Introduction to Python

Sandro Cumani

sandro.cumani@polito.it

Politecnico di Torino

Python

High-level, interpreted language

Multiple programming paradigms

- Procedural, object oriented, functional

Multi-platform open-source implementations (Windows, Linux, Mac OS)

We will use Python 3

- Available for different platforms
- Python tutorial available at https://docs.python.org/3/tutorial/

Python

You can install Python and the required libraries in many different ways (install packages, distribution packages, ...)

If you have no prior experience, you can use `Anaconda`, a framework that already includes Python and the numerical libraries we will need:
`https://www.anaconda.com/products/individual`

Python

The Python interpreter can execute scripts (i.e. code from source files)

```
$> python myscript.py
```

The interpreter also allows for interactive sessions

```
$> python
>>> print("Hello World")
Hello World
```

The Python shell is quite basic. Alternative shells such as IPython offer many more functionalities

In IPython, you can check function / method documentation simply calling `func?` (question mark included)

Numeric Types

Python natively supports several data types

Numerical types (int, float):

```
>>> 2 + 2
4

>>> 3.0 * 5.5
16.5
```

Operator `**` allows computing powers

```
>>> 3 ** 4
81
```

Numeric Types

Division `/` always results in float, integer division is obtained through `//`:

```
>>> 6 / 4
1.5

>>> 6 // 4
1
```

The remainder is computed with the operator `%`:

```
>>> 6 % 4
2
```

Strings

Strings:

- Sequences of characters enclosed in single quotes `'...'` or double quotes `"..."`

```
>>> "This is a string"
'This is a string'

>>> 'Also this is a string'
'Also this is a string'
```

Strings

Strings:

- Immutable (cannot be modified, but new strings can be obtained from existing ones)

```
>>> "This" + " is " + 'a string'
'This is a string'
```

Strings

Strings:

- Single quotes are allowed as characters in double-quoted strings and vice-versa. `\` allows escaping quotes

```
>>> "A ’ in a string"
"A ’ in a string"

>>> "A \" in a string"
’A " in a string’
```

Strings

Strings can be indexed (indexes start from 0 as in C)

```
>>> s = "Hello"

>>> s[0]
'H'
```

NOTE: in C, an element of a string is of type char. In Python, indexing a string works differently — `s[0]` is still a string

Strings

Negative indices can be used to index from the end (but -0 is equal to 0 and is the index of the first element)

```
>>> s = "Hello"

>>> s[-1]
'o'
```

Strings are immutable — we cannot assign a value to an element of a string

The built-in operator `len` allows computing the length of a string

```
>>> len(s)
5
```

Strings

Strings can be sliced — we can take the substring starting in s and ending in e (the ending point is always excluded)

```
>>> s[0:3]
'Hel'
```

The first and/or last elements can be omitted, and are implicitly assumed to be, respectively, 0 and the end of the string

```
>>> s[:3]
'Hel'

>>> s[3:]
'lo'
```

Strings

Negative indices can be used also in slices

```
>>> s[-3:-1]
'll'
```

Consecutive slices can be represented as `s[i:j]` and `s[j:k]`

Length of a slice:

```
>>> j-i
```

Accessing a slice of length `n`:

```
>>> s[i:i+n]
```

Objects variables and types

In Python everything is an object:

- int, float, str: built-in objects

We can bind values (objects) to names

A variable is a name bounded to a value

- In practice, the variable contains a reference (pointer) to the object

Strongly, dynamically typed language:

- Variables do not have types
- Values always have a type

Objects variables and types

No need to specify a variable type like in C

Variables can be bounded to values with different types

```
>>> a = 3

>>> print(a)
3

>>> a = 'Hello World'

>>> print(a)
Hello World
```

Objects variables and types

No need to declare variables in advance – however variables should be assigned a value before being used

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

>>> b = 3

>>> print(b)
3
```

Objects variables and types

Value types determine the operations that can be performed

Python allows for duck-typing

- Rather than checking the type of a value, we check whether it implements specific functionalities (i.e. methods) — more on this later

Data types

Python natively supports high-level data types, such as lists, tuples, hash tables and sets

A list can be built in Python by enclosing a sequence of comma-separated values in square brackets

