Fundamentals of Python

Dorin Mancu
dmancu@memiq.ro

memIQ
Progress by Education

Objectives

Learn the basic characteristics of the Python programming language

Use Python to solving several practical problems

Recommended Bibliography

- Learning Python, Fifth Edition, Mark Lutz, 2013, O'Reilly, ISBN-978-1-449-35573-9
- Programming Python, Fourth Edition, Mark Lutz, 2011, O'Reilly, ISBN: 978-0-596-15810-1

Contents

Introduction

- Python (’91, Guido van Rossum)– programming language used both for standalone programs and scripting applications in a wide variety of domains.
- Free, portable, expressive
- Monty Python’s Flying Circus (British comedy group)
- import this – the Zen of Python
- Implementations: Jython, IronPython, PyPy, Stackless
- Versions 2.x și 3.x – parallel versions
  - starting with version 3.0 (2008), a new branch not 100% compatible with version 2.x
  - version 3.x: Unicode, accent on iterators and generators, new style for classes, changes to standard libraries
  - 2.x – legacy, stable, still heavily used, de facto standard

Python characteristics:

- General programming language, support for procedural, object oriented & functional programming paradigms
- Code quality – readable, maintainable
- Expressiveness: 20 – 35% of Java or C++ code
- Portability – based on Python Virtual Machine (PVM)
- Standard or third party libraries
- Integration with other languages (C, C++, Java, C#) and technologies (.NET, COM, CORBA, SOAP, etc.)
- Interpreted language → major problem – speed!
- The language permanently evolves (last versions are 3.6.2 & 2.7.13)

Python is used in many places:

http://www.python.org/about/success

http://www.python.org/about/apps

Python is used for:

- Utilities and tools for system administration (shell tools), programs which access the OS resources
- Graphical interfaces – GUI: tkinter, wxPython, PyQT, etc.
- Internet scripting: socket based communication, ftp, mail, Django (web development framework)
- Component integration – glue language
- Database programming
- Building of prototypes – proof of concepts
- Scientific & mathematical calculus – libraries NumPy, SciPy
- Gaming, image processing, data mining, robots

- Interpreter >python nume.py

  Source → byte code (*.pyc) → Python Virtual Machine
- Alternative: frozen binaries (includes PVM and byte code in the same application) – py2exe, PyInstaller, py2app, freeze, cx_freeze
- JIT – just-in-time compiler used by PyPy
- How to run Python code:
  - interactive; prompt >>>
  - IDLE
  - IDE: Eclipse & PyDev, NetBeans IDE for Python, PyCharm

Install Python

Official site: python.org

>python –V # report the version

Windows launcher:

>py -3 hello.py

Sources: extension py, pyw (Windows) or none (Unix)

See hello.py:

```
#!/usr/bin/env python    #!/usr/bin/python
print("Hello", "World!")
```

Quick start…

Data types

- Built-in, offered by the language
- Built by others (standard library)
- Our types (class)

Examples: int, float, str

Immutable types

Conversions between types: int('12')

References to objects

```
x = 1
type(x)       # built-in function
```

Collections

Tuple ()

List []

len() – number of elements for any collection

Operations

Methods

aList.append(“one”)

list.append(aList, “one”)

Operator [] – access by position

Identity test operator – **is**

a is b # True if both refer to the same object

Comparison operators – compare the values

< <= > >= == != (Example: 0 < a < 1)

Membership test operator – **in**

- sequences & collections

Logical operators – and or not

Statements

- Control flow
- Indentation
- Suite – block of statements; **pass**
- Convention for **False**: None, 0, empty collection or sequence

if – elif – else

while, break, continue

for…in

Exceptions

```
try:
        suite
except Exception1 as var1:
        suite
```

Example:

```
s = input("give an integer: ")
try:
        i = int(s)
        print(i)
except ValueError as err:
        print(err)
```

Arithmetic operators

+ - * / // += -=

+ += - overloaded for str & list

Input / output

Examples:

sum1.py

sum2.py < sum2.dat

Functions

def name(argumets):

suite

return, implicit None

Modules

See: echoargs.py

Example:

```
import random
x = random.randint(1,4)
y = random.choice(["home", "table", "man"])
```

Exercises

- Guess a random generated integer
- Remember the given integers then display them sorted
- Interactive program, menu based, which manages strings: add, show, sort, remove

