Python Training

Aresys internal course

A. Valentino

June 2016

Outline

Outline

Language features

- General-purpose programming language
  - **like** C, C++ and Java, **differently** from Matlab
  - Scripting, GUI, Web, scientific programming, ...
- Multi paradigm
  - **like** C++, **differently** from Java
  - OOP, procedural, and also a little bit functional
- Interpreted
  - **like** Java, **differently** form C, C++ and Fortran
  - you never have to worry about byte-code compilation
- Focus on readability and productivity

Typing

- Dynamically typed
  - **like** Matlab, **differently** from C, C++ and Java
- Strongly typed
  - **like** C, C++ and Java, **differently** from Tcl/Tk and Perl

Typing: examples

Perl

```
DB<1> print "2" + 2
4
```

Python

```
>>> "2" + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str
implicitly
```

Language versions

- Python 2
  - latest version: 2.7.11
  - maintenance mode
  - end of life: 2020
- Python 3
  - latest version: 3.5.1
  - language cleanup
  - drop a lot of deprecated features
  - **breaks compatibility** with Python 2 (e.g. `print`)
  - ... but not too much if you import from `__future__`
  - in this presentation we will talk about Python 3

Importing from `__future__`

```
from __future__ import print_function
```

Implementations

- CPython (the reference and most widely used implementation written in C)
  - https://www.python.org
- PyPy (A fast python implementation with a JIT compiler)
  - http://pypy.org
  - compatibility: Python 2.7.10 and Python 3.2.5 (beta)
- IronPython (Python running on .NET)
  - http://ironpython.net
  - support for Python 2.7
- Jython (Python running on the Java Virtual Machine)
  - http://www.jython.org
  - support for Python 2.7

Distributions

- https://www.python.org: Official CPython distribution (for Python 2 and 3), interpreter and standard library (about 180 modules/packages)
- https://www.continuum.io/why-anaconda: Anaconda (Python 2 and 3 on Linux/Windows/Mac):
  - includes a package manager and a huge collection of packages
  - allows to manage different environments simultaneously
  - plays well with binary extensions
  - particularly suited for scientific users
- https://www.enthought.com/products/canopy: Enthought Canopy (Python 2 only)
- https://python-xy.github.io: *Pyhton*(x,y) (Python 2 and windows only)
- Any GNU/Linux distribution

Outline

Hello, World!

Python

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

Java

```
class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, world!");
    }
}
```

Interpreter invocation

Run a script

```
$ python3 script.py
```

Interactive mode

```
$ python3
Python 3.5.1+ (Mar 30 2016, 22:46:26) on linux
Type "help", "copyright", "credits" or "license" for
more information.

>>> import os
>>> os.path.getsize('somefile.txt')
20366
>>> _
```

Improved interactive shell: *Jupyter* (a.k.a. *IPython*)

The *Jupyter* (Qt)Console and the (web based) *notebook* provide:

- better auto-completion
- enhanced history management
- session profiles
- integration with the system shell
- integration with GUI envs and scientific tool-chains

Links

- Jupyter Project: http://jupyter.org
- IPython: http://ipython.org
- On-line *Jupyter* notebook: https://try.jupyter.org
- On-line interactive shell: https://www.python.org/shell

Getting help

help builtin function

- `help()`: start an interactive help session
- `help(soemthing)`: print help about an object or type
- `help('name')`: search for "name" in built-in functions and types and print the relative help

help example

```
>>> help(str)
Help on class str in module builtins:

class str(object)
|  str(object='') -> str
|  str(bytes_or_buffer[, encoding[, errors]]) -> str
|  ...
```

Getting help in IPython

```
In [1]: str?
Type:              type
String form:       <class 'str'>
Namespace:         Python builtin
Init definition:   str(self, *args, **kwargs)
Docstring:
str(object='') -> str
str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object.
If encoding or errors is specified, then the object must
expose a data buffer that will be decoded using the given
encoding and error handler.
Otherwise, returns the result of object.__str__()
(if defined) or repr(object).
...
```

Documentation string a.k.a. *docstrings*

Docstring example

```
def func(args):
    '''Short descr.

    Long description

    '''

    pass
```

- Docstrings are used by the `help` function to display object help
- the **first string** in the body of
  - functions/methods
  - types (classes)
  - modules

  is used as *docstring*
- Docstrings are *properties* of the object (`__doc__` special attribute)

Note

Strings enclosed in *triple quotes* (`"""` or `'''`) *are multi-line* strings

Introspection

```
In [1]: s = 'ciao'

In [2]: len(s)
Out[2]: 4

In [3]: type(s)
Out[3]: str

In [4]: type(2)
Out[4]: int

In [5]: isinstance(s, str)
Out[5]: True

In [6]: hasattr(s, 'upper')
Out[6]: True
```

```
In [6]: T = type(s)

In [7]: two = T(2)

In [8]: two
Out[8]: '2'

In [9]: dir(s)
Out[9]: ['__add__',
'__class__',
...
'swapcase',
'title',
'translate',
'upper',
'zfill']
```

All is an object

- In Python all is an **object**
- No *primitive* type: numbers, strings, containers are all **objects**
- Also functions, types and modules are **objects**

```
>>> def func(): pass

>>> type(func)
function

>>> isinstance(func, object)
True

>>> dir(func)
['__annotations__',
...
```

```
>>> type('ciao')
str

>>> type(str)
type

>>> isinstance(str, object)
True

>>> dir(str)
['__add__', ...
```

Indentation

Important

- white spaces are **syntactically significant**
- indentation is used to determine the grouping of statements

Hint

- never mix tabs and spaces for indentation
- always use 4 (four) spaces to indent your code (it is the most widely adopted coding convention in the Python ecosystem)

Pretty file size

```
def pretty_file_size_str(size):
    '''String representation of file size

    Get the size in bytes and return the string
    representation using appropriate units.
    '''

    if size >= 2**20:
        size /= 2**20
        units = 'MB'
    elif size >= 2**10:
        size /= 2**10
        units = 'KB'
    else:
        units = 'Bytes'

    return '%.3f %s' % (size, units)
```

Let's use our function

```
In [1]: import os
In [2]: from prettysize import pretty_file_size_str

In [3]: size = os.path.getsize('somefile.txt')
In [4]: size
Out[4]: 20366

In [5]: pretty_file_size_str(size)
Out[5]: 19.889 KB
```

C++ variables are *Boxes*

```
int a = 1;
```


```
a = 2;
```

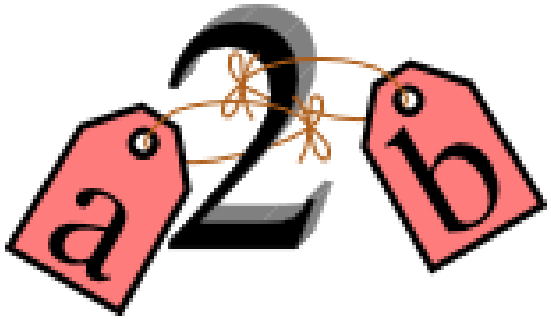

```
int b = a;
```

Python variables are *Labels*

```
In [1]: a = 1
```


```
In [2]: a = 2
```


```
In [3]: b = a
```



Summary

- indentation
- introspection
- all is an object
- variables
- how to import a module
- function definition
- string formatting
- docstrings

- built-ins functions and types:
  - `print`
  - `dir`
  - `hasattr`
  - `help`
  - `len`
  - `str`
- keywords and statements
  - `if elif else`
  - `def`
  - `return`
  - `import`

Outline

Overview

- Immutable types:
  - numbers (int, float, complex)
  - strings
  - tuples (heterogeneous sequences)
  - frozensets (immutable version of set)
  - `None` (the only value of the type `NoneType`, singleton, it is used to represent the absence of a value)
  - booleans (`True`, `False`)
- Mutable types
  - lists (heterogeneous sequences)
  - dictionaries or mapping types (associative arrays implemented as hash-maps: $\mathcal{O}(1)$ access)
  - sets (unordered collection of heterogeneous object with no duplication)
  - many others ...

