3x: 3](#_Toc491815122)

[2. Some basics: 3](#_Toc491815123)

[2.1. Operators: 3](#_Toc491815124)

[2.2. Truth value testing: 4](#_Toc491815125)

[2.3. Compound statements: 4](#_Toc491815126)

[2.3.1 if statement 4](#_Toc491815127)

[2.3.2 while statement: 5](#_Toc491815128)

[2.3.3 for statement: 5](#_Toc491815129)

[2.3.4 try statement: 6](#_Toc491815130)

[2.3.5 pass statement: 6](#_Toc491815131)

[2.3.6 break and continue statement: 7](#_Toc491815132)

[2.3.7 with statement: 7](#_Toc491815133)

[2.4. Looping techniques: 9](#_Toc491815134)

[2.5. Text-mode interactivity: 10](#_Toc491815135)

[3. Data types: 10](#_Toc491815136)

[3.1. List: 10](#_Toc491815137)

[3.2. Tuples: 11](#_Toc491815138)

[3.3. Set: 11](#_Toc491815139)

[3.4. Dictionary: 12](#_Toc491815140)

[3.5. Range: 12](#_Toc491815141)

[3.6. String: 13](#_Toc491815142)

[4. Module: 13](#_Toc491815143)

[4.1. Standard Modules 14](#_Toc491815144)

[4.2. Buit-in Functions: 15](#_Toc491815145)

[4.3. The dir() Function 15](#_Toc491815146)

[5. Packages 17](#_Toc491815147)

[5.1. Importing \* From a Package 18](#_Toc491815148)

[6. Classes 19](#_Toc491815149)

[6.1. Python Scopes and Namespaces 19](#_Toc491815150)

[6.1.1 Scopes and Namespaces Example 20](#_Toc491815151)

[6.2. Class Definition Syntax 20](#_Toc491815152)

[6.3. Class Objects 21](#_Toc491815153)

[6.4. Instance Objects 22](#_Toc491815154)

[6.5. Method Objects 22](#_Toc491815155)

[6.6. Class and Instance Variables 22](#_Toc491815156)

[6.7. Inheritance 24](#_Toc491815157)

[6.7.1 Multiple Inheritance 24](#_Toc491815158)

[6.8. Private Variables 25](#_Toc491815159)

[6.9. Struct like objects: 25](#_Toc491815160)

[7. Python package manager 26](#_Toc491815161)

[7.1. pip 26](#_Toc491815162)

[8. Python Basic Best Practices: 26](#_Toc491815163)

[8.1. PEP 8 and writing readable code: spaces and tabs, standard code layout and Naming convensions 26](#_Toc491815164)

[8.2. Using Version Control 26](#_Toc491815165)

[8.3. Using venv to create Stable and Isolated 26](#_Toc491815166)

[8.4. Docstrings: 27](#_Toc491815167)

[8.5. Doctest 27](#_Toc491815168)

[9. Creating command-line utility 27](#_Toc491815169)

[9.1. Making a Package executable via python –m 27](#_Toc491815170)

[9.2. Handling command-line arguments with argparse 27](#_Toc491815171)

[9.3. Text-mode interactivity: 28](#_Toc491815172)

[10. Internet Data Handling 28](#_Toc491815173)

[10.1. json – JSON encoder and decoder 28](#_Toc491815174)

[11. Some seriously good stuff: 28](#_Toc491815175)

[12. Something else 29](#_Toc491815176)

1. Python 2.7 vs 3x:

**Python 2.x is legacy, Python 3.x is the present and future of the language**

Python 3.0 was released in 2008. The final 2.x version 2.7 release came out in mid-2010, with a statement of extended support for this end-of-life release. The 2.x branch will see no new major releases after that. 3.x is under active development and has already seen over five years of stable releases, including version 3.3 in 2012, 3.4 in 2014, 3.5 in 2015, and 3.6 in 2016. This means that all recent standard library improvements, for example, are only available by default in Python 3.x.

Guido van Rossum (the original creator of the Python language) decided to clean up Python 2.x properly, with less regard for backwards compatibility than is the case for new releases in the 2.x range. The most drastic improvement is the better Unicode support (with all text strings being Unicode by default) as well as saner bytes/Unicode separation.

**Besides, several aspects of the core language (such as print and exec being statements, integers using floor division) have been adjusted to be easier for newcomers to learn and to be more consistent with the rest of the language, and old cruft has been removed (for example, all classes are now new-style, "range()" returns a memory efficient iterable, not a list as in 2.x)**.

The [What's New in Python 3.0](http://docs.python.org/py3k/whatsnew/3.0.html)document provides a good overview of the major language changes and likely sources of incompatibility with existing Python 2.x code. Nick Coghlan (one of the CPython core developers) has also created a [relatively extensive FAQ](http://python-notes.curiousefficiency.org/en/latest/python3/questions_and_answers.html)regarding the transition.

A non-exhaustive list of features which are only available in 3.x releases and won't be backported to the 2.x series:

* strings are Unicode by default
* clean Unicode/bytes separation
* exception chaining
* function annotations
* syntax for keyword-only arguments
* extended tuple unpacking
* non-local variable declarations

There's also the [3to2](https://wiki.python.org/moin/3to2) tool, which aims to convert 3.x code back to 2.x code.

1. Some basics:
   1. Operators:

|  |  |
| --- | --- |
| Operator | Explanation |
| \* | Times operator. If applied to a different datatype other than a number, such as String or List, it repeats the datatype value. For example. Print(‘hello’ \* 3) will result in hellohellohello. Or Print([‘a’,’b’,’c’] \* 2) will result in [‘a’, ’b’, ‘c’, ‘a’, ‘b’, ‘c’] |
| \*\* | exponent operator. Ex: 3 \*\* 3 will result in 27 |
| // | floor division, divides and rounds down. Ex: 16 // 5.0 will result in 3.0 |
| % | modulus operator, results in remainder. Ex: 13 % 5 will result in 3 |
| Casting | str(4) will cast the numeric 4 to string 4 |
| <> | Same as !=. ex: if 10 <> 11 print (“true”) will result in true. |

* 1. Truth value testing:

Any object can be tested for truth value, for use in an [if](https://docs.python.org/3/reference/compound_stmts.html#if) or [while](https://docs.python.org/3/reference/compound_stmts.html#while) condition or as operand of the Boolean operations below. The following values are considered false:

* None
* False
* zero of any numeric type, for example, 0, 0.0, 0j.
* any empty sequence, for example, '', (), [].
* any empty mapping, for example, {}.
* instances of user-defined classes, if the class defines a [\_\_bool\_\_()](https://docs.python.org/3/reference/datamodel.html#object.__bool__) or [\_\_len\_\_()](https://docs.python.org/3/reference/datamodel.html#object.__len__) method, when that method returns the integer zero or [bool](https://docs.python.org/3/library/functions.html" \l "bool" \o "bool) value False. [[1]](https://docs.python.org/3/library/stdtypes.html#id11)

All other values are considered true — so objects of many types are always true.