```
>>> l = [1,2,3]

>>> l
[1, 2, 3]
```

Data types

List elements can be heterogeneous

```
>>> l = [1,'A',[4,5]]

>>> l
[1, 'A', [4, 5]]
```

The length of a list can be obtained from function `len`

```
>>> l = [1,2,3]

>>> len(l)
3
```

Data types

Lists can be indexed and sliced like strings

```
>>> l = [1,2,3,4,5]

>>> l[0]
1

>>> l[2:-1]
[3, 4]
```

Slicing a list returns a new list which contains references to the values of the original list (shallow copy)

More on slicing

The slicing operator allows also providing a step value

```
>>> l = [1,2,3,4,5]
>>> l[0:5:2]
[1,3,5]
```

Slice start and end can be omitted (implicitly assumed to be the beginning and end of the iterable)

The step can be negative – useful to traverse the iterable in reverse

```
>>> l = [1,2,3,4,5]
>>> l[::-1]
[5,4,3,2,1]
```

More on slicing

Pay attention when providing start or end with negative steps

```
>>> l = [1,2,3,4,5]
>>> l[0:5:-1]
[]
>>> l[5:0:-1]
[5, 4, 3, 2] # End point is always excluded
```

If we want to slice and reverse, better to do both steps separately

```
>>> l = [1,2,3,4,5]
>>> l[0:3][::-1]
[3,2,1]
```

Data types

A list can be modified assigning values to its elements (as if it were an array)

```
>>> l = [1,2,3,4,5]

>>> l[0] = 7

>>> l
[7, 2, 3, 4, 5]
```

Slices can be assigned new values

- Assigned values should be iterable (e.g. lists)
- Assignment can both replace and remove values

Data types

Assigning values to a slice:

```
>>> l = [1,2,3,4,5]

>>> l[1:4] = [11,12,13]

>>> l
[1, 11, 12, 13, 5]
```

Removing values (`[]` is an empty list):

```
>>> l[1:3] = []

>>> l
[1, 13, 5]
```

Data types

We can append an element to a list using its `append` method (more on methods later):

```
>>> l = [1,2,3]

>>> l.append(4)

>>> print(l)
[1,2,3,4]
```

Data types

We can extend a list using its `extend` method:

```
>>> l = [1,2,3]

>>> l.extend([4,5])

>>> print(l)

[1,2,3,4,5]
```

Data types

We can pop an element using its `pop` method (optional, pass the index of the element to pop, default is last element):

```
>>> l = [1,2,3]

>>> x = l.pop(1)

>>> print(l)

[1,3]

>>> print(x)

2
```

Data types

We can create a list from another iterable object (e.g. another list, a string, etc.) using `list`

```
>>> l = list([1,2,3])

>>> print(l)

[1,2,3]

>>> l=list('Hello')

>>> print(l)

['H', 'e', 'l', 'l', 'o']
```

Blocks, indentations and control structures

Conditional execution: `if`, `elif`, `else`

```
>>> x=3

>>> if x < 0:
...     print('Negative')
... elif x == 0:
...     print('Zero')
... else:
...     print('Positive')

Positive
```

Blocks, indentations and control structures

Python does not have explicit delimiters for blocks of code

Blocks are represented through indentation:

```
x = 3
if x < 0:
    x = x+1
    x = x*2
print(x)
```

Output: 3

```
x = 3
if x < 0:
    x = x+1
x = x*2
print(x)
```

Output: 6

Blocks, indentations and control structures

Conditional execution: `if`, `elif`, `else`

The condition should be a Boolean (type `bool`) value

`True` and `False` built-in values (type bool)

Comparison operators return bool values

- `==`, `<`, `>`, `<=`, `>=`, `!=`

Blocks, indentations and control structures

Any object can be converted to bool (for control flow statements the conversion is automatic).

Almost everything is converted to True, except:

- The Boolean value `False`
- The numerical values `0`, `0.0`
- Empty collections (e.g. empty list `[]` or empty string `''`)
- The special value `None`

Note: explicit casting can be achieved invoking `bool()` over the value. The same applies for other types, such as `int()` or `float()`

Blocks, indentations and control structures

Logical operators can be used to form more complex logical expressions

- `and`, `or`, `not`