Immutable objects

- Immutable objects are object with a fixed value
- Such objects cannot be altered
- A new object has to be created if a different value has to be stored
- Immutable objects include numbers, strings, tuples, frozensets and `None`
- They have an important property: they are *hashable* (have a constant hash)
- *Hashable* objects can be used as keys in a dictionary (associative arrays implemented as hash maps)

Numeric types

```
In [1]: type(5)
Out[1]: int

In [2]: type(2.3)      # corresponds to the C double
Out[2]: float

In [3]: type(1 + 3j)
Out[3]: complex

In [4]: c = 1 + 3j     # immutable: c.imag = 0 --> ERROR
In [5]: type(c.real)
Out[5]: float

In [6]: big = 2**65    # pow(2, 65) does not fit in a C long
In [7]: type(big), big
Out[7]: (int, 36893488147419103232)
```

Numeric casts and conversion

Casting

```
In [8]: int(2.1)          # cast
Out[8]: 2

In [9]: complex(2)        # cast
Out[9]: (2+0j)

In [10]: float(1)         # cast
Out[10]: 1.0
```

Conversion from string

```
In [11]: float('1.3e-2')   # conversion from string
Out[11]: 0.013
```

Operators

```
In [12]: 2 + 2      # sum
Out[12]: 4

In [13]: 2 - 2      # subtraction
Out[13]: 0

In [14]: 3 * 3      # product
Out[14]: 9

In [15]: 3 ** 2     # pow
Out[15]: 9

In [16]: n = 1
In [17]: n *= 2.    # inplace product
In [18]: n
Out[18]: 2.0        # NOTE: type propotion (n = n * 2)
```

Division operators

```
In [19]: 3 / 2      # division
Out[19]: 1.5        # IMPORTANT: always "true" division
                    # (always float)
In [20]: -3. // 2   # floor division
Out[20]: -2.0       # NOTE: type is preserved

In [21]: 3. % 2     # reminder
Out[21]: 1.         # NOTE: type is preserved
```

Important

The behavior of the / (division) is changed in Python 3.
Previously it was not a "true division" if both operands was integers.
Please always use:

```
from __future__ import division
```

to get the *correct* behavior also in Python 2

Comparison operators

```
In [22]: 2 == 2.
Out[22]: True

In [23]: 3 != 2
Out[23]: True

In [24]: 3. >= 2          # >, <, >=, <=
Out[24]: True

In [25]: a = 0.5
In [26]: 0 <= a < 1       # Very nice: a in [0,1[
Out[26]: True

In [27]: 1j > 2.
TypeError                 Traceback (most recent call last)
[...]
TypeError: unorderable types: complex() > int()
```

Strings

- String are **immutable sequences** of characters
  - `text.replace('\t', '␣␣␣␣')` –> a **new string** is created
- There are two kind of strings:
  - the `str` type: are strings of `unicode` characters
  - the `bytes` type: are string of 8-bit characters
  - conversion between string types requires encoding/decoding
    - `str.encode('ascii') --> bytes`
    - `bytes.decode('UTF-8') --> str`

Backward incompatible change in Python 3

- (Python 2) bytes == (Python 3) bytes
- (Python 2) bytes == (Python 2) str == (Python 3) bytes
- (Python 2) unicode == (Python 3) str

String literals

```
In [1]: ('ciao' == "ciao") and ('ciao' == u'ciao')
Out[1]: True

In [2]: type('ciao'), type(b'ciao')     # str vs bytes
Out[2]: (str, bytes)

In [3]: r'Hello,\nWorld!'     # raw strings
Out[3]: 'Hello,\\nWorld!'

In [4]: s = '''Hello,
World!'''

In [5]: s     # --> repr(s)
Out[5]: 'Hello,\nWorld!'

In [6]: print(s)
Hello,
World!
```

Main string methods

- isalnum
- isalpha
- isdecimal
- isdigit
- isidentifier
- islower
- isnumeric
- isprintable
- isspace
- istitle
- isupper
- startswith
- endswith
- join
- split
- rsplit
- splitlines
- strip
- lstrip
- rstrip
- partition
- rpartition
- lower
- upper
- capitalize
- casefold
- title
- swapcase
- center
- ljust
- rjust
- zfill
- expandtabs
- count
- find
- rfind
- replace
- translate
- index
- rindex
- format

Other string tools

```
In [1]: len('ciao')
Out[2]: 4

In [2]: 'World' in 'Hello, World!'
Out[2]: True

In [3]: 'Hello, ' + 'World!'
Out[3]: 'Hello, World!'

In [4]: 'ciao' * 2
Out[4]: 'ciaociao'

In [5]: 'aaa' < 'bbb'
Out[5]: True

In [6]: bool(''), bool('a')
Out[6]: False, True
```

Indexing

```
In [7]: s = 'abcdefghijklmnopqrstuvwxyz'
In [8]: s[2]                    # zero based indexing
Out[8]: 'c'

In [9]: s[0:2]                  # start included, stop excluded
Out[9]: 'ab'

In [10]: s[:13], s[13:]         # no overlap & start/stop omitted
Out[10]: ('abcdefghijklm', 'nopqrstuvwxyz')

In [11]: s[-1]                  # negative indices are OK
Out[11]: 'z'

In [12]: s[1:20:3]              # start:stop:step
Out[12]: 'behknqt'

In [12]: s[::-1]                # negative step is OK --> revese seq.
Out[12]: 'zyxwvutsrqponmlkjihgfedcba'

In [13]: s[:]                   # all elements
```

Printf-style string formatting

```
In [1]: print('Hello, %s!' % 'World')    # %s --> str
Hello, World!

In [2]: print('%d / %d = %.2f' % (1, 3, 1/3))
1 / 3 = 0.33

In [3]: print('"%+10.2f"' % (1/3))
"     +0.33"

In [4]: mapping = {'name': 'Python', 'version': 3}
In [5]: print('%(name)s v%(version)d' % mapping)
Python v3
```

References

https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting

String formatting with *.format()*

```
In [1]: print('Hello, {}!'.format('World'))
Hello, World!

In [2]: print('{} / {} = {:.2f}'.format(1, 3, 1/3))
1 / 3 = 0.33

In [3]: print('{name} v{version:d}'.format(
   ...: name='Python', version=3))
Python v3

In [4]: print('{1}{0}{0}'.format('+', 'C'))
C++
```

References

https://docs.python.org/3/library/string.html#format-string-syntax

String anti-pattern

Bad

```
In [1]: notes = ('do', 're', 'mi')
In [2]: solfege = notes[0]
   ...: for note in notes[1:]:
   ...:     solfege += ', ' + note
In [3]: solfege
Out[3]: 'do, re, mi'
```

Good

```
In [1]: notes = ('do', 're', 'mi')
In [2]: solfege = ', '.join(notes)
In [3]: solfege
Out[3]: 'do, re, mi'
```

tuple

- Tuples are **immutable sequences** of zero or more elements
- You can consider tuples as **heterogeneous** containers (can store elements of different type)
- A tuple consists of a number of values separated by commas
- Parentheses are used for correct interpretation and nesting

tuple literals

```
In [1]: t = 'a', 'b', 3
In [2]: t = ('a', 'b', 3)
In [3]: t = ()                          # empty tuple
In [4]: t = ('aa',)                     # one element tuple
                                        # note the final comma
In [5]: t = ('a', (1, 2), 'b')          # nesting
In [6]: t[1]
Out[6]: (1, 2)
```

tuple constructor

```
In [7]: t = tuple(('a', 'b', 3))

In [8]: t = tuple(t)                    # tuple(sequence)

In [9]: t = tuple(['a','b', 3])         # tuple(sequence)

In [10]: t = tuple()                    # empty

In [11]: t = tuple((2,))                # one element

In [12]: tuple('ciao')                  # tuple(sequence)
Out[12]: ('c', 'i', 'a', 'o')

In [13]: t = ('ciao',)                  # tuple(sequence)
```

tuple methods

- index
- count

Examples

```
In [14]: t = ('a', 'b', 'b', 'c')

In [15]: t.index('c')
Out[15]: 3

In [16]: t.count('b')
Out[16]: 2
```

Other tuple tools (1/2)

```
In [17]: (1, 2) + ('x', 'y')          # concatenation
Out[17]: (1, 2, 'x', 'y')

In [18]: len(t)
Out[18]: 4

In [19]: 2 in t                       # contains
Our[19]: True

In [20]: t[1]                         # indexing
Out[20]: 2

In [21]: t[1:3]                       # slicing
Out[21]: (2, 'x')                     # output is a tuple
```

Other tuple tools (2/2)

```
In [22]: t
Out[22]: (1, 2, 'x', 'y')

In [23]: a, b, c, d = t                    # unpacking
In [24]: a
Out[24]: 1

In [25]: bool(t)
Out[25]: Ture

In [26]: bool(empty_tuple)
Out[26]: False

In [27]: ('a', 2) < ('a', 0)               # lexicographical
Out[27]: False
```

list

- Lists are **mutable sequences** of zero or more elements
- Lists are **heterogeneous** containers (can store elements of different type)

list literals

```
In [1]: L = ['a', 'b', 3]
In [2]: L = []                          # empty list
In [3]: L = ['aa']                      # one element list
                                        # no final comma here!
In [4]: L = ['a', [1, 2], 'b']          # nesting
In [5]: L[1]
Out[5]: [1, 2]
```

list constructor

```
In [6]: L = list(['a', 'b', 3])

In [7]: L = list(L)                    # list(sequence)

In [8]: L = list(['a','b', 3])         # list(sequence)

In [9]: L = list()                     # empty

In [10]: L = list([2])                 # one element

In [11]: list('ciao')                  # list(sequence)
Out[11]: ['c', 'i', 'a', 'o']

In [12]: L = ['ciao']                  # list(sequence)
```

list methods

- index (like tuples)
- count (like tuples)
- copy
- sort
- reverse
- append
- extend
- insert
- pop
- clear
- remove

Note

With the only exception of `list.pop`, all methods that change a `list` in-place return `None`

list method examples

```
In [1]: L = ['a', 'b']

In [2]: L.append(3)
In [3]: L
Out[3]: ['a', 'b', 3]

In [4]: L.insert(1, (1, 2))
In [5]: L
Out[5]: ['a', (1, 2), 'b', 3]

In [6]: L.pop()
Out[6]: 3
In [7]: L
Out[7]: ['a', (1, 2), 'b', 3]
```

```
In [8]: L.pop(1)
Out[8]: (1, 2)
In [9]: L
Out[9]: ['a', 'b']

In [10]: L.extend([1, 2])
In [11]: L
Out[11]: ['a', 'b', 1, 2]

In [12]: L.remove('b')
In [13]: L
Out[13]: ['a', 1, 2]
```