Operations and built-in functions that have a Boolean result always return 0 or False for false and 1 or True for true, unless otherwise stated.

* 1. Compound statements:
     1. if statement

|  |  |
| --- | --- |
| Syntax | Example |
| if <condition> :  stmt  elif<condition>:  stmt  else :  stmt | if(2==1):  print("if executed")  elif(2==3):  print("elif exucuted")  else:  print("else exucuted") |
| if <condition> and <condition> :  stmt | if(2==1) and (3==3) :  print("if executed")  else:  print("else executed") |
| if <condition> and not <condition> : stmt | if(2!=1) and not (3!=3) :  print("if executed")  else:  print("else executed") |
| if<condition> or <condition> : stmt | if(2==1) or (3==3) :  print("if executed")  else:  print("else executed") |

* + 1. while statement:

|  |  |
| --- | --- |
| Syntax | Example |
| while <condition> :  stmt  [continue]  [break]  [else : stmt} | count = 0  target = 10  while (count < target):  count = count + 1  if(count==5):  print("halfway there!")  continue  print('The count is:', count)  if(count==9):  print('target achieved')  break  else:  print('count is out of bound') |

* + 1. for statement:

|  |  |
| --- | --- |
| Syntax | Example |
| for <target\_list> in <expression\_list> : |  |

Some examples:

for element in [1, 2, 3]:

print(element)

for element in (1, 2, 3):

print(element)

for key in {'one':1, 'two':2}:

print(key)

for char in "123":

print(char)

for line in open("myfile.txt"):

print(line, end='')

Behind the scenes, the [for](https://docs.python.org/3.6/reference/compound_stmts.html#for) statement calls [iter()](https://docs.python.org/3.6/library/functions.html" \l "iter" \o "iter) on the container object. The function returns an iterator object that defines the method [\_\_next\_\_()](https://docs.python.org/3.6/library/stdtypes.html#iterator.__next__) which accesses elements in the container one at a time. When there are no more elements, [\_\_next\_\_()](https://docs.python.org/3.6/library/stdtypes.html#iterator.__next__) raises a [StopIteration](https://docs.python.org/3.6/library/exceptions.html" \l "StopIteration" \o "StopIteration) exception which tells the [for](https://docs.python.org/3.6/reference/compound_stmts.html#for) loop to terminate. You can call the [\_\_next\_\_()](https://docs.python.org/3.6/library/stdtypes.html#iterator.__next__) method using the [next()](https://docs.python.org/3.6/library/functions.html#next) built-in function.

* + 1. try statement:

The [try](https://docs.python.org/3/reference/compound_stmts.html#try) statement specifies exception handlers and/or cleanup code for a group of statements:

try\_stmt ::= try1\_stmt | try2\_stmt

try1\_stmt ::= "try" ":" [suite](https://docs.python.org/3/reference/compound_stmts.html#grammar-token-suite)

("except" [[expression](https://docs.python.org/3/reference/expressions.html#grammar-token-expression) ["as" [identifier](https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-identifier)]] ":" [suite](https://docs.python.org/3/reference/compound_stmts.html#grammar-token-suite))+

["else" ":" [suite](https://docs.python.org/3/reference/compound_stmts.html#grammar-token-suite)]

["finally" ":" [suite](https://docs.python.org/3/reference/compound_stmts.html#grammar-token-suite)]

try2\_stmt ::= "try" ":" [suite](https://docs.python.org/3/reference/compound_stmts.html#grammar-token-suite)

"finally" ":" [suite](https://docs.python.org/3/reference/compound_stmts.html#grammar-token-suite)

The [except](https://docs.python.org/3/reference/compound_stmts.html#except) clause(s) specify one or more exception handlers. When no exception occurs in the [try](https://docs.python.org/3/reference/compound_stmts.html#try) clause, no exception handler is executed. When an exception occurs in the [try](https://docs.python.org/3/reference/compound_stmts.html#try) suite, a search for an exception handler is started. This search inspects the except clauses in turn until one is found that matches the exception. An expression-less except clause, if present, must be last; it matches any exception.

|  |  |
| --- | --- |
| Syntax | Example |
| "try" ":" [suite](https://docs.python.org/3/reference/compound_stmts.html#grammar-token-suite)  ("except" [[expression](https://docs.python.org/3/reference/expressions.html#grammar-token-expression) ["as" [identifier](https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-identifier)]] ":" [suite](https://docs.python.org/3/reference/compound_stmts.html#grammar-token-suite))+  ["else" ":" [suite](https://docs.python.org/3/reference/compound_stmts.html#grammar-token-suite)]  ["finally" ":" [suite](https://docs.python.org/3/reference/compound_stmts.html#grammar-token-suite)] | try:  ...  except SomeException:  tb = sys.exc\_info()[2]  raise OtherException(...).with\_traceback(tb) |

**List of Exceptions:**

<https://docs.python.org/3/library/exceptions.html>

* + 1. pass statement:

The [pass](https://docs.python.org/3.6/reference/simple_stmts.html#pass) statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

>>>

>>> while True:

... pass # Busy-wait for keyboard interrupt (Ctrl+C)

...

This is commonly used for creating minimal classes:

>>>

>>> class MyEmptyClass:

... pass

...

Another place [pass](https://docs.python.org/3.6/reference/simple_stmts.html#pass) can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The [pass](https://docs.python.org/3.6/reference/simple_stmts.html#pass) is silently ignored:

>>>

>>> def initlog(\*args):

... pass # Remember to implement this!

...

* + 1. break and continue statement:

The break statement, like in C, breaks out of the smallest enclosing [for](https://docs.python.org/3.6/reference/compound_stmts.html#for) or [while](https://docs.python.org/3.6/reference/compound_stmts.html#while) loop.

Loop statements may have an else clause; it is executed when the loop terminates through exhaustion of the list (with [for](https://docs.python.org/3.6/reference/compound_stmts.html#for)) or when the condition becomes false (with [while](https://docs.python.org/3.6/reference/compound_stmts.html#while)), but not when the loop is terminated by a [break](https://docs.python.org/3.6/reference/simple_stmts.html#break) statement. This is exemplified by the following loop, which searches for prime numbers:

>>>

>>> for n in range(2, 10):

... for x in range(2, n):

... if n % x == 0:

... print(n, 'equals', x, '\*', n//x)

... break

... else:

... # loop fell through without finding a factor

... print(n, 'is a prime number')

...

2 is a prime number

3 is a prime number

4 equals 2 \* 2

5 is a prime number

6 equals 2 \* 3

7 is a prime number

8 equals 2 \* 4

9 equals 3 \* 3

The continue statement, also borrowed from C, continues with the next iteration of the loop:

>>>

>>> for num in range(2, 10):

... if num % 2 == 0:

... print("Found an even number", num)

... continue

... print("Found a number", num)

Found an even number 2

Found a number 3

Found an even number 4

Found a number 5

Found an even number 6

Found a number 7

Found an even number 8

Found a number 9

* + 1. with statement:

The ‘[with](https://docs.python.org/3/reference/compound_stmts.html#with)’ statement clarifies code that previously would use try...finally blocks to ensure that clean-up code is executed. The ‘[with](https://docs.python.org/3/reference/compound_stmts.html#with)’ statement is a control-flow structure whose basic structure is:

with expression [as variable]:

with-block

The expression is evaluated, and it should result in an object that supports the context management protocol (that is, has \_\_enter\_\_() and \_\_exit\_\_() methods).