```
>>> (2 == 3) or (1 < 3)

True

>>> not True

False
```

Blocks, indentations and control structures

Boolean operators may be applied to both Boolean and non-boolean values (and may return non boolean values)

The semantic is:

- `not x`: evaluate `x` and yield `True` if `x` is false, `False` otherwise
- `x and y`: evaluate `x`; if `x` is false, return its value, else evaluate y and return its value
- `x or y`: evaluate `x`; if `x` is true, return its value, else evaluate y and return its value

This allows for conditional evaluation of y (similar to C)

Blocks, indentations and control structures

Conditional execution: `if`, `elif`, `else`

Multiple `elif` conditions can be included — they act as a compact way to express else – if statements and avoid excessive code nesting

```
if x < 3:
    y = 4
else:
    if x > 10:
        y = 5
    else:
        y = 2
```

```
if x < 3:
    y = 4
elif x > 10:
    y = 5
else:
    y = 2
```

Blocks, indentations and control structures

Conditional execution: `if`, `elif`, `else`

The `else` block can be omitted

Blocks cannot be empty – the `pass` instruction can be used to do nothing

```
if x < 3:
    y = 4
print(y)
```

```
if x < 3:
    y = 4
else:
    pass
print(y)
```

Blocks, indentations and control structures

Python allows for conditional expressions, similar to C construct

```
cond ? exp1 : exp2
```

The syntax in Python is

```
exp1 if cond else exp2
```

```
>>> a = True
>>> x = 1 if a else 2
>>> print(x)
1
>>> y = 1 if not a else 2
>>> print(y)
2
```

Blocks, indentations and control structures

Loops: `while` and `for`

The `while` construct behaves like in C, repeating a block of code as long as the condition is true

```
>>> x = 3

>>> while x < 6:
...     print(x)
...     x = x+1
...
3
4
5
```

Blocks, indentations and control structures

The `for` construct behaves differently — it allows iterating over the elements of a sequence (e.g. string, list, etc.), following the order they have in the sequence

To avoid unintended behavior, it is better to avoid modifying the sequence while iterating

```
>>> for x in [1,2,3,'hello']:
...     print(x)
...
1
2
3
hello
```

Blocks, indentations and control structures

The `range` function can be used to generate a sequence of numbers on the fly (note: range does not create a list, but rather an iterator object that can be iterated)

```
>>> for x in range(3):
...     print(x)
...
0
1
2
```

Blocks, indentations and control structures

The `range` function allows defining also an initial value and a step (both are optional; if two values are passed they are assumed to be first and ending element of the range — remember that the ending is always excluded)

```
>>> for x in range(1, 6, 2):
...     print(x)
...
1
3
5

>>> print(list(range(1,6)))

[1,2,3,4,5]
```

Blocks, indentations and control structures

`break` and `continue` statements can be used to exit from the innermost cycle or to skip to the next iteration. The semantic is the same as in C

```
>>> for x in range(10):
...     print(x)
...     if x >= 3:
...         break
...
0
1
2
3
```

Blocks, indentations and control structures

`break` and `continue` statements can be used to exit from the innermost cycle or to skip to the next iteration. The semantic is the same as in C

```
>>> for x in range(10):
...     if x >= 3 and x <= 8:
...         continue
...     print(x)
...
0
1
2
9
10
```

Blocks, indentations and control structures

Loop statements allow for an `else` clause as well

We won't make use of it — if you're interested you can check the Python documentation

Functions

Python allows defining functions with the `def` keyword, followed by the function name and the list of formal parameters (in parenthesis)

```
>>> def f(x):
...     return x + 1
...
>>> f(2)
3
```

A `return` statement is used to make the function return a value

Functions with no `return` statement still return a default value when they reach the end of the function block. The value returned is the special value `None`

Functions

When a function is executed, a local symbol table is created to store local variables

Variables that are assigned inside the function are stored in the local symbol table

Referenced variables are looked-up in the local symbol table, then in the local tables of enclosing functions, and finally in the global symbol table

In general, it's not possible to directly assign to a global variable inside a function (there are exceptions, but we will avoid using them to keep things simpler)

Function arguments are passed by value