Data Types

dir(), help()

int - literals – prefix 0b, 0o, 0x

float, complex, decimal.Decimal

str – Unicode

- ‘’’ ‘’’’ – multiline string
- Escape codes
- Raw strings r” ” # raw
- [], IndexError
- Negative indexes!
- Slice operator s[start:end:step]
- format() “{0} is {1}”.format(“John”, 5)

Collections

Sequence types:

- Support for membership operator **in**
- len()
- Slices []
- Iterable

Examples: bytes, bytearray, list, tuple, str

```
a, b = 1, 2                    a, b = b, a
return a, b, c
for x, y in ((1,1), (2,3), (4,5)):     # tuple unpacking
        print(x,y)
 MIN, MAX = (0, 100)
```

collections.namedtuple

```
Sale = collections.namedtuple("Sale",
        "productid customerid date quantity price")
sales = []
sales.append(Sale(432, 921, "2010-12-14", 3, 7.99))
sales.append(Sale(419, 874, "2010-12-15", 1, 18.49))

sales[0][-2]        # quantity

total = 0
for sale in sales:
        total += sale.quantity * sale.price
print("Total ${0:.2f}".format(total))
```

list

Mutable, del

```
first, *rest = [1, 2, 3, 4, 5]
first ← 1
rest ← [2, 3, 4, 5]

def func(a, b, c):
        return a * b * c;
lista = [1, 2, 3]
func(*lista)                    # starred expression

for i in range(len(aList)):
        aList[i] = process(aList(i))
```

l[2:4] = []

l.sort(key = str.lower)

List comprehension

[*expression* for *item* in *iterable*]

[*expression* for *item* in *iterable* if *condition*]

Example:

```
leaps = [y for y in range(2000, 2020)
    if (y % 4 == 0 and y % 100 != 0) or (y % 400 == 0)]
```

Sets

- Supports **in** operator
- size()
- Iterable

Built-in: set, frozenset

Contains no duplicates

Items in set are hashable, i.e. they have:

- __hash__() # return the same int during the # object's lifetime
- __eq__() # support for ==

Set comprehension

{*expression* for *item* in *iterable*}

{*expression* for *item* in *iterable* if *condition*}

Example:

```
html = {x for x in files
        if x.lower().endswith((".htm", ".html"))}
```

Maps

- Have **in** operator
- len()
- Iterable
- Keys – only hashable objects

dict – unordered collection of pairs key – value

The key is unique

Mutable

```
d1 = dict({"id": 1948, "name": "Washer", "size": 3})
d2 = dict(id=1948, name="Washer", size=3)
d3 = dict([("id", 1948), ("name", "Washer"), ("size", 3)])
d4 = dict(zip(("id", "name", "size"), (1948, "Washer", 3)))
d5 = {"id": 1948, "name": "Washer", "size": 3}
```

```
d[“red”] = 223
del d[“red”]

for item in d.items():
        print(item[0], item[1])

for key, value in d.items():
        print(key, value)

for value in d.values():
        print(value)

for key in d:
        print(key)

for key in d.keys():
        print(key)
```

Dictionary comprehension

{*keyexpression*: *valueexpression* for *key*, *value* in *iterable*}

{*keyexpression*: *valueexpression* for *key*, *value* in *iterable* if *condition*}

Examples:

```
file_sizes =
        {name: os.path.getsize(name) for name in os.listdir(".")}

file_sizes =
        {name: os.path.getsize(name) for name in os.listdir(".")
                                        if os.path.isfile(name)}
```

collections.defaultdict()

Uses a factory function

See: uniquewords1(2).py

collections.OrderedDict

Copy collections

- copy.copy(col) # shallow copy
- copy.deepcopy(col)

Exercises

- Report the files in a directory in descending order of their sizes
- Report the file duplicates found in the given directory

Iterating collections

```
product = 1
for i in [1, 2, 4, 8]:
        product *= i
print(product)          #  64


product = 1
i = iter([1, 2, 4, 8])              # iterator
while True:
        try:
                product *= next(i)
        except StopIteration:
                break
print(product)          #  64
```