The object’s \_\_enter\_\_() is called before with-block is executed and therefore can run set-up code.

After execution of the with-block is finished, the object’s \_\_exit\_\_() method is called, even if the block raised an exception, and can therefore run clean-up code.

Some standard Python objects now support the context management protocol and can be used with the ‘[with](https://docs.python.org/3/reference/compound_stmts.html#with)’ statement. File objects are one example:

with open('/etc/passwd', 'r') as f:

for line in f:

print line

... more processing code ...

After this statement has executed, the file object in f will have been automatically closed, even if the [for](https://docs.python.org/3/reference/compound_stmts.html#for) loop raised an exception part-way through the block.

**Note:** In this case, f  is the same object created by open(), because file.\_\_enter\_\_() returns self.

The [with](https://docs.python.org/3/reference/compound_stmts.html#with) statement is used to wrap the execution of a block with methods defined by a context manager (see section [With Statement Context Managers](https://docs.python.org/3/reference/datamodel.html#context-managers)). This allows common [try](https://docs.python.org/3/reference/compound_stmts.html#try)…[except](https://docs.python.org/3/reference/compound_stmts.html#except)…[finally](https://docs.python.org/3/reference/compound_stmts.html#finally) usage patterns to be encapsulated for convenient reuse.

The execution of the with statement with one “item” proceeds as follows:

1. The context expression (the expression given in the with\_item) is evaluated to obtain a context manager.
2. The context manager’s \_\_exit\_\_() is loaded for later use.
3. The context manager’s \_\_enter\_\_() method is invoked.
4. If a target was included in the with statement, the return value from \_\_enter\_\_() is assigned to it.

**Note** The with statement guarantees that if the \_\_enter\_\_() method returns without an error, then \_\_exit\_\_() will always be called. Thus, if an error occurs during the assignment to the target list, it will be treated the same as an error occurring within the suite would be. See step 6 below.

1. The suite is executed.
2. The context manager’s \_\_exit\_\_() method is invoked. If an exception caused the suite to be exited, its type, value, and traceback are passed as arguments to \_\_exit\_\_(). Otherwise, three None arguments are supplied.

If the suite was exited due to an exception, and the return value from the \_\_exit\_\_() method was false, the exception is reraised. If the return value was true, the exception is suppressed, and execution continues with the statement following the with statement.

If the suite was exited for any reason other than an exception, the return value from \_\_exit\_\_() is ignored, and execution proceeds at the normal location for the kind of exit that was taken.

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

**with** A() **as** a, B() **as** b:

suite

is equivalent to

**with** A() **as** a:

**with** B() **as** b:

suite

Python doc: <https://docs.python.org/3/reference/compound_stmts.html#the-with-statement>

Python doc: <https://docs.python.org/3/whatsnew/2.6.html#pep-343-the-with-statement>

Python doc: [https://docs.python.org/3/reference/compound\_stmts.html#](https://docs.python.org/3/reference/compound_stmts.html)

More explanation and example: <http://preshing.com/20110920/the-python-with-statement-by-example/>

* 1. Looping techniques:

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.

>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}

>>> for k, v in knights.items():

... print(k, v)

...

gallahad the pure

robin the brave

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the [enumerate()](https://docs.python.org/3/library/functions.html#enumerate)function.

>>> for i, v in enumerate(['tic', 'tac', 'toe']):

... print(i, v)

...

0 tic

1 tac

2 toe

To loop over two or more sequences at the same time, the entries can be paired with the [zip()](https://docs.python.org/3/library/functions.html#zip) function.

>>> questions = ['name', 'quest', 'favorite color']

>>> answers = ['lancelot', 'the holy grail', 'blue']

>>> for q, a in zip(questions, answers):

... print('What is your {0}? It is {1}.'.format(q, a))

...

What is your name? It is lancelot.

What is your quest? It is the holy grail.

What is your favorite color? It is blue.

To loop over a sequence in sorted order, use the [sorted()](https://docs.python.org/3/library/functions.html#sorted) function which returns a new sorted list while leaving the source unaltered.

>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']

>>> for f in sorted(set(basket)):

... print(f)

...

apple

banana

orange

pear

It is sometimes tempting to change a list while you are looping over it; however, it is often simpler and safer to create a new list instead.

>>> import math

>>> raw\_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]

>>> filtered\_data = []

>>> for value in raw\_data:

... if not math.isnan(value):

... filtered\_data.append(value)

...

>>> filtered\_data

[56.2, 51.7, 55.3, 52.5, 47.8]

* 1. Text-mode interactivity:

|  |  |
| --- | --- |
| Statement | Explanation |
| raw\_input (v: 2.7), input([“*prompt*”]) | input([“*prompt*”]) function prints the “*prompt*”(optional parameter) as a prompt for user and read the text user types in. For example:  name = input (“Your name: ”)  will give the user a prompt to type their name and will assign to a variable ‘*name*’ once user finishes typing and presses enter.  raw\_input is deprecated post version 2.7 |
| getpass.getpass(*prompt*=“password: ”, stream=None) | A *‘getpass*’ module contains a ‘*getpass*” function which prompts the user for a password without echoing. The user is prompted using the string *prompt*, which defaults to 'Password: '  If echo free input is unavailable getpass() falls back to printing a warning message to *stream* and reading from sys.stdin and issuing a GetPassWarning. |
| getpass.getuser() | A *‘getpass*’ module contains a ‘*getuser()*’ function which returns the “login name” of the user.  This function checks the environment variables LOGNAME, USER, LNAME and USERNAME, in order, and returns the value of the first one which is set to a non-empty string. If none are set, the login name from the password database is returned on systems which support the [pwd](https://docs.python.org/3.7/library/pwd.html" \l "module-pwd" \o "pwd: The password database (getpwnam() and friends). (Unix)) module, otherwise, an exception is raised. |

1. Data types:
   1. List:

Given the following lists,

a = [11, 22, 33, 44, 55, 66, 77, 88]

b = [‘|||’]

|  |  |
| --- | --- |
| Example | Explanation |
| print(b \* 4) | print result multiplying the list 4 times, will output [‘|||’, ‘|||’, ‘|||’, ‘|||’] |
| print(a[2 : 4]) | print from item2 to item4(excluding 4th) in the list a, will output [33, 44] |
| print(a[0 : len(a)]) | print from first to the last item in list a |
| print(a[-1]) | print the last item in the list |
| print(a[-2]) | print the 2nd to last item in the list |
| print(a[::2]) | print from first to last with step 2, will output [11, 33, 55, 77]. Default params inside the [] are first\_element:last\_element:step\_1 |
| a.clear() | Removes all items in the list |
| a.pop() | Removes the last item (list is LIFO) in the list |
| a.pop([3]) | Removes the item at index[3], i.e. 44, from the list |
|  |  |

Python doc: <https://docs.python.org/3/tutorial/datastructures.html#more-on-lists>

* 1. Tuples:

Tuples are immutable lists that are enclosed in parenthesis but they can contain mutable items. Ex: (11, 22, 33, 444)

Given the following tuples,

t = 12345, 54321, 'hello!'

|  |  |
| --- | --- |
| Example | Output |
| t[0] | 12345 |
| t | (12345, 54321, 'hello!') |
| # Tuples may be nested:  ... u = t, (1, 2, 3, 4, 5)  >>> u | ((12345, 54321, 'hello!'), (1, 2, 3, 4, 5)) |

* 1. Set:

A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the [set()](https://docs.python.org/3/library/stdtypes.html#set) function can be used to create sets. Note: to create an empty set you have to use set(), not {}; the latter creates an empty dictionary.