```
>>> a = 3
>>> def f(x):
...     x = 4
...
>>> print(a)
3
>>> f(a)
>>> print(a)
3
```

However, the value that is copied is the object reference (similar to passing pointers in C):

```
>>> l = [2,3]
>>> def f(x):
...     x.append(4)
...
>>> print(l)
[2,3]
>>> f(l)
>>> print(l)
[2,3,4]
```

Here, we used method `append` to modify the list object whose reference is the value of `l`. Inside the function, `x` is a copy of the list reference. However, it refers to the same list.

Functions

Python functions allow for both positional and keyword arguments (arguments with default values):

```
>>> def f(x=4):
...     print(x)
...
>>> f()
4
>>> f(2)
2
>>> f(x=3)
3
```

Functions

Positional arguments must be specified when invoking the function, keyword arguments can be omitted (and will use the default value)

```
>>> def f(a, b=0):
...     print(a+b)
...
>>> f(1)
1
>>> f(1, b=4)
5
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() missing 1 required positional argument
    : 'a'
```

Functions

When invoking the function, positional arguments may be passed via keywords, and optional arguments may be passed as positional (following the order in which they are defined)

```
>>> f(1,2)
3
>>> f(a=1,b=2)
3
```

Positional arguments must be placed before keyword ones

```
>>> f(1,2)
3
>>> f(b=2,1)
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

Functions

Arguments can be restricted to be positional only or keyword only — if you're interested, check the tutorial

Python also allows for arbitrary argument lists and arbitrary keyword only argument dictionaries

```
>>> def f(*a):
...     print(a)
...
>>> f(1,2,3)
(1,2,3)
```

In this case, the arguments are packed in a tuple (see below) and passed as value for the parameter `a`

Functions

Arguments can be restricted to be positional only or keyword only — if you're interested, check the tutorial

Python also allows for arbitrary argument lists and arbitrary keyword only argument dictionaries

```
>>> def f(*a, **kw):
...     print(kw)
...
>>> f(a=1,b=2)
{'b': 2, 'a': 1}
```

In this case, the keyword arguments are packed in a dictionary (see below) and passed as value for the parameter `kw`

Functions

In python functions are objects as well

They can be assigned and passed to functions

```
>>> def fpr(x):
...     print(x)
...
>>> def g(f, a):
...     f(a+1)
...
>>> g(fpr, 1)
2
```

Functions

Anonymous functions can be created on the fly through the keyword `lambda`

The body of the function can only be an expression, and is the value returned by the function

```
>>> (lambda x: x+1)(3)
4

>>> f = lambda x: x+1
>>> f(0)
1
```

Functions

Functions can be defined in nested scopes

The nested function can access, but not modify (unless explicitly indicated) the variables of the enclosing function

```
>>> def f(x):
...     def g(y):
...         return x+y
...     print g(1)
...
>>> f(3)
4
```

Functions

This can be useful to create functions with “pre-set” parameters. In the example, we define a generic “add_n” function:

```
>>> def add_n(n):
...     def add(x):
...         return x+n
...     return add
...
>>> add_3 = add_n(3)
>>> add_3(2)
5
```

Lists

Python implements high-level data types

We have already introduced lists – you can refer to the Python manuals for the complete set of list methods

Python allows for concise syntax to create new lists: list comprehension

This construct allows creating a list by applying some operations on the elements of another list or iterable object

```
>>> l = [i**2 for i in range(4)]
>>> print(l)
[0, 1, 4, 9]
```

List comprehension

Filters can be applied to select only a set of elements

```
>>> [i**2 for i in range(10) if i % 2 == 0]
[0, 4, 16, 36, 64]
```

No `else` branch should be provided, only elements for which the condition is True will be selected

Note the difference with conditional expressions

```
>>> [i**2 if i % 2 else 0 for i in range(10)]
[0, 1, 0, 9, 0, 25, 0, 49, 0, 81]
```

Tuples

Besides lists, Python supports another sequence type: tuples

A tuple is a sequence of values separated by commas

```
>>> t = 1,2,3,'A'
>>> print(t)
(1, 2, 3, 'A')
```

Although in some case it's not necessary, to ensure the proper operation order, it's better to enclose the values in parentheses

```
>>> t = (1,2,3,'A')
>>> print(t)
(1, 2, 3, 'A')
```

