Fundamentals of Python

Ciprian Stoica
ciprian.stoica@memiq.ro

Objectives

Learn the basic characteristics of the Python programming language

Use Python to solving several practical problems

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: CPython, 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
  - 2.x – legacy, stable, still heavily used, de facto standard supported until 2020! (http://python3porting.com/)

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!

Python is used in many places:

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

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

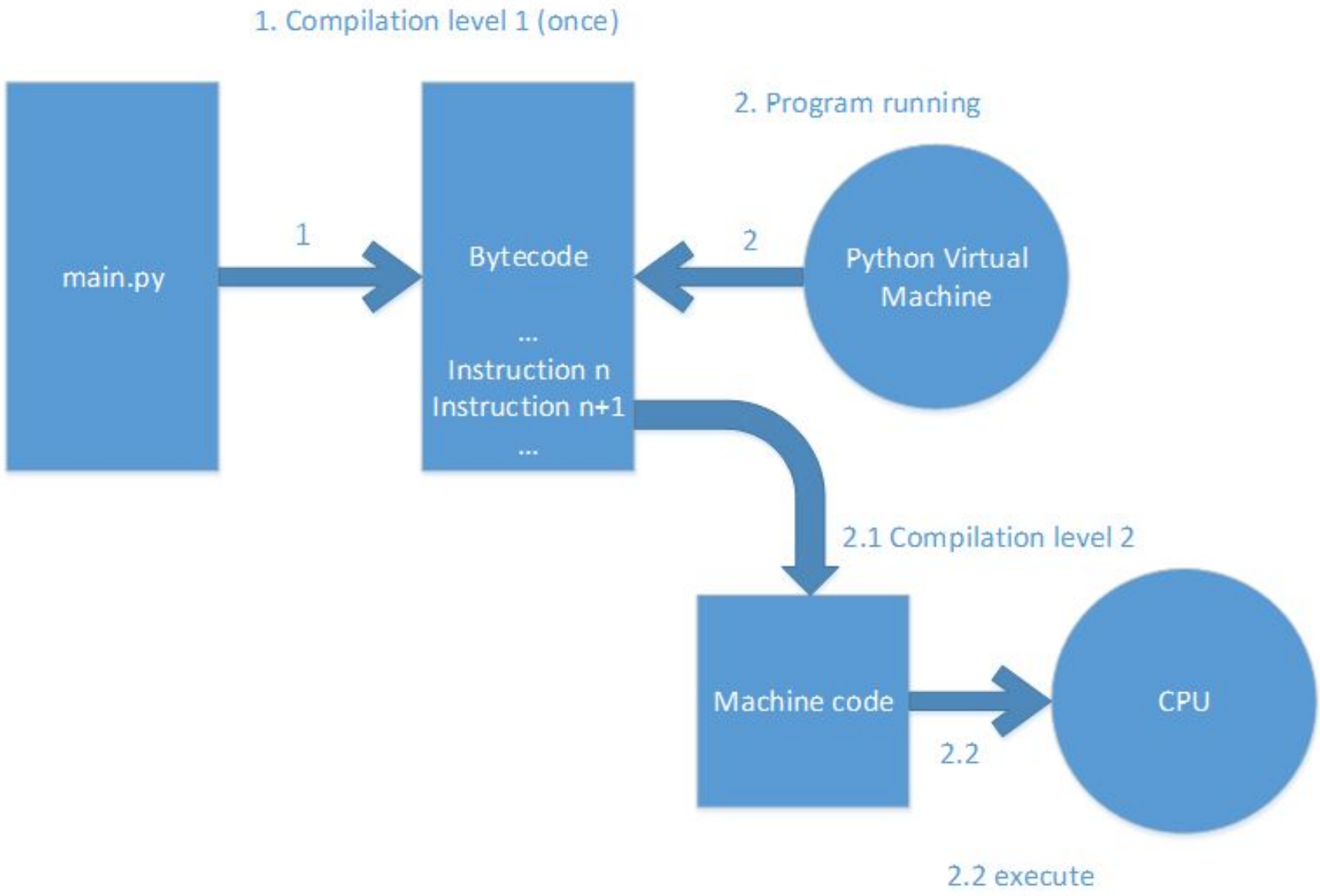
Python is used for:

- Utilities and tools for system administration (shell tools), programs which access the OS resources
- Graphical interfaces – GUI: PyQT, tkinter, wxPython, 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
- How to run Python code:
  - interactive; prompt >>>
  - IDLE
  - IDE: Eclipse & PyDev, NetBeans IDE for Python, PyCharm