Other list tools (1/3)

```
In [14]: [1, 2] + ['x', 'y']   # concatenation
Out[14]: [1, 2, 'x', 'y']

In [15]: [1, 2] + ('x', 'y')
TypeError: can only concatenate list (not "tuple") to list

In [16]: len(L)
Out[16]: 4

In [17]: 2 in L                # contains
Our[17]: True

In [18]: L[1]                  # indexing
Out[18]: 2

In [19]: L[1:3]                # slicing
Out[19]: (2, 'x')              # output is a tuple
```

Other `list` tools (2/3)

```
In [20]: a, b, c, d = [1, 2, 'x', 'y']   # unpacking
In [21]: b
Out[21]: 2

In [22]: L = [1, 2, 'x', 'y']
In [23]: del L[1]
In [24]: L
Out[24]: [1, 'x', 'y']

In [25]: L = [1, 2, 'x', 'y']
In [26]: del L[1:3]
In [27]: L
Out[27]: [1, 'y']

In [26]: L = [1, 2, 'x', 'y']
In [27]: del L[:]                    # equivalent to L.clear()
In [28]: L
Out[28]: []
```

Other `list` tools (3/3)

```
In [28]: bool(L)
Out[28]: True

In [29]: bool([])
Out[29]: False

In [27]: ['a', 2] < ['a', 0]          # lexicographical
Out[27]: False

In [28]: (1, 2) == [1, 2]             # different types
Out[28]: False

In [29]: (1, 2) == tuple([1, 2])
Out[29]: True
```

dict

- Dictionaries (`dict`) are associative array (mapping types)
- Dictionaries are **heterogeneous** containers (can store elements of different type)
- Unordered (but exists the `collections.OrderedDict` type)
- ... and randomized (can't rely on the key order)
- Implemented as hash-maps
  - Unique keys
  - $\mathcal{O}(1)$ access
- Keys must be *hashable*
  - immutable types are usually OK (numbers, strings, etc.)
  - tuples or frozensets must only contain immutable object

dict constructors and literals

```
In [1]: d = {'a': 1, 'b': 2, 3: 'ciao'}

In [2]: d = {}      # empty dict

In [3]: d = dict(a=1, b=2)

In [4]: # dict(sequence of pairs)
In [5]: d = dict([('a', 1), ('b', 2), (3, 'ciao')])
Out[5]: {3: 'ciao', 'b': 2, 'a': 1}

In [6]: dict.fromkeys(['a','b', 3])
Out[6]: {3: None, 'b': None, 'a': None}

In [7]: dict.fromkeys(['a','b', 3], 'zero')
Out[7]: {3: 'zero', 'b': 'zero', 'a': 'zero'}
```

dict indexing (1/2)

```
In [1]: d = {'a': 1, 'b': 2}

In [2]: d['b']
Out[2]: 2

In [3]: d['unexinsing key']
KeyError: 'unexinsing key'

In [4]: print(d.get('unexinsing key'))
None

In [5]: d[list()] = [1, 2, 3]
TypeError: unhashable type: 'list'
```

dict indexing (2/2)

```
In [6]: d['new key'] = [1, 2, 3]
In [7]: d
Out[7]: {'b': 2, 'a': 1, 'new key': [1, 2, 3]}

In [8]: d['new key'] = None     # repalces existing value
In [9]: d
Out[9]: {'b': 2, 'a': 1, 'new key': None}

In [10]: del d['new key']
In [11]: d
Out[11]: {'b': 2, 'a': 1}
```

dict methods

Python 3

- keys (–> iterator)
- values (–> iterator)
- items (–> iterator)

Python 2

- keys (–> list)
- values (–> list)
- items (–> list)
- iterkeys (–> iterator)
- itervalues (–> iterator)
- iteritems (–> iterator)

- fromkeys (static method)
- copy
- get
- update
- setdefault
- pop
- popitem
- clear

dict method examples (1/2)

```
In [1]: d = {'a': 1, 'b': 2}

In [2]: list(d.keys())
Out[2]: ['b', 'a']

In [3]: list(d.values())
Out[3]: [2, 1]                          # match the order of keys

In [4]: list(d.items())
Out[4]: [('b', 2), ('a', 1)]

In [5]: for k, v in d.items():
   ...:     print('key={!r}, value={!r}'.format(k, v))
key='b', value=2
key='a', value=1
```

dict method examples (2/2)

```
In [6]: d.update({'aa': 11, 'bb': 22})      # --> None
In [7]: d
Out[7]: {'b': 2, 'a': 1, 'bb': 22, 'aa': 11}

# if key not in d:
#       d[key] = default
# return d[key]
In [8]: d.setdefault('new key', 'default value')
Out[8]: 'default value'

In [9]: d.pop('new key')
Out[9]: 'default value'

In [10]: d.popitem()
Out[10]: ('b', 2)

In [11]: d
Out[11]: {'a': 1, 'bb': 22, 'aa': 11}
```

Other dict tools

```
In [12]: len(d)      # d = {'a': 1, 'bb': 22, 'aa': 11}
Out[12]: 3

In [13]: 'a' in d                          # contains
Our[13]: True

In [14]: 'a' not in d                      # contains
Our[14]: False

In [15]: bool(d)
Out[15]: True

In [16]: bool({})
Out[16]: False
```

set (and frozenset)

- A `set` is an unordered collection with no duplicate elements
- Sets are **heterogeneous** containers (can store elements of different type)
- Elements must be *hashable*
  - immutable types are usually OK (numbers, strings, etc.)
  - tuples or frozensets must only contain immutable object
- A `frozenset` is the immutable version of a `set`

set constructors and literals

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

In [2]: type({})                          # WARNING: empty dict
Out[2]: dict

In [3]: s = set()                         # empty set
In [4]: s = set(['a', 'b', 2, 3])         # set(sequence)

In [5]: set(['a', 'b', (2, 3)])           # tuple element
Out[5]: {'b', 'a', (2, 3)}

In [6]: set('ciao')                       # set(sequence)
Out[6]: {'a', 'c', 'i', 'o'}

In [6]: set(['ciao'])                     # one str element
Out[6]: {'ciao'}
```

set tools

```
In [1]: vowels = set('aeiou')

In [2]: len(vowels)
Out[2]: 5

In [3]: 'b' in vowels          # contains
Out[3]: False

In [4]: 'b' not in vowels      # not contains
Our[4]: True

In [5]: bool(s), bool(set())
Out[5]: (True, False)

In [6]: for item in set(['item1', 'item2']):
   ...:     do_something(item)
```

set comparison (1/2)

```
In [7]: vowels == {'a', 'e', 'i', 'o', 'u'}
Out[7]: True

In [8]: vowels != {'a', 'e', 'i', 'o', 'u'}
Out[8]: False

In [9]: vowels <= set(ascii_lowercase) # set.issubset
Out[9]: True

In [10]: vowels < {'a', 'e', 'i', 'o', 'u'}
Out[10]: False

In [11]: vowels >= set(['a', 'e'])      # set.issuperset
Out[11]: True

In [12]: {'a', 'e', 'i', 'o', 'u', 'x'} > vowels
Out[12]: True
```

set comparison (2/2)

```
In [13]: vowels | set([1, 2])     # set.union
Our[13]: {'o', 'a', 1, 2, 'e', 'u', 'i'}

In [14]: vowels & set('abc')      # set.intersection
Out[14]: {'a'}

In [15]: vowels - set('abc')      # set.difference
Out[15]: {'e', 'i', 'o', 'u'}

In [16]: vowels ^ set('abc')      # set.symmetric_difference
Out[16]: {'b', 'c', 'e', 'i', 'o', 'u'}
```

set methods

Both `set` and `frozenset`

- copy
- difference
- intersection
- symmetric_difference
- union
- isdisjoint
- issubset
- issuperset

Modifier methods (`set` only)

- add
- update (like `dict`)
- clear
- discard
- pop
- remove
- intersection_update
- difference_update
- symmetric_difference_update

set method examples (1/2)

```
In [1]: vowels.issubset(ascii_lowercase)          # <=
Out[1]: True
In [2]: vowels.issuperset(['a', 'u'])             # >=
Out[2]: True
In [3]: vowels.isdisjoint('cd')
Out[3]: True

In [4]: vowels.union([1, 2])                      # |
Out[4]: {'o', 'a', 1, 2, 'u', 'e', 'i'}

In [5]: vowels.difference(['a', 'u'])             # -
Out[5]: {'e', 'i', 'o'}

In [6]: vowels.symmetric_difference(['a', 'b'])   # ^
Out[6]: {'b', 'e', 'i', 'o', 'u'}

In [7]: vowels.intersection('abcde')              # &
Out[7]: {'a', 'e'}
```

set method examples (2/2)

```
In [8]: s = set()
In [9]: s.update(vowels)          # add multiple elements
In [10]: s
Out[10]: {'a', 'e', 'i', 'o', 'u'}

In [11]: s.add('ciao')            # add one elements
In [12]: s
Out[12]: {'a', 'ciao', 'e', 'i', 'o', 'u'}