Tuples

Tuples are immutable (their values cannot be changed), but can be sliced, indexed and iterated in the same way as lists

The empty tuple is represented by a set of empty brackets `()`

Pay attention: `x = 1` is different from `x = 1,`

Python allows also for tuple unpacking — the elements of a tuple can be assigned to a sequence of variables directly

```
>>> x, y, z = (1,2,3)
>>> print(x)
1
```

Tuples

Tuples packing and unpacking is useful, for example, to return more than a single value from a function

```
>>> def f():
...     return 1, 2
...
>>> a, b = f()
```

Tuples

Another interesting use case is iteration of tuples of values

```
>>> l = [(1,'a'), (2, 'b'), (3, 'c')]
>>> for val, name in l:
...     print(val, name)
...
1 a
2 b
3 c
```

Tuples

The `zip` function can be used to create tuples of values from a set of lists (`zip` actually returns an *iterator*, i.e. an object that we can iterate to get values from, but can be iterated only once — we use the `list` function to create a real list in the example)

```
>>> l1 = [1,2,3]
>>> l2 = ['a', 'b', 'c']
>>> l = list(zip(l1, l2))
>>> print(l)
[(1, 'a'), (2, 'b'), (3, 'c')]
```

Tuples

The `zip` function can be used to create tuples of values from a set of lists (`zip` actually returns an iterator, i.e. an object that we can iterate to get values from)

```
>>> l1 = [1,2,3]
>>> l2 = ['a', 'b', 'c']
>>> for val, name in zip(l1, l2):
...     print(val, name)
...
1 a
2 b
3 c
```

Tuples

The function `enumerate` returns an iterator that provides pairs of `(index, value)` from a given iterable

```
>>> l2 = ['a', 'b', 'c']
>>> for val, name in enumerate(l2):
...     print(val, name)
...
0 a
1 b
2 c
```

NOTE: Iterators like those returned by zip, range and enumerate are consumed once they are used, if we need to iterate the result multiple times use `list` to convert the result to a list

Sets

Sets are unordered collections of items without duplicates

Sets are usually employed for testing membership and removing duplicate entries from a collection

Python allows creating sets using a specific notation or the built-in class `set`, which receives an iterable object:

```
>>> a = {1,2,3}
>>> print(a)
set([1, 2, 3])

>>> b = set([1,2,3])
>>> print(b)
set([1, 2, 3])
```

Sets

Note: the object `{}` is not the empty set, but the empty dictionary. To create an empty set, use `set()`

To test set membership, we can use the operator `in`:

```
>>> a = {1,2,3}
>>> 1 in a
True
```

The `in` operator can be used also for lists and tuples, but the look-up is much slower

Sets

Python sets implement standard set operations (union, intersection, ...)

```
>>> a = {1,2,3}
>>> b = {2,3,4}
>>> a.intersection(b)
set([2, 3])
```

As with lists, we have set comprehensions as well:

```
>>> {i//2 for i in range(10)}
set([0, 1, 2, 3, 4])
```

Dictionaries

Python includes a built-in dictionary (`dict`) type

Dictionaries are objects that allow associating keys to values

Think of dicts as sets of pairs *key:value*, where keys are unique

Keys are required to be immutable (and hashable)

Dictionaries

We can define dictionaries explicitly from key:value pairs

```
>>> a = {'A': 1, 'B': 2, 'C': 3}
>>> print(a)
{'A': 1, 'C': 3, 'B': 2}
```

We can look-up values in the dict:

```
>>> a = {'A': 1, 'B': 2, 'C': 3}
>>> print(a['A'])
1
```

Dictionaries

The empty dict is defined as `{}`

We can assign values to keys (overwriting if the key was present):

```
>>> a = {}
>>> a['A'] = 1
>>> a['B'] = 2
>>> print(a)
{'A': 1, 'B': 2}

>>> a['A'] = 3
>>> print(a)
{'A': 3, 'B': 2}
```