Example:

>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}

>>> print(basket) # show that duplicates have been removed

{'orange', 'banana', 'pear', 'apple'}

>>> 'orange' in basket # fast membership testing

True

>>> 'crabgrass' in basket

False

Demonstrate set operations on unique letters from two words

>>> a = set('abracadabra')

>>> b = set('alacazam')

>>> a # unique letters in a

{'a', 'r', 'b', 'c', 'd'}

>>> a - b # letters in a but not in b

{'r', 'd', 'b'}

>>> a | b # letters in either a or b

{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}

>>> a & b # letters in both a and b

{'a', 'c'}

>>> a ^ b # letters in a or b but not both

{'r', 'd', 'b', 'm', 'z', 'l'}

Python doc: <https://docs.python.org/3/library/stdtypes.html#set>

* 1. Dictionary:

1. Key:Value pairs in curly brackets, ex: {“one”:1, “two”:2, “three”:3}
2. Keys are immutable
3. Keys must be unique while values can be duplicated
4. Items in a dictionary are unordered

To illustrate, the following examples all return a dictionary equal to {"one": 1, "two": 2, "three": 3}:

>>> a = dict(one=1, two=2, three=3)

>>> b = {'one': 1, 'two': 2, 'three': 3}

>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))

>>> d = dict([('two', 2), ('one', 1), ('three', 3)])

>>> e = dict({'three': 3, 'one': 1, 'two': 2})

|  |  |
| --- | --- |
| Example | Explanation |
|  |  |
|  |  |

Python doc: <https://docs.python.org/3/library/stdtypes.html#mapping-types-dict>

* 1. Range:

The [range](https://docs.python.org/3/library/stdtypes.html#range) type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in [for](https://docs.python.org/3/reference/compound_stmts.html#for) loops.

For range(n) or range(0,n), the last item in the range is n-1.

The arguments to the range constructor must be integers (either built-in int or any object that implements the \_\_index\_\_ special method). If the step argument is omitted, it defaults to 1. If the start argument is omitted, it defaults to 0. If step is zero, ValueError is raised.

|  |  |
| --- | --- |
| Syntax | Example |
| range(stop) | Looping will continue until *stop-1* index. For example:  **>>>** list(range(10))  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] |
| Range(start, stop[,step]) | Looping starts at *start* index and stops at *stop-1*. The parameter s*tep* is optional and if absent with default to 1. For example:  **>>>** list(range(0, 30, 5))  [0, 5, 10, 15, 20, 25]  **>>>** list(range(0, 10, 3))  [0, 3, 6, 9]  **>>>** list(range(0, -10, -1))  [0, -1, -2, -3, -4, -5, -6, -7, -8, -9] |

Python doc: <https://docs.python.org/3/library/stdtypes.html#ranges>

* 1. String:

|  |  |
| --- | --- |
| Example | Explanation |
|  |  |
|  |  |

Python doc: <https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str>

1. Module:

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a script. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you’ve written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module; definitions from a module can be imported into other modules or into the main module

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module’s name (as a string) is available as the value of the global variable \_\_name\_\_. For instance, use your favorite text editor to create a file called fibo.py in the current directory with the following contents:

# Fibonacci numbers module

def fib(n): # write Fibonacci series up to n

a, b = 0, 1

while b < n:

print(b, end=' ')

a, b = b, a+b

print()

def fib2(n): # return Fibonacci series up to n

result = []

a, b = 0, 1

while b < n:

result.append(b)

a, b = b, a+b

return result

Now enter the Python interpreter and import this module with the following command:

>>>

>>> import fibo

This does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters the module name fibo there. Using the module name you can access the functions:

>>>

>>> fibo.fib(1000)

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

>>> fibo.fib2(100)

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

>>> fibo.\_\_name\_\_

'fibo'

If you intend to use a function often you can assign it to a local name:

>>>

>>> fib = fibo.fib

>>> fib(500)

1 1 2 3 5 8 13 21 34 55 89 144 233 377

* 1. Standard Modules

Python comes with a library of standard modules, described in a separate document, the Python Library Reference (“Library Reference” hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the [winreg](https://docs.python.org/3.6/library/winreg.html" \l "module-winreg" \o "winreg: Routines and objects for manipulating the Windows registry. (Windows)) module is only provided on Windows systems. One particular module deserves some attention: [sys](https://docs.python.org/3.6/library/sys.html#module-sys), which is built into every Python interpreter. The variables sys.ps1 and sys.ps2 define the strings used as primary and secondary prompts:

>>> import sys

>>> sys.ps1

'>>> '

>>> sys.ps2

'... '

>>> sys.ps1 = 'C> '

C> print('Yuck!')

Yuck!

C>

These two variables are only defined if the interpreter is in interactive mode.

The variable sys.path is a list of strings that determines the interpreter’s search path for modules. It is initialized to a default path taken from the environment variable [PYTHONPATH](https://docs.python.org/3.6/using/cmdline.html#envvar-PYTHONPATH), or from a built-in default if [PYTHONPATH](https://docs.python.org/3.6/using/cmdline.html#envvar-PYTHONPATH) is not set. You can modify it using standard list operations:

>>>

>>> import sys

>>> sys.path.append('/ufs/guido/lib/python')

* 1. Buit-in Functions:

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Built-in Functions |  |  |
| [abs()](https://docs.python.org/3/library/functions.html#abs) | [dict()](https://docs.python.org/3/library/functions.html#func-dict) | [help()](https://docs.python.org/3/library/functions.html#help) | [min()](https://docs.python.org/3/library/functions.html#min) | [setattr()](https://docs.python.org/3/library/functions.html#setattr) |
| [all()](https://docs.python.org/3/library/functions.html#all) | [dir()](https://docs.python.org/3/library/functions.html#dir) | [hex()](https://docs.python.org/3/library/functions.html#hex) | [next()](https://docs.python.org/3/library/functions.html#next) | [slice()](https://docs.python.org/3/library/functions.html#slice) |
| [any()](https://docs.python.org/3/library/functions.html#any) | [divmod()](https://docs.python.org/3/library/functions.html#divmod) | [id()](https://docs.python.org/3/library/functions.html#id) | [object()](https://docs.python.org/3/library/functions.html#object) | [sorted()](https://docs.python.org/3/library/functions.html#sorted) |
| [ascii()](https://docs.python.org/3/library/functions.html#ascii) | [enumerate()](https://docs.python.org/3/library/functions.html#enumerate) | [input()](https://docs.python.org/3/library/functions.html#input) | [oct()](https://docs.python.org/3/library/functions.html#oct) | [staticmethod()](https://docs.python.org/3/library/functions.html#staticmethod) |
| [bin()](https://docs.python.org/3/library/functions.html#bin) | [eval()](https://docs.python.org/3/library/functions.html#eval) | [int()](https://docs.python.org/3/library/functions.html#int) | [open()](https://docs.python.org/3/library/functions.html#open) | [str()](https://docs.python.org/3/library/functions.html#func-str) |
| [bool()](https://docs.python.org/3/library/functions.html#bool) | [exec()](https://docs.python.org/3/library/functions.html#exec) | [isinstance()](https://docs.python.org/3/library/functions.html#isinstance) | [ord()](https://docs.python.org/3/library/functions.html#ord) | [sum()](https://docs.python.org/3/library/functions.html#sum) |
| [bytearray()](https://docs.python.org/3/library/functions.html#func-bytearray) | [filter()](https://docs.python.org/3/library/functions.html#filter) | [issubclass()](https://docs.python.org/3/library/functions.html#issubclass) | [pow()](https://docs.python.org/3/library/functions.html#pow) | [super()](https://docs.python.org/3/library/functions.html#super) |
| [bytes()](https://docs.python.org/3/library/functions.html#func-bytes) | [float()](https://docs.python.org/3/library/functions.html#float) | [iter()](https://docs.python.org/3/library/functions.html#iter) | [print()](https://docs.python.org/3/library/functions.html#print) | [tuple()](https://docs.python.org/3/library/functions.html#func-tuple) |
| [callable()](https://docs.python.org/3/library/functions.html#callable) | [format()](https://docs.python.org/3/library/functions.html#format) | [len()](https://docs.python.org/3/library/functions.html#len) | [property()](https://docs.python.org/3/library/functions.html#property) | [type()](https://docs.python.org/3/library/functions.html#type) |
| [chr()](https://docs.python.org/3/library/functions.html#chr) | [frozenset()](https://docs.python.org/3/library/functions.html#func-frozenset) | [list()](https://docs.python.org/3/library/functions.html#func-list) | [range()](https://docs.python.org/3/library/functions.html#func-range) | [vars()](https://docs.python.org/3/library/functions.html#vars) |
| [classmethod()](https://docs.python.org/3/library/functions.html#classmethod) | [getattr()](https://docs.python.org/3/library/functions.html#getattr) | [locals()](https://docs.python.org/3/library/functions.html#locals) | [repr()](https://docs.python.org/3/library/functions.html#repr) | [zip()](https://docs.python.org/3/library/functions.html#zip) |
| [compile()](https://docs.python.org/3/library/functions.html#compile) | [globals()](https://docs.python.org/3/library/functions.html#globals) | [map()](https://docs.python.org/3/library/functions.html#map) | [reversed()](https://docs.python.org/3/library/functions.html#reversed) | [\_\_import\_\_()](https://docs.python.org/3/library/functions.html#__import__) |
| [complex()](https://docs.python.org/3/library/functions.html#complex) | [hasattr()](https://docs.python.org/3/library/functions.html#hasattr) | [max()](https://docs.python.org/3/library/functions.html#max) | [round()](https://docs.python.org/3/library/functions.html#round) |  |
| [delattr()](https://docs.python.org/3/library/functions.html#delattr) | [hash()](https://docs.python.org/3/library/functions.html#hash) | [memoryview()](https://docs.python.org/3/library/functions.html#func-memoryview) | [set()](https://docs.python.org/3/library/functions.html#func-set) |  |

* 1. The [dir()](https://docs.python.org/3.6/library/functions.html" \l "dir" \o "dir) Function

The built-in function [dir()](https://docs.python.org/3.6/library/functions.html" \l "dir" \o "dir) is used to find out which names a module defines. It returns a sorted list of strings:

>>>

>>> import fibo, sys

>>> dir(fibo)

['\_\_name\_\_', 'fib', 'fib2']

>>> dir(sys)

['\_\_displayhook\_\_', '\_\_doc\_\_', '\_\_excepthook\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_stderr\_\_', '\_\_stdin\_\_', '\_\_stdout\_\_',

'\_clear\_type\_cache', '\_current\_frames', '\_debugmallocstats', '\_getframe',

'\_home', '\_mercurial', '\_xoptions', 'abiflags', 'api\_version', 'argv',

'base\_exec\_prefix', 'base\_prefix', 'builtin\_module\_names', 'byteorder',

'call\_tracing', 'callstats', 'copyright', 'displayhook',

'dont\_write\_bytecode', 'exc\_info', 'excepthook', 'exec\_prefix',

'executable', 'exit', 'flags', 'float\_info', 'float\_repr\_style',

'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',

'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',

'getrefcount', 'getsizeof', 'getswitchinterval', 'gettotalrefcount',

'gettrace', 'hash\_info', 'hexversion', 'implementation', 'int\_info',

'intern', 'maxsize', 'maxunicode', 'meta\_path', 'modules', 'path',

'path\_hooks', 'path\_importer\_cache', 'platform', 'prefix', 'ps1',

'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit',

'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout',

'thread\_info', 'version', 'version\_info', 'warnoptions']

Without arguments, [dir()](https://docs.python.org/3.6/library/functions.html" \l "dir" \o "dir) lists the names you have defined currently:

>>>

>>> a = [1, 2, 3, 4, 5]

>>> import fibo

>>> fib = fibo.fib

>>> dir()

['\_\_builtins\_\_', '\_\_name\_\_', 'a', 'fib', 'fibo', 'sys']

**Note that it lists all types of names: variables, modules, functions, etc.**

[dir()](https://docs.python.org/3.6/library/functions.html#dir) does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module [builtins](https://docs.python.org/3.6/library/builtins.html#module-builtins):

>>>

>>> import builtins

>>> dir(builtins)

['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',

'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',

'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',

'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',

'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',

'FileExistsError', 'FileNotFoundError', 'FloatingPointError',

'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',

'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',

'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',

'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',

'NotImplementedError', 'OSError', 'OverflowError',

'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',

'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',

'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',

'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',

'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',

'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',

'ValueError', 'Warning', 'ZeroDivisionError', '\_', '\_\_build\_class\_\_',

'\_\_debug\_\_', '\_\_doc\_\_', '\_\_import\_\_', '\_\_name\_\_', '\_\_package\_\_', 'abs',

'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',

'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',

'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',

'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',

'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',

'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',

'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',

'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',

'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',

'zip']

Python doc: <https://docs.python.org/3.6/tutorial/modules.html>

1. Packages

Packages are a way of structuring Python’s module namespace by using “dotted module names”. For example, the module name A.B designates a submodule named B in a package named A. Just like the use of modules saves the authors of different modules from having to worry about each other’s global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or the Python Imaging Library from having to worry about each other’s module names.