In [13]: s.remove('invalid')      # one element
KeyError: 'invalid'

In [14]: s.discard('invalid')     # no error
In [15]: s
Out[15]: {'a', 'ciao', 'e', 'i', 'o', 'u'}

In [16]: s.pop() # remove and return an arbitrary element
Out[16]: 'o'
```

Other useful containers

The `collections` module provides a useful set of containers, e.g.:

namedtuple	*factory function* for creating tuple subclasses with named fields
deque	list-like container with fast appends and pops on either end
OrderedDict	dict subclass that remembers the order entries were added
defaultdict	dict subclass that calls a factory function to supply missing values

How to get collections

```
In [1]: import collections      # or e.g.
In [2]: from collections import namedtuple
```

Mutable vs immutable object

```
In [1]: s1 = 'Hello, World'
In [2]: s2 = s1    # assignment: bind a name to an object
In [3]: s1 is s2   # identity
Out[3]: True       # s1 and s2 are labels for the same obj
In [4]: s1 += '!'  # the obj bound to s1 can't be modified
                   # --> rebinding: s1 = str(s1 + '!')
In [5]: s1 is s2
Out[5]: False

In [4]: d1 = {}    # mutable
In [5]: d2 = d1
In [6]: d2['key'] = 'value'
In [7]: d1
{'key': 'value'}

In [7]: d1 is d2   # d1 and d2 are still bound
Out[4]: True       # the same object
```

x Comprehensions

$$I = \{f(x), x \in A\}$$

```
In [1]: A = set([1, 2, 3, 4, 5])
In [2]: I = {x**2 for x in A}     # set comprehension
In [3]: I
Out[3]: {1, 4, 9, 16, 25}

In [4]: I_odd = {x**2 for x in A if x % 2 != 0}
In [5]: I_odd
Out[5]: {1, 9, 25}

In [6]: L = [x**2 for x in A]     # list comprehension

In [7]: D = {x: x**2 for x in A} # dict comprehension
Out[7]: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

x Comprehensions

$$I = \{f(x), x \in A\}$$

```
In [1]: A = set([1, 2, 3, 4, 5])
In [2]: I = {x**2 for x in A}     # set comprehension
In [3]: I
Out[3]: {1, 4, 9, 16, 25}

In [4]: I_odd = {x**2 for x in A if x % 2 != 0}
In [5]: I_odd
Out[5]: {1, 9, 25}

In [6]: L = [x**2 for x in A]     # list comprehension

In [7]: D = {x: x**2 for x in A} # dict comprehension
Out[7]: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

x Comprehensions

$$I = \{f(x), x \in A\}$$

```
In [1]: A = set([1, 2, 3, 4, 5])
In [2]: I = {x**2 for x in A}     # set comprehension
In [3]: I
Out[3]: {1, 4, 9, 16, 25}

In [4]: I_odd = {x**2 for x in A if x % 2 != 0}
In [5]: I_odd
Out[5]: {1, 9, 25}

In [6]: L = [x**2 for x in A]     # list comprehension

In [7]: D = {x: x**2 for x in A} # dict comprehension
Out[7]: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

x Comprehensions

$$I = \{f(x), x \in A\}$$

```
In [1]: A = set([1, 2, 3, 4, 5])
In [2]: I = {x**2 for x in A}     # set comprehension
In [3]: I
Out[3]: {1, 4, 9, 16, 25}

In [4]: I_odd = {x**2 for x in A if x % 2 != 0}
In [5]: I_odd
Out[5]: {1, 9, 25}

In [6]: L = [x**2 for x in A]     # list comprehension

In [7]: D = {x: x**2 for x in A} # dict comprehension
Out[7]: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

x Comprehensions

$$I = \{f(x), x \in A\}$$

```
In [1]: A = set([1, 2, 3, 4, 5])
In [2]: I = {x**2 for x in A}      # set comprehension
In [3]: I
Out[3]: {1, 4, 9, 16, 25}

In [4]: I_odd = {x**2 for x in A if x % 2 != 0}
In [5]: I_odd
Out[5]: {1, 9, 25}

In [6]: L = [x**2 for x in A]      # list comprehension

In [7]: D = {x: x**2 for x in A} # dict comprehension
Out[7]: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Generator expressions

```
In [8]: G = (x**2 for x in A)      # it is not a tuple!
In [9]: type(G)                    # a tuple is defined
Out[9]: generator                  # by commas

In [10]: hasattr(G, '__next__')
Out[10]: True

In [11]: for item in G:
    ...:     do_something(item)

In [12]: T = tuple(x**2 for x in A)
```

What is a generator?

A generator (iterator) is a iterable object that generates values dynamically when the `next(G)` function is called on it

Outline

If elif else

Example

```
if expr:
    pass
elif expr:
    pass
elif ...:
    ...
else:
    pass
```

- `if` followed by a boolean expression and ":" (colon)
- multiple (optional) `elif` clauses followed by a boolean expression and ":" (colon)
- (optional) `else` clause followed by ":" (colon)

Note

The `pass` keyword can be used a a placeholder where a code block is expected

No switch statement

Use if elif else

```
if key == 'a':
    do_stuff_for_a()
elif key == 'b':
    do_stuff_for_b()
elif key == 'c':
    do_stuff_for_c()
elif ...:
    ...
else:
    do_default_stuff()
```

... or a mapping

```
mapping = {
    'a': do_stuff_for_a,
    'b': do_stuff_for_b,
    'c': do_stuff_for_c,
    ...
}
# if key not in mapping
# select do_default_stuff
func = mapping.get(
    key, do_default_stuff)

# call the correct fnction
func()
```

for loop

for loop example

```
# simple loop
for item in sequence:
    do_stuff_with(item)

# complete example
for item in sequence:
    if not is_ok(item):
        continue
    elif should_we_break(item):
        break
    do_stuff_with(item)
else:   # no break
    do_some_other_stuff()
```

- `for` and `in` keywords followed by a ":" (colon)
- ... followed by a *code block*
- `break` and `continue` are allowed in loops
- the (optional) `else` clause if the loop ends regularly (no `break`)

while loop

while loop example

```
while condition:
    do_stuff()
    if some_special_case:
        continue
    elif some_other_specia_case:
        break
else:
    # no break
    do_some_other_stuff()
```

- `while` keyword followed by a boolean expression and ":" (colon)
- ... followed by a *code block*
- `break` and `continue` are allowed in loops
- the (optional) `else` clause if the loop ends regularly (no `break`)

More loops

```
In [1]: for index, item in enumerate(['a', 'b']):
   ...:     print('Item {}: {}'.format(index, item))
Item 1: a
Item 2: b

In [2]: for key, value in [('a', 1), ('b', 2)]:
   ...:     print('key "{}": {}'.format(key, value))
key "a": 1
key "b": 2

In [3]: for key, value in zip(['a', 'b'], [1, 2]):
   ...:     print('key "{}": {}'.format(key, value))
key "a": 1
key "b": 2

In [4]: for char in 'string':
   ...:     print(char, end='  ')
s  t  r  i  n  g
```

for loop anti-pattern

Bad: indexing

```
In [1]: seq = ['a', 'b']
In [2]: for i in range(2):
   ...:     print('Item {}: {}'.format(i, seq[i]))
```

- lower level approach
- not necessary indexing (bound checking –> inefficiency)
- only works on objects that can be indexed ([] operator)
- ... doesn't work in the general case (iterators, generators, ...)

Good: use enumerate instead

```
In [1]: for index, item in enumerate(['a', 'b']):
   ...:     print('Item {}: {}'.format(index, item))
```

Exception handling

```
try:
    do_something()
except ExceptionType:
    handle_this_exception_type()
except (ExcType1, ExcType2):
    pass    # explicitly ignore
except AnotherType as exc:
    msg = str(exc)
    do_something_with(msg)
except Exception: # error base class
    logger.warning('message')
    raise   # re-raise
finally:
    pass # always executed
else:
    pass # no exception occurred
```

- `try` keyword followed by a ":" (colon)
- the *code block*
- multiple (optional) except clauses
- the (optional) `finally` clause is always executed
- the (optional) `else` clause is executed only id no exception occurred

Exception raising

```
# standard exception raising
raise ExceptionType('message')

# re-raising
except ExceptionType:
    raise

# chained exceptions (Python 3 only)
# give more information about the exception origin
except ExceptionType as exc:
    raise AnotherExceptionType('message') form exc
```

with statement

Problem

```
fd = open(filename)
do_something(fd) # raise
fd.close() # not executed
```

Old style solution (don't use it)

```
fd = open(filename)
try:
    do_something(fd)
finally:
    fd.close()
```

Resources acquired in the with line are automatically released

with statement

```
with open(fname) as fd:
    do_something(fd)

lock = threading.Lock()
with lock:
    do_something_else()

with lock, open(fname) as fd:
    do_something(fd)
```

Iterating on files

Iteration of text files

```
with open(src) as fin, open(dst, 'w') as fout:
    for line in fin:          # line oriented iteration
        line = line.rstrip('\n') # newlines are kept
        fout.write(line[::-1])    # reverse the line
        fout.write('\n')
```