Dictionaries

The dictionary can be iterated, the iteration runs over the dictionary keys

No guarantees on the order in which keys are visited (this was recently changed in Python 3.7) — sort the keys (builtin-in function `sorted` or use an OrderedDict from the collection module)

```
>>> a = {'A': 1, 'B': 2, 'C': 3}
>>> print list(a)
['A', 'C', 'B']

>>> [a[i] for i in sorted(a)]
[1,2,3]
```

Dictionaries

The collection of keys can be recovered also using the `.keys` method

The collection of values can be recovered using the `.values` method

Elements can be removed using the `del` keyword:

```
>>> a = {'A': 1, 'B': 2, 'C': 3}
>>> del a['A']
>>> print(a)
{'C': 3, 'B': 2}
```

Keyword `del` can also be used to remove elements from lists, or to delete variables

Dictionaries

Dictionaries can be created using dict comprehensions:

```
>>> {i: i**2 for i in range(5)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

The `dict` construct allows creating dictionaries from sequences of key-value pairs

```
>>> dict([('A', 1), ('B', 2)])
{'A': 1, 'B': 2}
```

Importing modules

Python allows using functionalities defined in different files

Each file is called a module, and modules can be imported in other modules, or in an interactive session

Importing a module allows accessing functions and variables defined in the file

File fact.py

```
def fact_func(n):
    if n == 0:
        return 1
    return n*fact_func(n-1)
```

File script.py

```
import fact
print(fact.fact_func(4))
```

```
$> python script.py
24
```

Importing modules

The `import` statement allows importing a module

Remember that the imported file name extension should be `.py`, but the extension should not be specified when importing the module

The symbols defined in the module are placed in a private symbol table associated to the module, and can be accessed through the dot `.` operator

Importing modules

When a module is imported, it is actually executed (multiple imports are allowed, but the module is executed only the first time)

File mod.py

```
a = 3
b = a + 4
print ('Hello world')
```

File script.py

```
import mod
print(mod.b)
```

```
$> python script.py
Hello world
7
```

Importing modules

Inside a module, its own name is accessible through the variable `__name__`

The main module (the module we pass to the interpreter at launch) has the special name `'__main__'`

This can be used to execute parts of a script only when it's the main module but not when it's imported

Importing modules

File mod.py

```
a = 3
b = a + 4
if __name__ == '__main__':
    print ('Hello world')
```

File script.py

```
import mod
if __name__ == '__main__':
    print(mod.b)
```

```
$> python script.py
7
```

```
$> python mod.py
Hello world
```

Importing modules

Assigning a different name to a module: use the `as` keyword

```
>>> import mod as mymodule
>>> print(mymodule.b)
7
```

We can import module symbols in the current local symbol table:

```
>>> from mod import b
>>> print(b)
7
```

We can import all module symbols in the local symbol table with `import *` — Pay attention that you do not accidentally overwrite a local variable or function (suggestion: avoid `import *`)

Importing modules

Libraries can be treated as modules (they are in general packages, but we won't go into details)

Library modules sometimes have sub-modules. In some cases these need to be imported as well before their content is accessible (but this depends on the library implementation)

We will see some examples when introducing the numerical libraries

Input / output

Files can be accessed using the function `open`

```
>>> f = open('filename', 'r')
```

The first argument is the filename. The second argument is the *mode*. Text modes are similar to C: `'r'`, `'w'`, `'a'`, `'r+'`.

Binary encoding is represented by `'b'` appended to the mode. In this case, I/O operations do not return strings, but byte sequences (we won't use explicitly the binary encoding)

Input / output

To avoid data loss on writing, files should be always closed when I/O is finished

We can let the file object handle this automatically using the with keyword

```
>>> with open('filename', 'r') as f:
...     #do something with f here
```

The with statement takes care of closing the file when the block is finished (this is done by automatically calling some methods of the file object when the with statement starts and ends)

Input / output

To read an entire file, we can use the method `read(size)`

`size` is an optional parameter specifying the maximum number of characters or bytes to read.

file tmp.txt

```
This is a file
with multiple lines
```

script.py

```
f=open('tmp.txt')
print(f.read())
```

```
$> python script.py
This is a file
with multiple lines
```

Input / output

We can read a file line by line using the `readline` method. An empty string denotes that the end of file has been reached