Suppose you want to design a collection of modules (a “package”) for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: .wav, .aiff, .au), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here’s a possible structure for your package (expressed in terms of a hierarchical filesystem):

sound/ Top-level package

\_\_init\_\_.py Initialize the sound package

formats/ Subpackage for file format conversions

\_\_init\_\_.py

wavread.py

wavwrite.py

aiffread.py

aiffwrite.py

auread.py

auwrite.py

...

effects/ Subpackage for sound effects

\_\_init\_\_.py

echo.py

surround.py

reverse.py

...

filters/ Subpackage for filters

\_\_init\_\_.py

equalizer.py

vocoder.py

karaoke.py

...

When importing the package, Python searches through the directories on sys.path looking for the package subdirectory.

The \_\_init\_\_.py files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as string, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, \_\_init\_\_.py can just be an empty file, but it can also execute initialization code for the package or set the \_\_all\_\_ variable, described later.

Users of the package can import individual modules from the package, for example:

import sound.effects.echo

This loads the submodule sound.effects.echo. It must be referenced with its full name.

sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)

An alternative way of importing the submodule is:

from sound.effects import echo

This also loads the submodule echo, and makes it available without its package prefix, so it can be used as follows:

echo.echofilter(input, output, delay=0.7, atten=4)

Yet another variation is to import the desired function or variable directly:

from sound.effects.echo import echofilter

Again, this loads the submodule echo, but this makes its function echofilter() directly available:

echofilter(input, output, delay=0.7, atten=4)

Note that when using from package import item, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The import statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an [ImportError](https://docs.python.org/3.6/library/exceptions.html" \l "ImportError" \o "ImportError) exception is raised.

**Contrarily, when using syntax like import item.subitem.subsubitem, each item except for the last must be a package; the last item can be a module or a package but can’t be a class or function or variable defined in the previous item.**

* 1. Importing \* From a Package

Now what happens when the user writes from sound.effects import \*? Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all. This could take a long time and importing sub-modules might have unwanted side-effects that should only happen when the sub-module is explicitly imported.

The only solution is for the package author to provide an explicit index of the package. The [import](https://docs.python.org/3.6/reference/simple_stmts.html#import) statement uses the following convention: if a package’s \_\_init\_\_.py code defines a list named \_\_all\_\_, it is taken to be the list of module names that should be imported when from package import \* is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don’t see a use for importing \* from their package. For example, the file sound/effects/\_\_init\_\_.py could contain the following code:

\_\_all\_\_ = ["echo", "surround", "reverse"]

This would mean that from sound.effects import \* would import the three named submodules of the sound package.

If \_\_all\_\_ is not defined, the statement from sound.effects import \* does not import all submodules from the package sound.effects into the current namespace; it only ensures that the package sound.effects has been imported (possibly running any initialization code in \_\_init\_\_.py) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by \_\_init\_\_.py. It also includes any submodules of the package that were explicitly loaded by previous [import](https://docs.python.org/3.6/reference/simple_stmts.html#import) statements. Consider this code:

import sound.effects.echo

import sound.effects.surround

from sound.effects import \*

In this example, the echo and surround modules are imported in the current namespace because they are defined in the sound.effects package when the from...import statement is executed. (This also works when \_\_all\_\_ is defined.)

Although certain modules are designed to export only names that follow certain patterns when you use import \*, it is still considered bad practice in production code.

Remember, there is nothing wrong with using from Package import specific\_submodule! In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.

Python doc: <https://docs.python.org/3.6/tutorial/modules.html>

1. Classes

Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are public (except see below [Private Variables](https://docs.python.org/3.6/tutorial/classes.html#tut-private)), and all member functions are virtual.

Classes themselves are objects. built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

* 1. Python Scopes and Namespaces

A namespace is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function maximize without confusion — users of the modules must prefix it with the module name.

The word attribute is used for any name following a dot — for example, in the expression z.real; real is an attribute of the object z. Strictly speaking, references to names in modules are attribute references: in the expression modname.funcname, modname is a module object and funcname is an attribute of it.

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can writemodname.the\_answer = 42. Writable attributes may also be deleted with the [del](https://docs.python.org/3.6/reference/simple_stmts.html#del) statement. For example, del modname.the\_answer will remove the attribute the\_answer from the object named by modname.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are at least three nested scopes whose namespaces are directly accessible:

* the innermost scope, which is searched first, contains the local names
* the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names
* the next-to-last scope contains the current module’s global names
* the outermost scope (searched last) is the namespace containing built-in names

A special quirk of Python is that – if no [global](https://docs.python.org/3.6/reference/simple_stmts.html#global) statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement del x removes the binding of x from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, [import](https://docs.python.org/3.6/reference/simple_stmts.html#import) statements and function definitions bind the module or function name in the local scope.

The [global](https://docs.python.org/3.6/reference/simple_stmts.html#global) statement can be used to indicate that particular variables live in the global scope and should be rebound there; the [nonlocal](https://docs.python.org/3.6/reference/simple_stmts.html#nonlocal)statement indicates that particular variables live in an enclosing scope and should be rebound there.

* + 1. Scopes and Namespaces Example

This is an example demonstrating how to reference the different scopes and namespaces, and how [global](https://docs.python.org/3.6/reference/simple_stmts.html#global) and [nonlocal](https://docs.python.org/3.6/reference/simple_stmts.html#nonlocal) affect variable binding:

def scope\_test():

def do\_local():

spam = "local spam"

def do\_nonlocal():

nonlocal spam

spam = "nonlocal spam"

def do\_global():

global spam

spam = "global spam"

spam = "test spam"

do\_local()

print("After local assignment:", spam)

do\_nonlocal()

print("After nonlocal assignment:", spam)

do\_global()

print("After global assignment:", spam)

scope\_test()

print("In global scope:", spam)

The output of the example code is:

After local assignment: test spam

After nonlocal assignment: nonlocal spam

After global assignment: nonlocal spam

In global scope: global spam

Note how the local assignment (which is default) didn’t change scope\_test‘s binding of spam. The [nonlocal](https://docs.python.org/3.6/reference/simple_stmts.html#nonlocal) assignment changed scope\_test‘s binding of spam, and the [global](https://docs.python.org/3.6/reference/simple_stmts.html#global) assignment changed the module-level binding.

You can also see that there was no previous binding for spam before the [global](https://docs.python.org/3.6/reference/simple_stmts.html#global) assignment.

* 1. Class Definition Syntax

The simplest form of class definition looks like this:

class ClassName:

<statement-1>

.

.

.

<statement-N>

Class definitions, like function definitions ([def](https://docs.python.org/3.6/reference/compound_stmts.html" \l "def) statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an [if](https://docs.python.org/3.6/reference/compound_stmts.html#if) statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful. When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

* 1. Class Objects

Class objects support two kinds of operations:

* attribute references and
* instantiation.