Iteration of binary files

```
def simple_copy_file(src, dst):
    with open(src, 'rb') as fin, \
         open(dst, 'wb') as fout:
        for block in fin:         # default block size
            fout.write(block)  # is 4096 or 8192
```

The io module and the like-like interface

- The io module provides Python's main facilities for dealing with various types of I/O: text I/O, binary I/O and raw I/O
- A concrete object belonging to any of these categories is called a file object (or **file-like object**)
- io.StringIO and io.BytesIO provide a **file-like interface** for strings and buffers

The built-in open factory (io.open in Python 3)

```
open(file, mode='r', buffering=-1, encoding=None,
     errors=None, newline=None, closefd=True,
     opener=None)
```

File like objects

Attributes

- closed
- newlines
- line_buffering
- encoding
- errors
- name (real file only)
- mode (real file only)

Methods

- isatty
- seekable
- readable
- writable
- fileno
- truncate

- close
- flush
- read
- readline
- readlines
- write
- writelines
- seek
- tell

Outline

Function definition

```
def func(arg1, arg2, ...):
    '''Docstring'''

    [function body]

    return result

# calling a function
result = func(arg1, arg2, ...)

# or
func(arg1, arg2, ...)
```

- def keyword
- function name
- (optional) argument
- colon
- indentation
- (optional) docstring
- (optional) body
- (optional) return keyword followed bay and (optional) value

Return values

```
def func():
    pass

result = func()
result is None  # --> True


def func():
    return 1, 2

result = func()
type(result)    # --> tuple

a, b = func()   # unpacking
```

- if no return value is specified the None is returned
- multiple values can be returned
- multiple return values are packed into a tuple
- tuple unpacking can be used to set multiple values in the caller scope

Default arguments

- Function arguments may have default values
- Arguments with default values must come after arguments without defaults

```
In [1]: def hello(name='World'):
   ...:     print('Hello, %s!' % name)

In [2]: hello()
Hello, World!

In [3]: import os
In [4]: name = os.getenv('USER')      # antonio
In [5]: hello(name.capitalize())
Hello, Antonio!
```

Never use mutable objects for default arguments

- default arguments are attributes of the **function object**
- if someone modifies the mutable object it stay modified for all the following calls

```
def bad_func(a, L=[]):
    L.append(a)
    return L

def good_func(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

```
In [1]: bad_func.__defaults__
Out[1]: ([],)

In [2]: bad_func(1)
Out[2]: [1]

In [3]: bad_func(2)
Out[3]: [1, 2]

In [4]: bad_func.__defaults__
Out[4]: ([1, 2],)
```

Positional vs keyword arguments

```
In [1]: def func(a, b, c=None, d=None):
   ...:     pass

In [2]: func(1, 2, 3, 4)            # all positional

In [3]: func(1, 2, d=4)             # 'd' passed by name
                                    # default 'c' parameter

In [4]: func(1, d=4, b=2)           # arbitrary order
                                    # for keyword args

In [5]: func(c=3, a=1, d=4, b=2)    # all keyword

In [6]: args = tuple(1, 2)
In [7]: kwargs = {'c': 3, 'd': 4}
In [5]: func(*args, **kwargs)       # args unpacking
```

Variable number of arguments (1/2)

```
def trace(func, *args, **kwargs):
    # func is mandatory
    # type(args)   --> tuple - default ()
    # type(kwargs) --> dict  - default {}

    name = func.__name__                # functions are objects
    args_str = ', '.join(
            [repr(arg) for arg in args] +
            ['%s=%r' % (k, v) for k, v in kwargs.items()])

    print('TRACE: %s(%s)' % (name, args_str))

    ret = func(*args, **kwargs)  # argumnts unpacking

    print('TRACE: %s --> %r' % (name, ret))

    return ret
```

Variable number of arguments (2/2)

```
In [1]: def simple_sum(a, b, c=0):
   ...:     print('This is "simple_sum"')
   ...:
   ...:     return a + b + c

In [2]: trace(simple_sum, 10, b=20, c=30)

TRACE: simple_sum(10, c=30, b=20)
This is "simple_sum"
TRACE: simple_sum --> 60
```

Scopes

Variable look-up:

- local (function) scope
- parent scope
- parent scope of the parent scope
- ...
- up to the module global scope

Example

```
In [1]: BASE = 10
In [2]: def pow10(n):
   ...:     return BASE ** n
In [3]: pow10(2)
Out[3]: 100
```

Closures

References to the globals scope and to non-local variables from parent scopes are stored in the function object

Example

```
In [1]: def pow_func_generator(base=10):
   ...:     def pow_func(n):
   ...:         return base ** n # base: parent scope
   ...:     return pow_func      # returns a function

In [2]: # this is another way to define a function
In [3]: pow2 = pow_func_generator(base=2)

In [4]: # no 'base' variable defined in this scope
In [5]: pow2(3)
Out[5]: 8
```

Decorators

```
import logging
import functools

def trace(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        name = func.__name__
        args_str = ', '.join(
            [repr(arg) for arg in args] +
            ['%s=%r' % (k, v) for k, v in kwargs.items()])
        logging.debug('%s(%s)' % (name, args_str))
        ret = func(*args, **kwargs)
        logging.debug('%s --> %r' % (name, ret))
        return ret
    return wrapper
```

Decorators what?

A **decorator**

- is function returning another function
- performs some generic transformation (checks, logging, arguments filtering, etc.)
- can be applied using the special `@wrapper` syntax

Example

```
# equivalent to: do_something = trace(do_something)
In [1]: @trace
   ...: def do_something():
   ...:     pass

In [2]: do_something()
DEBUG: do_something()
DEBUG: do_something --> None
```

Prefer EAFP over LBYL

EAFP: It's **E**asier to **A**sk **F**orgiveness than **P**ermission (É più facile chiedere perdono che il permesso)

LBYL: **L**ook **B**efore **Y**ou **L**eap (Guarda prima di saltare)

Example

```
def simple_sum_eafp(a, b):
    return a + b

def simple_sum_lbyl(a, b):
    if not isinstance(a, (int, float)):
        raise TypeError('invalid type for "a"')
    [...]
    return a + b
```

Duck typing

The duck test

If it walks like a duck, and talks like a duck, and looks like a duck: it's a duck. (Goose? Close enough.)

```
In [1]: simple_sum_eafp(2, 2)
Out[1]: 4

In [2]: simple_sum_lbyl(2, 2)
Out[2]: 4

In [3]: # str quacks
In [4]: simple_sum_eafp('Hello, ', 'World!')
Out[4]: 'Hello, World!'
```

Duck typing vs C++ templates

Duck typing may look similar to C++ templates in the practical use.
Technically specking they are dramatically different:

- all is resolved at runtime in Python
- all have to be known at compile time in C++ templates (static polymorphism)

Python

```
def simple_sum(a, b):
    return a + b
```

C++

```
template<typename T>
T simple_sum(const T& a,
             const T& b)
{
    return a + b;
}
```

Outline

Modules

- a module is a file containing Python definitions and statements (functions, classes, variables, etc.)
- the name of the module in the code is the same of the file without the suffix `.py`
- within a module, the module's name (as a string) is available as the value of the global variable `__name__`
- modules are also `objects`, `__name__` and `__doc__` (for *docstrings*) are attributes of the module object
- modules can be `imported` from other modules and scripts
- the import process automatically compiles modules (not scripts) into bytecode and caches it into a `.pyc` (or `.pyo`) file

Packages

- a packages is a directory collecting modules
- a directory is a package if it includes an `__init__.py` file (eventually empty)

Package example

```
pkg
|-- __init__.py
|-- mod01.py
|-- mod02.py
+-- __pycache__
    |-- __init__.cpython-35.pyc
    +-- mod01.cpython-35.pyc
+-- subpkg
    |-- __init__.py
    +-- mod11.py
```

Using modules and packages

The `import` statement:

- search the module file
- compile it to bytecode if necessary
- executes the module (creates a module object, classes and functions in it), and
- binds the module to a local variable

Import examples

```
import module
import module as alias
from module import something

import pkg
import pkg.mod01
from pkg.subpkg import mod11

# In pkg.subpkg.__init__.py
from pkg.subpkg import mod11
from . import mod11 # relative
from .. import mod01
```

Modules search path

- Only modules and packages in the Python search path can be imported
- The system search path and the current working directory are always in the Python search path

Path control from code

```
import sys
sys.path.insert(0, "some/path")   # sys.path is a list
```

Path control from the environment

```
export PYTHONPATH=some/path:${PYTHONPATH}
```

Using modules as scripts

- the *shebang* string is used to specify the interpreter
- the encoding specification
- modules can have a *docstring*
- function and classes are declared
- if the file is imported as a module, then the `__name__` is set to the module name
- if the file is run as a script then the `__name__` is set to `'__main__'`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

'''Module docstring'''

def func():
    pass

def main():
    func()


if __name__ == '__main__':
    main()
```

Outline

class

```
class UserTypeName(object):
    '''Class docstring'''

    def __init__(self, arg1, ...):
        '''Initializer docstring'''

        self.attr = arg1

    def method(self, arg1, ...):
        '''Method docstring'''

        do_something()
        ...

        return value
```

- class keyword
- base class(es), object is redundant in Python 3
- class *docstring*
- initializer (special method)
- methods like function
- explicit **self** parameter (instance) in method definition

self

```
from math import pi

class Circle(object):
    def __init__(self, r):
        self.r = r

    def area(self):
        # pi * r**2 --> error
        return pi * self.r**2

c = Circle(r=1)
area = c.area()

# also possible
area = Circle.area(c)
```

- **self** is the instance on which the method acts
- the **self** argument must be explicitly listed in method definition
- **self** is the only way to access instance attributes and methods
- no need specify **self** in (*bound*) method call
- it is also possible to call *unbound methods* passing the instance explicitly

__init__

- __init__ is a **special method**
- it is automatically called when an object is created
- __init__is used to initialize instance attributes (*initializer* method)

Readability counts

Although it is not mandatory it is strongly recommended to initialize **all** attributes in the `__init__` method or `base.__init__`

Freedom and good judgment

```
class UserType(object):
    def __init__(self, a):
        self.a = a

    def method(self, b):
        self.b = b

obj = UserType('a')
hasattr(obj, 'a')   # True
hasattr(obj, 'b')   # False

obj.method('b')
hasattr(obj, 'b')   # True!

obj.c = 'c'   # also allowed
```

- Python is very *flexibility* and gives the developer *freedom* to do almost anything
- **it is responsibility of the developer to mess around**
- the fact that it is possible to do something doesn't mean it is a good idea

Visibility (public, private, ...)