```
>>> f = open('tmp.txt', 'r')
>>> f.readline()
'This is a file\n'
>>> f.readline()
'with multiple lines\n'
>>> f.readline()
''
```

Input / output

A file object can be iterated, and can thus be used in a `for` loop. The file is read line by line (remember, lines can end with `'\n'`)

```
>>> f = open('tmp.txt', 'r')
>>> for line in f:
...     print(line.rstrip())
...
This is a file
with multiple lines
```

Input / output

To write to a file we can use the method `write`

```
>>> f = open('out.txt', 'w')
>>> f.write('Hello')
>>> f.close()
```

We can also use print, providing an argument for parameter `file`

```
>>> f = open('out.txt', 'w')
>>> print('Hello', file=f)
>>> f.close()
```

Note: `print` adds a newline at the end of the string, unless we pass `end=''`

Serialization

Complex data can be serialized (i.e. transformed to a string that can be written to a file) in different ways

Using `json` (more portable, less powerful):

```
>>> import json
>>> l=[1, ('Hello', 'world'), 3]
>>> f=open('out', 'w')
>>> json.dump(l, f)
>>> f.close()

>>> f = open('out', 'r')
>>> json.load(f)
[1, ['Hello', 'world'], 3]
```

Serialization

Complex data can be serialized (i.e. transformed to a string that can be written to a file) in different ways

Using Python `pickle` (less portable, supports more complex data):

```
>>> import pickle
>>> l=[1, ('Hello', 'world'), 3]
>>> f=open('out', 'wb') # Binary mode
>>> pickle.dump(l, f)
>>> f.close()

>>> f = open('out', 'rb')
>>> pickle.load(f)
[1, ('Hello', 'world'), 3]
```

String formatting

Python supports C-style string interpolation. The syntax is `'string' % values`, where values is either a single value or a tuple of values

```
>>> '%d is %s than %d' % (2, 'lower', 3)
'2 is lower than 3'
```

Format specifiers are similar to those of C `printf`

Python supports more advanced string formatting, you can refer to the tutorial

Strings

Some other useful string methods (they return new strings, check the documentation for the explanations of their parameters):

- `lower`, `upper`: convert a string to lower or upper case
- `lstrip`, `rstrip`, `strip`: remove leading / trailing / both whitespaces from a string
- `replace`: replace occurrencies of text
- `split`: returns a list of words of a given string, separated using a second string as delimiter

Exceptions

Python allows handling errors at execution time (exceptions)

```
>>> try:
...     y = 'a' + 3
... except:
...     y = 4
...
>>> print(y)
4
```

Exceptions

We can specify which exception should be handled by each `except` block

```
>>> try:
...     y = 'a' + 3
... except ValueError:
...     y = 4
... except TypeError:
...     y = 5
...
>>> print(y)
5
```

Exceptions

We can specify code to be executed regardless of whether the `try` block incurs in an exception

```
>>> try:
...     y = 'a' + 3
... except:
...     pass
... finally:
...     y = 3
...
>>> print(y)
3
```

```
>>> try:
...     y = 2
... except:
...     pass
... finally:
...     y = 3
...
>>> print(y)
3
```

Exceptions

We can raise an exception through the `raise` keyword

```
>>> raise TypeError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError
```

More on namespaces

We have seen the concept of namespaces when discussing modules and nested functions

In Python, a *namespace* is a mapping from names (variables, functions, ...) to objects

Examples of namespace include

- The set of built-in names (`sum`, `print`, ...)
- The global variables of a module
- The local variables of a function

Names in different namespace are independent

More on namespaces

Scopes define which names can be directly accessed

Scopes are searched in the following order

- Innermost scope, containing the local names
- Scopes of enclosing functions (starting from the innermost)
- The current module, containing the module global names
- The outermost scope, containing the built-in names

More on namespaces

We can access names in a different namespace through the dot `.` operator

We have already seen this when accessing methods of built-in types, or variables in a module

In general, we refer to the element after the dot as an attribute of the object on the left

Attributes can be read-only or writable

```
>>> import mymodule
>>> mymodule.newvar = 1
>>> print(mymodule.newvar)
1
```

Classes and objects

Python classes and objects allow writing code in an object-oriented way

To define a class, we use the `class` keyword, followed by a block of statements

These statements are executed in a local namespace associated to the class

Note: classes are themselves objects

We can access class attributes with the dot `.` operator

Classes and objects

script.py