Attribute references use the standard syntax used for all attribute references in Python: obj.name. Valid attribute names are all the names that were in the class’s namespace when the class object was created. So, if the class definition looked like this:

class MyClass:

"""A simple example class"""

i = 12345

def f(self):

return 'hello world'

then MyClass.i and MyClass.f are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of MyClass.i by assignment. \_\_doc\_\_ is also a valid attribute, returning the docstring belonging to the class: "A simple example class".

Class instantiation uses function notation. Just pretend that the class object is a parameter-less function that returns a new instance of the class. For example (assuming the above class):

x = MyClass()

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

The instantiation operation (“calling” a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named [\_\_init\_\_()](https://docs.python.org/3.6/reference/datamodel.html#object.__init__), like this:

def \_\_init\_\_(self):

self.data = []

When a class defines an [\_\_init\_\_()](https://docs.python.org/3.6/reference/datamodel.html#object.__init__) method, class instantiation automatically invokes [\_\_init\_\_()](https://docs.python.org/3.6/reference/datamodel.html#object.__init__) for the newly-created class instance. So in this example, a new, initialized instance can be obtained by:

x = MyClass()

Of course, the [\_\_init\_\_()](https://docs.python.org/3.6/reference/datamodel.html#object.__init__) method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to [\_\_init\_\_()](https://docs.python.org/3.6/reference/datamodel.html#object.__init__). For example,

>>>

>>> class Complex:

... def \_\_init\_\_(self, realpart, imagpart):

... self.r = realpart

... self.i = imagpart

...

>>> x = Complex(3.0, -4.5)

>>> x.r, x.i

(3.0, -4.5)

* 1. Instance Objects

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

Data attributes correspond to “instance variables”. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if x is the instance of MyClass created above, the following piece of code will print the value 16, without leaving a trace:

x.counter = 1

while x.counter < 10:

x.counter = x.counter \* 2

print(x.counter)

del x.counter

The other kind of instance attribute reference is a method. A method is a function that “belongs to” an object. (In Python, the term method is not unique to class instances: other object types can have methods as well. For example, list objects have methods called append, insert, remove, sort, and so on. However, in the following discussion, we’ll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.)

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, x.f is a valid method reference, since MyClass.f is a function, but x.i is not, since MyClass.i is not. But x.f is not the same thing as MyClass.f — it is a method object, not a function object.

* 1. Method Objects

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

x.f()

In the MyClass example, this will return the string 'hello world'. However, it is not necessary to call a method right away: x.f is a method object, and can be stored away and called at a later time. For example:

xf = x.f

while True:

print(xf())

will continue to print hello world until the end of time.

x.f() was called without an argument above, even though the function definition for f() specified an argument. So what happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn’t actually used.

However, the special thing about methods is that the instance object is passed as the first argument of the function. In our example, the call x.f() is exactly equivalent to MyClass.f(x). In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method’s instance object before the first argument.

* 1. Class and Instance Variables

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

class Dog:

kind = 'canine' # class variable shared by all instances

def \_\_init\_\_(self, name):

self.name = name # instance variable unique to each instance

>>> d = Dog('Fido')

>>> e = Dog('Buddy')

>>> d.kind # shared by all dogs

'canine'

>>> e.kind # shared by all dogs

'canine'

>>> d.name # unique to d

'Fido'

>>> e.name # unique to e

'Buddy'

As discussed earlier, shared data can have possibly surprising effects with involving [mutable](https://docs.python.org/3.6/glossary.html#term-mutable) objects such as lists and dictionaries. For example, the tricks list in the following code should not be used as a class variable because just a single list would be shared by all Dog instances:

class Dog:

tricks = [] # mistaken use of a class variable

def \_\_init\_\_(self, name):

self.name = name

def add\_trick(self, trick):

self.tricks.append(trick)

>>> d = Dog('Fido')

>>> e = Dog('Buddy')

>>> d.add\_trick('roll over')

>>> e.add\_trick('play dead')

>>> d.tricks # unexpectedly shared by all dogs

['roll over', 'play dead']

Correct design of the class should use an instance variable instead:

class Dog:

def \_\_init\_\_(self, name):

self.name = name

self.tricks = [] # creates a new empty list for each dog

def add\_trick(self, trick):

self.tricks.append(trick)

>>> d = Dog('Fido')

>>> e = Dog('Buddy')

>>> d.add\_trick('roll over')

>>> e.add\_trick('play dead')

>>> d.tricks

['roll over']

>>> e.tricks

['play dead']

* 1. Inheritance

Of course, a language feature would not be worthy of the name “class” without supporting inheritance. The syntax for a derived class definition looks like this:

class DerivedClassName(BaseClassName):

<statement-1>

.

.

.

<statement-N>

The name BaseClassName must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

class DerivedClassName(modname.BaseClassName):

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There’s nothing special about instantiation of derived classes: DerivedClassName() creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it.

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call BaseClassName.methodname(self, arguments). This is occasionally useful to clients as well. (Note that this only works if the base class is accessible as BaseClassName in the global scope.)

Python has two built-in functions that work with inheritance:

* Use [isinstance()](https://docs.python.org/3.6/library/functions.html" \l "isinstance" \o "isinstance) to check an instance’s type: isinstance(obj, int) will be True only if obj.\_\_class\_\_ is [int](https://docs.python.org/3.6/library/functions.html" \l "int" \o "int) or some class derived from [int](https://docs.python.org/3.6/library/functions.html#int).
* Use [issubclass()](https://docs.python.org/3.6/library/functions.html" \l "issubclass" \o "issubclass) to check class inheritance: issubclass(bool, int) is True since [bool](https://docs.python.org/3.6/library/functions.html#bool) is a subclass of [int](https://docs.python.org/3.6/library/functions.html#int). However,issubclass(float, int) is False since [float](https://docs.python.org/3.6/library/functions.html#float) is not a subclass of [int](https://docs.python.org/3.6/library/functions.html#int).
  + 1. Multiple Inheritance

Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

class DerivedClassName(Base1, Base2, Base3):

<statement-1>

.

.

.

<statement-N>

For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in DerivedClassName, it is searched for in Base1, then (recursively) in the base classes of Base1, and if it was not found there, it was searched for in Base2, and so on.

* 1. Private Variables

“Private” instance variables that cannot be accessed except from inside an object don’t exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. \_spam) should be treated as a non-public part of the API (whether it is a function, a method or a data member).