- In Python do not exist visibility access modifiers
- Python does not enforces access restrictions at language level
- an universally known naming convention is used instead
- Protected/private attributes/methods start with a single underscore: `self._protected_attribute`

Avoiding name clashes

- Attributes starting with a double underscore (`__name`) are not directly accessible from outside the class definition code
- If have to be intended as a mechanism to avoid name clashes with names defined by sub-classes rather then a way to implement access restrictions
- Those attributes are accessible by the client code simply using: `obj._ClassName__name`

Visibility examples

```
In [1]: class UserType(object):
   ...:     def __init__(self):
   ...:         self._a = 'protected/private'
   ...:         self.__b = 'safe name' # protected/private
   ...:
   ...:     def method(self):
   ...:         self._a = 'something'
   ...:         self.__b = ''              # simple access

In [2]: obj = UserType()
In [3]: obj._a
Pur[3]: 'protected/private'

In [4]: obj.__b
AttributeError: 'UserType' object has no attribute '__b'

In [5]: obj._UserType__b
Out[5]: 'safe name'
```

Inheritance

- Python supports both single and multiple inheritance
- Base classes are listed in parentheses after the (derived) class name
- The base class constructor have to be called explicitly (if it is necessary)

```
class Derived(Base):
    def __init__(self, *args, **kwargs):
        # Base.__init__(self, *args, **kwargs) # or
        super().__init__(*args, **kwargs) # recommended

        self.derived_attr = None    # derived class code

    def method(self):
        do_other_stuff()                # derived class code
        return super().method(*args, **kwargs)
```

super

- super returns a proxy object that delegates method calls to a parent class
- It is the key element to implement the so called **cooperative-inheritance**
- In Python 2 you have explicitly inherit form object to use super

super in Python 2 and Python 3

```
# Pythpn 2 and Python 3
super(Derived, self).__init__(*args, **kwargs)

# Python 3 only
super().__init__(*args, **kwargs)
```

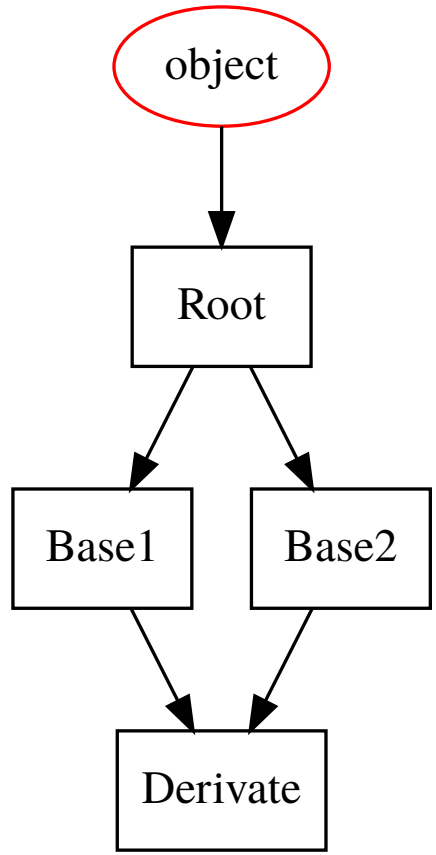
Multiple inheritance (1/2)

```
class Root(object):
    def __init__(self, arg):
        print('Root(%r)' % arg)
        # no args for object
        super().__init__()

class Base1(Root):
    def __init__(self, arg):
        print('Base1(%r)' % arg)
        super().__init__(arg)

class Base2(Root):
    def __init__(self, arg):
        print('Base2(%r)' % arg)
        super().__init__(arg)
```

Diamond inheritance



Multiple inheritance (2/2)

```
class Derived(Base1, Base2):
    def __init__(self, arg):
        print('Derived(%r)' % arg)
        super().__init__(arg)
```

Example

```
In [1]: obj = Derived('arg')
Derived('arg')
Base1('arg')
Base2('arg')        # NOTE: Base2 before Root
Root('arg')         # NOTE: Root appears only once
```

Method Resolution Order (MRO)

Methods are called according to the MRO

- current class (`Derived`)
- parent classes in order (`Base1`, `Base2`)
- then parents of parent classes (`Root` and `object`)

Please note

- the method of the `Root` class is called only once
- and always after the methods of all the `Base*` classes

The `mro()` static method

```
In [2]: Derived.mro()
Out[2]: [<class '__main__.Derived'>,
    ...:  <class '__main__.Base1'>,
    ...:  <class '__main__.Base2'>,
    ...:  <class '__main__.Root'>,
    ...:  <class 'object'>]
```

Breaking the super chain

What if I forgot to call the super().method in e.g. Base1?

```
class Base1(Root):
    def __init__(self, arg):
        print('Base1(%r)' % arg)
        # super().__init__(arg)   # call chain broken
```

Then Base1('arg') and Root('arg') are never called

```
In [1]: obj = Derived('arg')
Derived('arg')
Base1('arg')
# Base2('arg')    # not called
# Root('arg')     # not called
```

Inheriting form `object`

- The `object` type was introduced in Python 2.2 to maintain compatibility with the old object model:
  - classes not inheriting from `object` maintained the old behavior
  - classes inheriting form `object` behave according to the new object model
- *Old style classes* are still the default Python 2.7 while the new object model is to only one in Python 3
- In short: inheriting form `object` is
  - mandatory in Python 2 if woy want the new object model
  - is redundant in Python 3
  - recommended in Python 3 if you want to write code that is compatible with Python 2

Special methods (protocols)

- Methods starting and anding with a double underscore (`__*__`) are called *special methods*
- sometimes special methods receive a special treatment, e.g.
  - `__init__` is automatically called when an instance is created
  - operators: `__add__` (+, and `sum`), `__sub__` (-), `__mul__` (*), `__truediv__` (/), `__floordiv__` (//), `__mod__` (%)
  - comparison: `__eq__` (==), `__ne__` (!=), `__lt__` (<), `__le__` (<=), `__gt__` (>), `__ge__` (>=)
  - `__contains__` (`in`), `__call__`
  - `__getitem__`, `__setitem__` (indexing `[]`)
- .. or they are used to implement the so called **protocols**:
  - `len(obj)` (`__len__`), `pow(num)` (`__pow__`), `abs(num)` (`__abs__`), `str(obj)` (`__str__`), `repr(obj)` (`__repr__`), `iter(obj)` (`__iter__`), `next(obj)` (`__next__`)
  - `with` statement (`__enter__`, `__exit__`)

Defining/overloading special methods

```
class UserType(object):
    def __repr__(self):
        return 'bla bla bla'

    def __enter__(self):
        print('acquiring resources')
        return self    # used by the 'as' clause

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('releasing resources')
```

```
In [1]: obj = UserType()
In [2]: obj
Out[2]: bla bla bla
```

```
In [3]: with obj:
   ...:     print('do job')
acquiring resources
do job
releasing resources
```

Static methods/attributes

```
class UserType(object):
    STATIC_CONSTANT = None   # static attribute

    def __init__(self):
        self.default = self.STATIC_CONSTANT # note the self

    @staticmethod                # property
    def some_static_method(*args, **kwargs):
        # NOTE: no self argument here
        print('static')

In [1]: print(UserType.STATIC_CONSTANT)
Out[1]: None

In [2]: # also works with obj.some_static_method()
In [3]: UserType.some_static_method()
Out[3]: static
```

Properties

- Sometimes some of the attributes of an object are strictly related and have values that depend on each other: e.g. the radius and the diameter, circumference , area, etc. of a circle
- It is natural to think to them as attributes of the object
- In other programming languages (e.g. C++ and Java) it is recommended to keep attributes private or protected and access them via *getter/setter methods*
- ... but *getter/setter methods* are boring
- *getter/setter methods* are basically a work-around for a limitation of the language: it is not possible to keep in sync two attributes if one if them is modified directly