```
class AClass:
    x = 3
    def f(self):
        print('Hello world')

if __name__ == '__main__':
    print(AClass.x)
    print(AClass.f)
```

```
$> python script.py

3
<function AClass.f at 0x7f2e4eab3ea0>
```

Classes and objects

Class instances (called objects) can be created by “calling” the class object

script2.py

```
from script import AClass
obj = AClass()
print(obj.x)
print(obj.f)
obj.f()
```

```
$> python script2.py

3
<bound method AClass.f of <script.AClass object at 0
    x7f0db0bb7ef0>>
Hello world
```

Classes and objects

The instantiated object allows for attribute look-up. Attributes can be either data attributes (similar to variables) or methods

Similar to variables, data attributes are not declared, but are created on assignment

script3.py

```
from script import AClass
obj = AClass()
obj.z = 1
print(obj.z)
```

```
$> python script3.py
1
```

Classes and objects

script4.py

```
from script import AClass
obj = AClass()
obj.f()
AClass.f(obj)
```

```
$> python script4.py

Hello world
Hello world
```

Classes and objects

When a class is instantiated, its special method `__init__` is invoked (if not present, it's inherited from the built-in `'object'` class)

We can use the method to initialize the object attributes

```
class Point:
    name = '2D Point'
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Note: `x` and `y` are *object* attributes that are created on the new object when it's instantiated

Class attributes (e.g. `name`) are shared by all instances

Classes and objects

Class attributes are used in place of object attributes whenever a given attribute is not found on an object

```
class A:
    x  = 3
    def __init__(self, y):
        self.y = y

>>> a = A(1)
>>> print(a.x)
3
```

NOTE: class attributes are shared among all instances of the class, but can be masked by object attributes

Classes and objects

```
class A:
    x  = 3

>>> a = A()
>>> b = A()
>>> print(a.x, b.x)
3 3
>>> a.x = 4
>>> print(a.x, b.x)
4 3
```

In this case, `a.x = 4` creates a *new* attribute called `x` on the object `a`

Classes and objects

```
class A:
    x  = [3]

>>> a = A()
>>> b = A()
>>> print(a.x, b.x)
[3] [3]
>>> a.x.append(4)
>>> print(a.x, b.x)
[3, 4] [3, 4]
```

In this case, `a.x` is not assigned, but is accessed. Since `a` has no attribute `x`, class A is searched, and its `x` attribute is used. We then call a method of the object `A.x`. Its modifications are visible to all objects that do not specify a local attribute `x`

Classes and objects

Python classes can define *special* (or *magic*) methods

These methods, whose names start and end with a double underscore, allow defining how the object should behave in particular cases, such as when it appears in an expression

For example, the method `__add__` allows defining how to compute the result of a sum (`+`) of two objects

Classes and objects

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Point(self.x + other.x, self.y +
            other.y)

>>> p1 = Point(1,2)
>>> p2 = Point(5,5)
>>> p3 = p1 + p2
>>> print(p3.x, p3.y)
6, 7
```

Other methods allow overriding different operators (arithmetic, logical, slicing, getting and setting items, ...)

For example, the `__str__` method defines how an object is converted to a string

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return 'Point(%.2f,%.2f)' % (self.x, self.y)

>>> p = Point(1,2)
>>> print(p)
Point(1.00,2.00)
```

Classes and objects

Python allows a class to inherit from a different one

The syntax requires specifying a comma-separated sequence of base classes

```
class DerivedClass(Base1, Base2, Base3):
    <statements>
```

Attribute look-up becomes more complex

For this course, we will mainly use classes and objects to simulate C-style structures, i.e. as containers of data

If you're interested, you can learn more on Python objected oriented programming on the Python tutorial and on the Python documentation

Command line arguments

Command line arguments can be accessed from the attribute `argv` of module sys

`argv` is a list of strings, each element corresponds to one argument

Command line arguments

script.py

```
import sys
print(len(sys.argv))
for arg in sys.args:
    print(arg)
```

```
$> python myscript.py 1 aaa 42

4
script.py
1
aaa
42
```