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called name mangling. Any identifier of the form \_\_spam (at least two leading underscores, at most one trailing underscore) is textually replaced with \_classname\_\_spam, where classname is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. For example:

class Mapping:

def \_\_init\_\_(self, iterable):

self.items\_list = []

self.\_\_update(iterable)

def update(self, iterable):

for item in iterable:

self.items\_list.append(item)

\_\_update = update # private copy of original update() method

class MappingSubclass(Mapping):

def update(self, keys, values):

# provides new signature for update()

# but does not break \_\_init\_\_()

for item in zip(keys, values):

self.items\_list.append(item)

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

* 1. Struct like objects:

Sometimes it is useful to have a data type similar to the Pascal “record” or C “struct”, bundling together a few named data items. An empty class definition will do nicely:

class Employee:

pass

john = Employee() # Create an empty employee record

# Fill the fields of the record

john.name = 'John Doe'

john.dept = 'computer lab'

john.salary = 1000

Python doc: <https://docs.python.org/3.6/tutorial/classes.html>

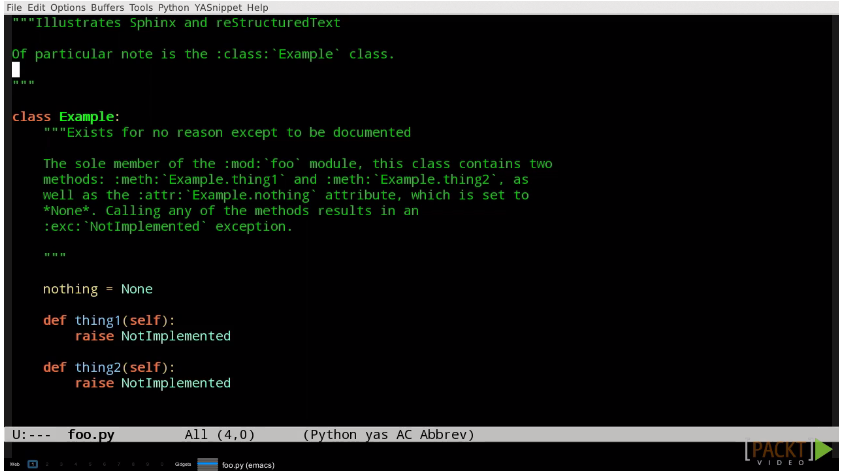
1. Python package manager
   1. pip
2. Python Basic Best Practices:
   1. PEP 8 and writing readable code: spaces and tabs, standard code layout and Naming convensions
3. Python PEP8 code standard site: <https://www.python.org/dev/peps/pep-0008/>
4. 4 spaces to indicate one level of indentation, don’t use tab characters
5. A single line of code should not exceed 17-19 characters
6. Import statements should be at the top of the module
7. Standard libraries should be imported first, then third party modules and import from other modules in the same project
8. There should be a blank line between each group of imports
9. Classes and Functions at the top level can have 2 blank lines between them
10. Methods within a Class should have 1 blank line separating them
11. Within a method, blank lines should be inserted to separate conceptual grouping of code
12. Don’t insert extra space before parenthesis, brackets or braces
13. Don’t insert space before commas and colons
14. Always put a single space on either sides of binary operators
15. Don’t put more than 1 statement on the same line
16. Comments should be in human language
17. Comment should precede every section of code describing the code and be indented at the code level
18. Every **public** module, class, or method should have a properly formatted doc string
19. Objects are named according how they are used, for example a *Factory* function should be named as if it were a class because the usage case is the same as it were a class
20. Packages and modules have to have short names and lowercase letters
21. Module names have leading and trailing double underscores
22. Class names are CamelCased starting with a Capital letter
23. Exception are classes so should follow naming convention for Classes but the last word in the name should be *Err*.
24. Functions or methods, instance variables and global variables should all be lowercased with underscore separating words
25. Private (internal) methods or variable should begin with a single underscore
26. The first parameter of an instance method should always be named *self*
27. Constants to be written in all capital letters with underscore separating words
    1. Using Version Control

Git

* 1. Using venv to create Stable and Isolated

1. Venv keeps the project’s underpinning stable. The python version and installed libraries stay the same for the life of the project.
2. Created using command:

python –m venv

1. Inside of the venv pip installs the private packages automatically so there’s no need for ---user switch
   1. Docstrings:
2. PEP 257 is the python standard for docstrings
3. Use triple quotes
4. Closing triple quotes should be on their own line if the docstring is longer than one line
5. First line should present a short description, such as this program shows how to get from A to B. This can be followed by further documentation with a line in-between for more detailed description.
6. First line is shown as a popup tooltip in programs explaining what the module is about
7. 
8. **Sphinx** is installed using pip (python –m pip install –user Sphinx) and can be used as a documentation compiler to generate html documentation files.
   1. Doctest

Doctest is a tool that checks whether the documentation agrees with the code. If the documentation if correct, a failing example means that the code is flawed.

1. Creating command-line utility
   1. Making a Package executable via python –m

“Python –m” actually asks python to run a module which is the same mechanism as importing a module’s name to find a module and executing it. When the “python –m” command followed by a package name is run, python looks for module named \_\_main\_\_ in the package name, which is equivalent to “python –m packagename.\_\_main\_\_”

* 1. Handling command-line arguments with argparse

1. Import the required module, ex: import argparse
2. Create an argparse instance, ex: parser = argparse.argumentparser(). However this could be made more useful by passing in some parameters inside the constructor, such as parser = argparse.argumentparser(prog = ‘python –m pipeline, description = \_\_doc\_\_)
3. The argparse instance parser also has add\_argument method that can take one or more parameters, some optional and some standard. Optional parameters start with a –. Example: parser.add\_argument(‘-p’, ‘—print’, action=’strore\_true’, default=false)
4. The add\_argument can also take one parameter at a time with number of expected arguments, such as parser.add\_argument(‘name’, nargs=’+’)
5. Finally call that instance’s parse\_args function. Ex: args = parser.parse\_args()
   1. Text-mode interactivity:

|  |  |
| --- | --- |
| Statement | Explanation |
| raw\_input (v: 2.7), input(“*message*”) | Takes user input as a string, if given a string parameter, e.g. input(“message”), it prints the string before the prompt. |
| getpass(“*Prompt*”) | Takes user’s password without displaying them on-screen in clear text |
| pprint | While “print” statement cannot do a good job printing complex data type, “pprint” addresses this nicely. Ex:  pprint([{1:2, 3;4}, {5:6, 7:list(range(25))}]) |
|  |  |

1. Internet Data Handling
   1. json – JSON encoder and decoder
2. doc: [https://docs.python.org/3.6/library/json.html#](https://docs.python.org/3.6/library/json.html)
3. Supports the following objects and types by default:

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

1. Some seriously good stuff:

|  |  |
| --- | --- |
| Topic | Resource |
| Python Official Tutorial | <https://docs.python.org/3.6/tutorial/index.html> |
| Downloads | <https://www.python.org/downloads/> |
| Built-in Functions | <https://docs.python.org/3/library/functions.html> |
| Python HOWTOs | <https://docs.python.org/3/howto/index.html> |
| Python Package Index | <https://pypi.python.org/pypi> |
| Requests: HTTP for Humans (Non-GMO HTTP library for Python) | <http://docs.python-requests.org/en/master/>  **Warning**: Recreational use of other HTTP libraries may result in dangerous side-effects, including: security vulnerabilities, verbose code, reinventing the wheel, constantly reading documentation, depression, headaches, or even death. |
| Internet Protocols and Support | <https://docs.python.org/3.6/library/internet.html> |
|  |  |
|  |  |
|  |  |
|  |  |

1. Something else