Control structures & functions

```
if  bool_expr1:
        suite1
elif bool_expr2:
        suite2
….
elif bool_exprN:
        suiteN
else:
        suiteElse
```

expression1 if *boolean_expression* else *expression2*

Example:

```
offset = 20 if sys.platform.startswith("win") else 10
print(“{0} file{1}”.format((count if count != 0 else “no”),
                           (”s” if count != 1 else ””)))
```

while *boolean_expression*:

    while_suite

else:

    else_suite

Example:

```
def list_find(lst, target):
    index = 0
    while index < len(lst):
        if lst[index] == target:
            break
        index += 1
    else:
        index = -1
    return index
```

for *expression* in *iterable*:

    for_suite

else:

    else_suite

Example:

```
def list_find(lst, target):
    for index, x in enumerate(lst):
        if x == target:
            break
    else:
        index = -1
    return index
```

Exceptions

try:
    try_suite
except *exception_group1* as *variable1*:
    except_suite1
…
except *exception_groupN* as *variableN*:
    except_suiteN
else:
    else_suite
finally:
    finally_suite

The class of exception objects has to inherit from BaseException (see the documentation)

Example

```
def read_data(filename):
        lines = []
        fh = None
        try:
                fh = open(filename, encoding="utf8")
                for line in fh:
                        if line.strip():
                                lines.append(line)
        except (IOError, OSError) as err:
                print(err)
                return []                       # finally suite is performed first!
        finally:
                if fh is not None:
                        fh.close()
        return lines
```

Generate exceptions

raise *exception*(*args*)

chain the exceptions:
raise *exception*(*args*) from *original_exception*

rethrows the active exception (in handler) or
throws TypeError if not active exception
raise

User defined exception class

class *exceptionName*(*baseException*): pass

BaseException – usually is Exception

Functions – first class entities

- Global
- Local (nested in other function)
- Lambda (anonymous)
- Method (belongs to an object)

Default values for parameters:

```
def multiply(val, mul = 2):
    return val * mul
```

Pass params by name:

```
multiply(mul = 5, val = 8)
```

Example:

```
def append_if_even(x, lst=[]):  # [] is created when the function is
if x % 2 == 0:                  # defined, not when it is called!
    lst.append(x)
return lst
```

Docstrings

```
def calculateArea(x, y):
    """Calc. aria unui dreptunghi
    ....
    """
    return a*b
```

Show help:

```
help(calculateArea)
```

Sequence unpack operator

```
def product(*args):                    # args is a tuple
        result = 1
        for arg in args:
                result = result * arg
        return result
```

Calls:

```
product(1, 2, 3)
product(1, 2, 3, 5, 6, 7)
```

Example:

```
def sum_of_powers(*args, power=1):
        result = 0
        for arg in args:
                result += arg ** power
        return result
sum_of_powers(1, 2, 3, 4, power=3)
sum_of_powers(1, 2, 3, 4, 6, 7, 8, power=2)
```

Mapping unpacking operator

Parameters:

```
options = dict(paper="A4", color=True)
print_setup(**options)
```

Arguments:

```
def add_person_details(ssn, surname, **kwargs):
    print("SSN =", ssn)
    print(" surname =", surname)
    for key in sorted(kwargs):
        print(" {0} = {1}".format(key, kwargs[key]))
```

Call:

```
add_person_details(2234566, “Smith”)
add_person_details(2234566, “Doe”, age=33, forename=”John”)
```

Example: function which can be called with any parameters

```
def print_args(*args, **kwargs):
        for i, arg in enumerate(args):
                print("positional argument {0} = {1}".format(i, arg))
        for key in kwargs:
                print("keyword argument {0} = {1}".format(key,kwargs[key]))
```

Access to the global variables

```
a = 100
def f():
        global a
        a = 7        # l-value assignment means local!
f()
print(a)  # 100 or 7?

def outer():
        y = 0                       # 2.x:   d = {'y': 0}
        def inner():
                nonlocal y          # only for 3.x!
                y+=1                # 2.x:   d['y'] += 1
                return y            # 2.x:   return d['y']
        return inner
f = outer()
f()             # 1
f()             # 2
```

Lambda functions

lambda params: expression

No cycles, no return

Params – optional

Example:

```
s = lambda x: "" if x == 1 else "s“
print("{0} file{1} processed".format(count, s(count)))
```

Reading & writing files

C style:

- file open
- Text (chars) or binary (bytes)
- Read, write, append, etc.

```
fin = open(filename, encoding="utf8")      # read text
fout = open(filename, "w", encoding="utf8") # write

for line in fin:
        process(line)
fin.close()

with open(filename) as fh:
        for line in fh:
                process(line)
```

Exercises:

- The string management application use a dict to register and call the main functions; add the save & restore options in order to make persistent strings in a file
- Design & implement a management system for personal books in order to keep track:
  - what books do I have
  - record what books I lent, to whom & when
  - search books

Persistence will be done in an csv file or with pickles.