Properties: use case

```
In [1]: c = Circle()
In [2]: c.radius = 3          # set the redius
In [3]: c.diameter
Out[3]: 6                     # the diameter stay consistent

In [4]: c.area
Out[4]: 28.274333882308138    # ... and the area too

In [5]: c.diameter = 5        # now the diameter is changed
In [6]: c.radius
Out[6]: 2.5                   # hey, the radius has been
                              # updated as well!

In [7]: c.circumference
Out[7]: 15.707963267948966    # and the circumference too
```

Defining Properties

```
class Circle(Shape):
    def __init__(self, r=0):
        self._r = r                 # one protected attribute

    @property                       # decorator
    def radius(self):               # getter
        return self._r
    @radius.setter
    def radius(self, value):        # setter
        self._r = value

    @property
    def diameter(self):             # getter
        return 2 * self._r
    @diameter.setter
    def diameter(self, value):      # setter: if omitted then
        self._r = value / 2         # the property is read-only
```

Outline

Setting up a virtual environment

```
$ python3 -m venv testenv

$ ls testenv/
bin  include  lib  lib64  pyvenv.cfg  share

$ source testenv/bin/activate

(testenv) $ which python
/home/antonio/python-training/examples/testenv/bin/python

(testenv) $ pip install pep8
pip install pep8
Collecting pep8
  Using cached pep8-1.7.0-py2.py3-none-any.whl
Installing collected packages: pep8
Successfully installed pep8-1.7.0
```

Exercise solution

```
def pretty_file_size_str(size):
    '''String representation of file size

    Gets a size in bytes and return the its string
    representation using appropriate units.

    '''

    UNITS = ('Bytes', 'KB', 'MB', 'GB', 'TB', 'PB')

    for exponent, units_str in enumerate(UNITS):
        normalized_size = size / 1024**exponent
        if normalized_size < 1024:
            break

    return '%.2f %s' % (normalized_size, units_str)
```

Using doctests

```
def pretty_file_size_str(size):
    '''String representation of file size

    ...

    >>> pretty_file_size_str(1.3 * 1024**5)
    '1.30 PB'

    >>> pretty_file_size_str(1.3 * 1024)
    '1.30 KB'

    >>> pretty_file_size_str(18)
    '18.00 Bytes'

    >>> pretty_file_size_str(0.3)
    '0.30 Bytes'

    '''
```

Running doctests

```
$ python3 -m doctest -v prettysize_exercise.py
Trying:
    pretty_file_size_str(1.3 * 1024**5)
Expecting:
    '1.30 PB'
ok
Trying:
    pretty_file_size_str(1.3 * 1024)
Expecting:
    '1.30 KB'
ok
...
1 items passed all tests:
   4 tests in prettysize_exercise.pretty_file_size_str
4 tests in 2 items.
4 passed and 0 failed.
Test passed.
```

Exercise solution - Part 2

```
if __name__ == '__main__':
    import os, hashlib, functools as fts

    # a. print the fill module path
    print('filename: {}'.format(os.path.abspath(__file__)))

    # b. print the module size
    size = os.path.getsize(__file__)
    print('size: {}'.format(pretty_file_size_str(size)))

    # c. print the MD5 checksum of the module file
    md5 = hashlib.md5()
    with open(__file__, 'rb') as fd:
        for data in iter(fts.partial(fd.read, 2048), b''):
            md5.update(data)
    basename = os.path.basename(__file__)
    print('{} *{}'.format(basename, md5.hexdigest()))
```

Output

```
$ python3 prettysize_exercise.py

filename: /home/antonio/python-training/.../prettysize.py
size: 1.28 KB
prettysize_exercise.py *20623019d6b3bdc22898c9b567262434
```

Finding duplicate files

Problem:

- FTP site for Sentinel 1 data activities
- Huge amount of data
- Organized in a directory tree
- Unfortunately the same product can be present in different directories
- ... even worst, the same product processed by different entities or at different time is stored with different file name

Requirements

Write a Python program that finds all duplicate files in the specified directory and:

- prints the total number of files
- prints the total number of duplicate files
- (optionally) computes the amount of disk space wasted by duplicates
- (optionally) prints the list of duplicate files

The program shall have a simple command line user interface also having an option for printing the usage documentation

Data structure

Duplicate files can be stored in a dictionary:

keys: are base names of files (and are unique)

values: are a list of full path names of all replicas

The dictionary is associated to `Result` class that also provides methods to manipulate it

Example

```
result.data = {
    'A.SAFE': ['PATH/1/A.SAFE', 'PATH/2/A.SAFE', ...],
    'B.SAFE': ['PATH/1/B.SAFE', 'PATH/2/B.SAFE'],
}
```

Test Driven Development approach (TDD)

- write tests **before** the program code
- use automated tests for unit testing
- tests express specifications
- the SW is OK when it passes all tests
- iterative development
- short development cycles
- phases:
  - write automatic tests for the new function (specifications/API design): the code still doesn't exist so tests fail
  - development of the new function by writing the minimal amount code to make all tests pass
  - refactoring: re-work the code to harmonize all parts of the program and have a better overall design. All test shall pass untouched

Basic tests for our duplicate file finder

```
class ScanDuplicatesTestCase(CreateTestDirMixin,
                             unittest.TestCase):
    def test_scan_result(self):
        result = s1duplicates.scan_duplicates(self.root)
        self.assertTrue(hasattr(result, 'data'))

    def test_scan_result_data(self):
        result = s1duplicates.scan_duplicates(self.root)
        self.assertTrue(
            isinstance(result.data, collections.Mapping))

    def test_duplicate_number(self):
        result = s1duplicates.scan_duplicates(self.root)
        self.assertEqual(result.duplicate_count(),
                         len(self.DATASET))
```

Basic tests for our duplicate file finder

```
class ScanDuplicatesTestCase(CreateTestDirMixin,
                             unittest.TestCase):
    def test_scan_result(self):
        result = s1duplicates.scan_duplicates(self.root)
        self.assertTrue(hasattr(result, 'data'))

    def test_scan_result_data(self):
        result = s1duplicates.scan_duplicates(self.root)
        self.assertTrue(
            isinstance(result.data, collections.Mapping))

    def test_duplicate_number(self):
        result = s1duplicates.scan_duplicates(self.root)
        self.assertEqual(result.duplicate_count(),
                         len(self.DATASET))
```

Basic tests for our duplicate file finder

```
class ScanDuplicatesTestCase(CreateTestDirMixin,
                             unittest.TestCase):
    def test_scan_result(self):
        result = s1duplicates.scan_duplicates(self.root)
        self.assertTrue(hasattr(result, 'data'))

    def test_scan_result_data(self):
        result = s1duplicates.scan_duplicates(self.root)
        self.assertTrue(
            isinstance(result.data, collections.Mapping))

    def test_duplicate_number(self):
        result = s1duplicates.scan_duplicates(self.root)
        self.assertEqual(result.duplicate_count(),
                         len(self.DATASET))
```

Test directory

- defined at `TestCase` level
- automatically generated before each test execution
- automatically removed after each test case execution

Same `self.DATASET` replicated in two places

```
dataIN
+-- S1A
    +-- S1A_IW
    |   |-- S1A_IW_RAW__0A
    |   |   +-- S1A_IW_RAW__0ADV_20160619T053500.SAFE
    |   +-- S1A_IW_RAW__0S
    |       +-- S1A_IW_RAW__0SDV_20160619T053500.SAFE
    +-- DT_SITE
        |-- S1A_IW_RAW__0ADV_20160619T053500.SAFE
        +-- S1A_IW_RAW__0SDV_20160619T053500.SAFE
```

Unit testing framework

- Python provides a complete unit testing framework
- the `unittest` package of the standard library
- it is based on the xUnit architecture. Main elements
  - test fixtures
  - test cases
  - test suites
  - test runners

Automatic test discovery

- All classes inheriting form `unittest.TestCase` are tests
- Only methods starting with `test*` are assumed to be tests
- There is a machinery to automatically discover all tests in a source tree: it relies on naming conventions and only files starting with `test*.py` are considered

unittest package

```
class TestStringMethods(unittest.TestCase):
    def setUp(self):
        # prepare preconditions before test execution:
        # objects creation, initialization, etc.

    def tearDown(self):
        # perform cleanup actions after the test execution
        # e.g. remove files, closes connections, etc.

    def test_something(self):
        # specific test case: use assertions
        self.assertEqual(something, something_else)

if __name__ == '__main__':
    # run all tests when executed as a script
    unittest.main()
```

Mixin classes in Python

- Mixin classes factorize some common behavior that can be part of possibly unrelated classes
- Usually mixins does not have a complete state
- Features of a mixin are included by inheriting it

```
class CreateTestDirMixin(object):
    ROOT = 'dataIN'

    def setUp(self):
        self.testdir = tempfile.mkdtemp()
        self.root = os.path.join(self.testdir, self.ROOT)
        self._setup_test_dataset()

    def tearDown(self):
        shutil.rmtree(self.testdir)
```

Creating the dataset tree

```
DATASET = ( ... )
REPLICA_DIR = 'S1A/DT_SITE'
DEFAULTSIZE = 1024

def _setup_test_dataset(self):
    replicadir = os.path.join(self.root, self.REPLICA_DIR)
    os.makedirs(replicadir)

    for path in self.DATASET:
        dirname, basename = os.path.split(path)
        dirname = os.path.join(self.root, dirname)
        os.makedirs(dirname)
        filename = os.path.join(dirname, basename)
        create_dummy_file(filename, self.DEFAULTSIZE)

        replica = os.path.join(replicadir, basename)
        create_dummy_file(replica, self.DEFAULTSIZE)
```