>python main.py



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: boolean, int, float, string, list, tuple

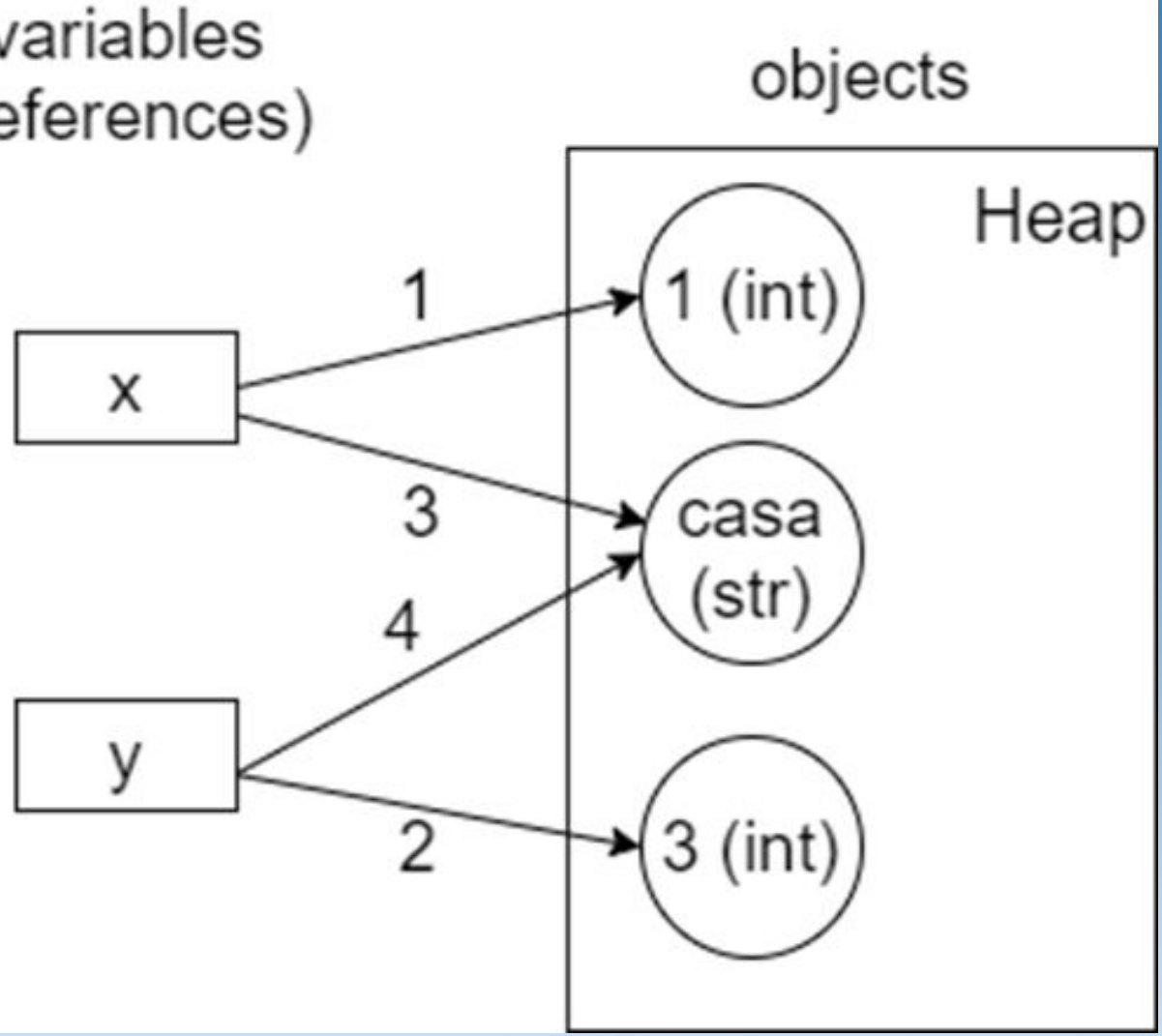
Immutable vs. mutable types

Conversions between types: int('12')

References to objects

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

```
x = 1          # 1
y = 3          # 2
x = 'casa'     # 3
y = x          # 4
```



Collections

tuple ()

list []

len() – number of elements for any collection

Operations

Methods

aList.append("one")

list.append(aList, "one")