Modules & Packages

Module – Python file with functions & classes for reusing

Package – group of modules (files)

Syntax:

import *importable*

import *importable1*, *importable2*, ..., *importableN*

import *importable* as *preferred_name*

from *importable* import *object* as *preferred_name*

from *importable* import *object1*, *object2*, ..., *objectN*

from *importable* import (*object1*, *object2*, *object3*, *object4*, *object5*, *object6*,..., *objectN*)

from *importable* import *

sys.path – list of directories where the modules are searched

PYTHONPATH – environment variable

Convention: standard module names are lowercase

See: TextUtil.py (doctring, main script, unit testing)

Package

Director with modules and a file __init__.py

Example: in Graphics directory we have Bmp.py, Jpeg.py, Png.py, Tiff.py and an empty file __init__.py

Use: import Graphics.Bmp

image = Graphics.Bmp.load("ex.bmp")

import Graphics.Bmp as Bmp

from Graphics import Bmp

from Graphics import Bmp as picture

In __init__.py we can write:

__all__ = ["Bmp", "Jpeg", "Tiff", "Png"]

Use:

```
from Graphics import * # all modules specified in __all__
image = Jpeg.load("a.jpeg")
```

Classes & Objects - OOP

class *className*(*base_classes*):

suite

See: Shape.py for components

Object creation – phases (dunder functions):

- Space allocation with __new__()
- Initial state with __init__()

Constructor in Circle – we could have:

Point.__init__(self, x, y)

super(self.__class__, self).__init__(x, y)

super().__init__(x, y)

Special methods – names like __len__():

__lt__(self, other)	x < y	Returns True if x is less than y
__le__(self, other)	x <= y	Returns True if x is less than or equal to y
__eq__(self, other)	x == y	Returns True if x is equal to y
__ne__(self, other)	x != y	Returns True if x is not equal to y
__ge__(self, other)	x >= y	Returns True if x is greater than or equal to y
__gt__(self, other)	x > y	Returns True if x is greater than y

__ne__() is automatically generated based on __eq__()

How we can avoid comparing a Point to anything else?

- assert isinstance(other, Point)
- if not isinstance(other, Point): raise TypeError()
- if not isinstance(other, Point): return NotImplemented
  In this case Python will try also other.__eq__(self) – if it returns NotImplemented Python will generate TypeError

Support for built-in operators

- Direct __add__()
- Reversed __radd__()
- In place __iadd__()

Exercise: define a Complex class which can be used like below:

```
v1 + v2
v1 += v2
5 + v
v * 6
v1 == v2, v1 <= v2, …
abs(v)
str, repr
```

Object **repr**esentation – generates a string from which the object can be recreated with eval():

```
p = Shape.Point(3, 9)
repr(p)                          # returns: 'Point(3, 9)'
q = eval(p.__module__ + "." + repr(p))
repr(q)
```

__str__ - generates a string used by humans (for example by print())

Properties – see ShapeAlt.py

Decorators: @property, @radius.setter

"Private" properties

self.__radius – name mangling in _Circle__radius

Static members

- Data – defined in class
- Methods - @staticmethod (don’t get self!)

Example: use a method in a function dispatcher:

```
class A:
    def __init__(self, x):
        self.x = x
    def f(self):
        print('f din A', self.x)

a = A(6)
dispatcher = {1:a.f }      # object + method!
dispatcher[1]()
```

Exercise: – 2D graphical editor with operations:

- Create a Circle
- Create a Rectangle
- Calculate the total area
- Calculate the total perimeter
- Delete the forms which have the area smaller than a given value
- Obtain the forms list intersected by a given point
- Obtain a list of forms ordered by their area

Iterable objects

Exercise: modify Editor to be iterable, in order to use to obtain the forms.

- Define method iter()
- Class iterator which implements next()

Note: the alternative is to implement __getitem__() (maybe also __len__()) --> slicing definition on Editor

Hash-able objects

Exercise: make the forms hash-able objects in order to put them in sets or use them as keys in maps:

1. __hash__()
2. __eq__()

Functors – class with method __call__()

Exercise: sort on any atribute, in any order, a list of Person (FirstName, LastName, Age, etc.).

Suggestion: the sort key can be a functor object