Creating dummy files

```
def create_dummy_file(filename, size=None):
    with open(filename, 'wb') as fd:
        if size:
            fd.seek(size-1)
            fd.write(b'\0')
```

Note

- using the `filelike_obj.seek` method it is possible to create potentially huge files without managing large buffers and performing virtually no I/O: just one byte is written
- please note the bytes literal (`b'\0'`)

Run the test suite

```
$ python3 test_s1duplicates.py
======================================================
ERROR: test_scan_result (__main__.ScanDuplicatesTestCase)
------------------------------------------------------
Traceback ...
result = s1duplicates.scan_duplicates(self.root)
AttributeError: module 's1duplicates' has no attribute
'scan_duplicates'
======================================================
...
Ran 3 tests in 0.004s
FAILED (errors=3)
```

Note

None of the test pass at this stage, we just have an empty `s1duplicate.py` module

Step 2: write the code

```
def scan_duplicates(dataroot):
    scanned_files = 0
    data = defaultdict(list)
    for root, dirs, files in os.walk(dataroot):
        scanned_files += len(files)
        for filename in files:
            fullpath = os.path.join(root, filename)
            if not os.path.islink(fullpath):
                data[filename].append(fullpath)

    # remove non duplicates
    for key in list(data.keys()):   # note: copy keys
        val = data[key]
        if len(val) < 2:
            del data[key]

    return Result(data, scanned_files)
```

... and the Result class

```
class Result(object):
    def __init__(self, data, scanned_files):
        self.data = data
        self.scanned_files = scanned_files

    def duplicate_count(self):
        values = self.data.values()
        return sum(len(item) - 1 for item in values)
```

Test execution

```
$ python3 test_s1duplicates.py
...
----------------------------------------
Ran 3 tests in 0.003s

OK
```

New requirement: computation of wasted disk space

```
def test_duplicate_size(self):
    result = s1duplicates.scan_duplicates(self.root)
    size = self.DEFAULTSIZE * len(self.DATASET)
    self.assertEqual(result.duplicate_size(), size)
```

Test execution

```
ERROR: test_duplicate_size ...
----------------------------------------------------------------
Traceback (most recent call last): ...
    self.assertEqual(result.duplicate_size(), size)
AttributeError: 'Result' object has no attribute
'duplicate_size'
----------------------------------------------------------------
Ran 4 tests in 0.005s
FAILED (errors=1)
```

New requirement ... implementation

```
def duplicate_size(self):
    size = 0
    for key in self.data:
        duplicates = self.data[key]
        filename = duplicates[0]     # first path
        filesize = os.path.getsize(filename)
        size += filesize * (len(duplicates) - 1)

    return size
```

Test execution

```
....
Ran 4 tests in 0.005s

OK
```

Command line interface

```
$ python3 s1duplicates.py -h
usage: s1duplicates.py [-h] [--version] [-s] [-l] [-v] dataroot

Search all duplicate files in the specified directory tree

positional arguments:
  dataroot              the root of the directory tree to scan
                        for duplicates

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -s, --compute-size    compute the total size of duplicate files
                        (default: False)
  -l, --list-files      dump the entire list of duplicate files
                        (default: False)
  -v, --verbose         print verbose help messages
```

Sample output

```
$ python3 s1duplicates.py dataIN
2 duplicate files found

$ python3 s1duplicates.py -s dataIN
2 duplicate files found
2.00 Bytes of duplicate files can be removed

$ python3 s1duplicates.py -l dataIN
2 duplicate files found
Duplicates in "dataIN/"
defaultdict(<class 'list'>,
{'BASENAME_1.SAFE': ['FULLPATH_1.SAFE', 'FULLPATH_2.SAFE'],
 'BASENAME_2.SAFE': ['FULLPATH_3.SAFE', 'FULLPATH_4.SAFE']}
```

Unittests for the command line UI (1/3)

```
class UiTestCase(CreateTestDirMixin, unittest.TestCase):
    VERSION = '1.0'
    BASEARGS = [sys.executable, '-u', 's1duplicates.py']

    def test_version(self):
        args = self.BASEARGS + ['--version']
        outbytes = subprocess.check_output(args)
        out = outbytes.decode('utf-8')
        self.assertTrue(self.VERSION in out)

    def test_help(self):
        args = self.BASEARGS + ['--help']
        outbytes = subprocess.check_output(args)
        out = outbytes.decode('utf-8')
        self.assertTrue(out.startswith('usage:'))
```

Unittests for the command line UI (2/3)

```
def test_duplicate_number(self):
    args = self.BASEARGS + [self.root]
    outbytes = subprocess.check_output(args)
    out = outbytes.decode('utf-8')
    self.assertEqual(
        out.strip(),
        '%d duplicate files found' % len(self.DATASET))

def test_duplicate_size(self):
    args = self.BASEARGS + ['-s', self.root]
    outbytes = subprocess.check_output(args)
    out = outbytes.decode('utf-8')
    count_line, size_line = out.strip().splitlines()
    size = self.DEFAULTSIZE * len(self.DATASET)
    self.assertEqual(
        size_line,
        'duplicate file size: %s' % pretty_file_size_str(si
```

A bit of regular expressions (re)

ITEM_RE

```
"'(?P<key>.*\.SAFE)':
\[(?P<values>'.*\.SAFE'(,\s+'.*\.SAFE')*\])"
```

- .: matches any character
- .*: matches an arbitrary number of characters
- \.: (escaped dot) matches a dot
- .*\.SAFE: is equivalent to the glob expression *.SAFE
- (?P<key>.*\.SAFE): the matched expression is saved into the group named "key"
- \s: matches ant white character
- \s+: matches one or more (at least one) white character

Unittests for the command line UI (3/3)

```
def test_duplicate_list(self):
    args = self.BASEARGS + ['-l', self.root]
    outbytes = subprocess.check_output(args)
    out = outbytes.decode('utf-8')

    keys = set(
        os.path.basename(item) for item in self.DATASET)
    found_keys = []
    for i, mobj in enumerate(self.ITEM_RE.finditer(out)):
        key = mobj.group('key')
        self.assertTrue(key in keys)
        found_keys.append(key)

        values = mobj.group('values')
        self.assertEqual(len(values.split(',')), 2)

    self.assertEqual(len(found_keys), len(keys))
```

Command line UI implementation

```
parser = ArgumentParser(description='Search all '
    'duplicate files in the specified directory tree')
parser.add_argument(
    '--version', action='version', version='%(prog)s v1.0')
parser.add_argument('-s', '--compute-size',
    action='store_true', default=False, help='compute '
    'the total size of duplicate files '
    '(default: %(default)s)')
parser.add_argument('-l', '--list-files',
    action='store_true', default=False,
    help='dump the entire list of duplicate files '
    '(default: %(default)s)')
parser.add_argument('-v', '--verbose', action='store_true',
    default=False, help='print verbose help messages')
parser.add_argument('dataroot', help='the root of the '
    'directory tree to scan for duplicates')
```

Main program

```
args = parser.parse_args()

if args.verbose:
    logging.getLogger().setLevel(logging.DEBUG)

result = scan_duplicates(args.dataroot)
logging.info(
    '%d duplicate files found' % result.duplicate_count())

if args.compute_size:
    size = result.duplicate_size()
    sizestr = prettysize.pretty_file_size_str(size)
    logging.info(
        '%s of duplicate files can be removed' % sizestr)
if args.list_files:
    logging.info('Duplicates in "%s"' % args.dataroot)
    pprint.pprint(result.data)
```

Final test

```
$ python3 -m unittest -v test_s1duplicates.py
test_duplicate_number (test_s1duplicates.ScanDuplicatesTestCase)
    ... ok
test_duplicate_size (test_s1duplicates.ScanDuplicatesTestCase)
    ... ok
test_scan_result (test_s1duplicates.ScanDuplicatesTestCase)
    ... ok
test_scan_result_data (test_s1duplicates.ScanDuplicatesTestCase)
    ... ok
test_duplicate_list (test_s1duplicates.UiTestCase) ... ok
test_duplicate_number (test_s1duplicates.UiTestCase) ... ok
test_duplicate_size (test_s1duplicates.UiTestCase) ... ok
test_help (test_s1duplicates.UiTestCase) ... ok
test_version (test_s1duplicates.UiTestCase) ... ok

----------------------------------------------------------------------
Ran 9 tests in 0.267s

OK
```

References I

 Official Python documentation

 Official Python tutorial

 Code Like a Pythonista: Idiomatic Python, by David Goodger

 Introduction to Python, by Nowell Strite

 Mark Pilgrim
Dive into Python 3
Apress 2009

 Alex Martelli
Python in a Nutshell, 3rd Edition
O'Reilly 2016

References II


Mark Lutz
Learning Python, 5th Edition - Powerful Object-Oriented Programming
O'Reilly 2013


Mark Summerfield
Programming in Python 3: A Complete Introduction to the Python Language (Second Edition)
Addison-Wesley, November 2009


Marco Beri
Python
Apogeo 2010