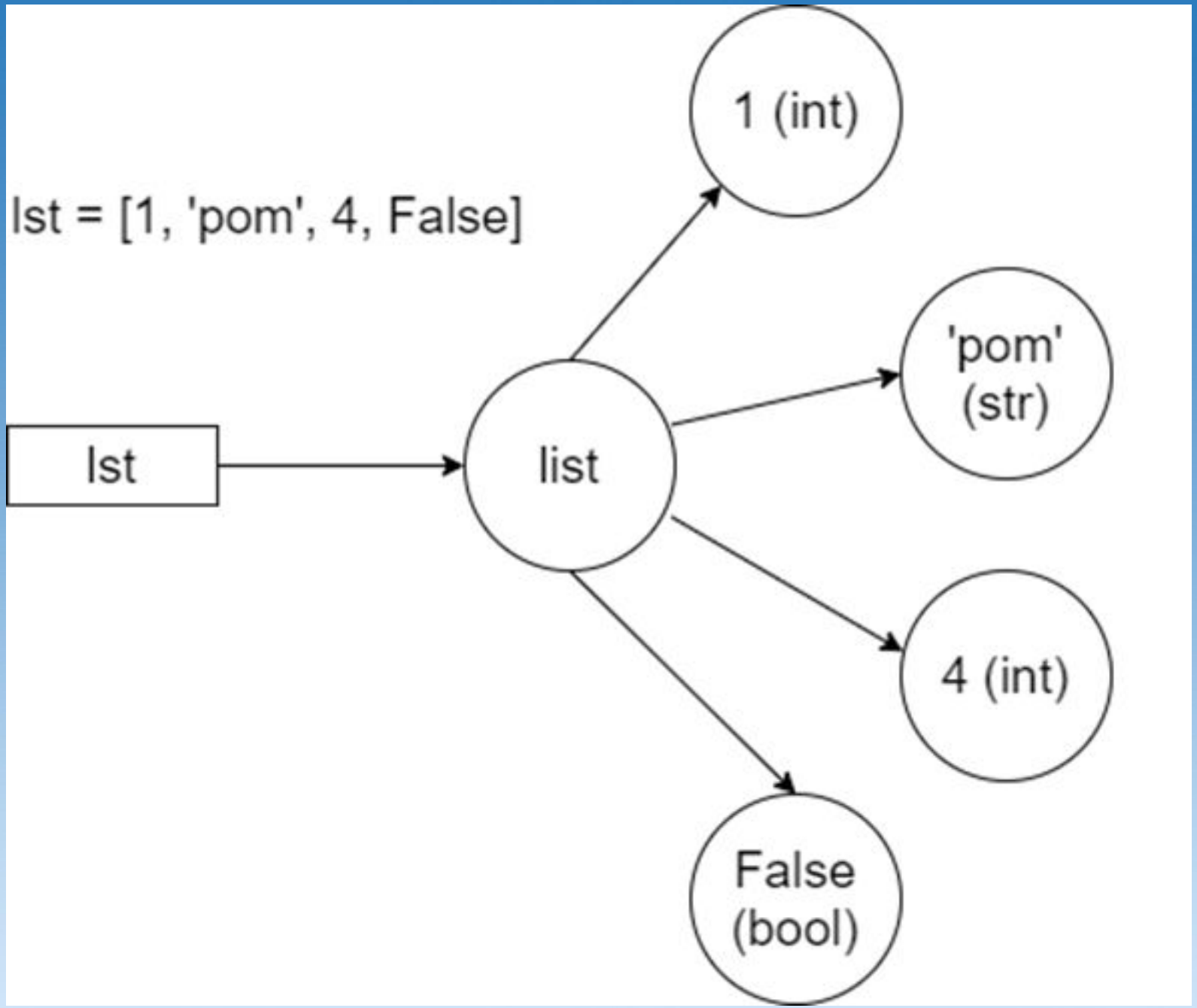
Operator [] – access by position

Identity test operator – **is**

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

Comparison operators – compare the values

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



Operators

=, +, -, *, /, //, %, **, +=, -=, *=, :=, v++

is, in, not, not in, is not, and, or

<<, >>, |, &, ^, ~

<, <=, >, >=, !=, ==

a < b <= c != d ⇔ a < b and b <= c and c != d

https://docs.python.org/3/reference/expressions.html#operator-precedence

Dynamically typed vs. statically typed
Strong vs. weak typing, strongly typed vs. weakly typed (loosely typed)

Python statements

```
if condition1:
    # do something
elif condition2:
    # do something else
elif condition3:
     pass
else:
    # do something else


# ternary operator
d = a if a > b else b  # one comment more
```

Python statements

```
temperature = 25

match temperature:
    case 0:
        print('Cold')
    case 15:
        print('Cool')
    case 35:
        print('Warm')
    case _:
        print('Unknown')
```

Python statements

```
while condition:
    # do something
    if condition2:
        break
    if condition3:
        continue
    # do something
else:
    # do something only if there was no break

# nu avem do while
```

Python statements

```
s = ['a', 'b', 'c', '']
for e in s:
    # do something
    if condition:
        break
    if condition:
        continue
else:
    # do something only if there was no break

for i in range(start, stop, step):
    print(s[i])
```

Python statements

```
s = [‘a’, ‘b’, ‘c’]

for i, e in enumerate(s):
    print(i, e) # both index and element


for i, e in enumerate(s, start=1):
    print(i, e) # both index and element
```

Read data from keyboard

```
v = input('Enter a value:')
print(type(v))  # always a string
```

Print

```
print(12, 'abc', [5, True])
print(12, 'abc', [5, True], sep=' # ', end='')
```

String operations

```
# split, join
str = 'some string here'
s = str.split()
s = str.split('#')
'!!'.join(s)
s.strip() # remove whitespaces
s.strip('ab') # rstrip, lstrip

# multiline string
str = """aaa
    bbb
    cccc"""
str = “aaa\nbbb\nccc”
```

Slicing

```
s = 'abcd'
l = ['a', 'b', 'c', 'd']
t = ('a', 'b', 'c', 'd')

print(s[1:4:2])
print(l[1:4:2])
print(t[1:4:2])
```

l[::] == l -> True

l[::] is l -> False

l[::-1] -> reverse structure

```
sl = slice(2, 4)
print(l[sl])
```

File operations

```
my_file = open("file.txt", "w", encoding='utf-8' )
my_file.write("first line\n")
my_file.write("second line\n")
my_file.close()


with open("file.txt", "a", encoding='utf-8' ) as my_file:
    my_file.write("first line\n")
    my_file.write("second line\n")
    print('3rd line', file=my_file)


with open("file.txt", "r", encoding='utf-8' ) as my_file:
    for line in my_file:
        print(line)
    v = my_file.read()
    v = my_file.readlines()
```

Sets

- unique values
- non-indexable

```
s = {‘a’, ‘cd’, ‘b'}
s.add(‘d’)
s.add(‘b’)

s[1] # error

for e in s:
    pass
```

```
{3,5} - {7,5}
{3,5} | {7,5}
{3,5} & {7,5}
{3,5} ^ {7,5}
```

Dictionaries

- key -> value pairs
- non-indexable
- key only hashable objects

```
d = {
        'd' : 5,
        7 : (‘ac’, 4.5)
    }

d[7]
s = {}
```

Sequences and Collections

- Sequences - string, list, tuple
- deterministic ordering

- Collections - set, dict
- non-deterministic ordering

Functions

```
def func():
   print('Hello World')

def func2(p):
   print('Hello ' + p)

def func3(p):
   return 'Hello ' + p

func()
# None, function doesn't return anything
v = func2('Dolly')

v = func3('Dolly') # Hello Dolly
```

Passing params by name

```
def media(a, b, c):
   return (a + b + c) / 3

media(5, 2, 3) # ok
media(b=5, c=2, a=3) # ok
media(c=2, a=3) # NOT ok
media(4, c=2, a=3) # NOT ok
media(4, c=2, b=3) # ok
```

Parameter default values

```
# ok; some default values and some not
def media(a, b=2, c=3):
    return (a + b + c) / 3

media(4, 5, 6) # ok
media(4, 5) # ok
media(4) # ok
media() # NOT ok

# not ok
def media(a=2, b, c=3):
    return (a + b + c) / 3
```

Variable number of parameters

```
# accepts variable number of positional
arguments
def media(*args):
   print(args)
   print(type(args)

media() # ok
media(3, 'ab') # ok
media([34, 9], True, 'abcd') # ok
media(a=7, b=9)  # Not ok

def media(*args):
   if len(args) == 0:
      return 0
   return sum(args) / len(args)
```

Variable number of parameters

```
# accepts variable number of keyword arguments
def func(**kwargs):
    print(kwargs)
    print(type(kwargs)

func() # ok
func(34, 'a') # NOT ok
func(b=34, d='a') # ok
```

Variable number of parameters

```
# accepts variable number of
# positional arguments and keyword arguments
def func(*args, **kwargs):
   print(args)
   print(kwargs)


func() # ok
func(34, 'a', b=123) # ok
func(b=34, d='a') # ok
func(b=34, 55, d='a') # NOT ok
```

Variable number of parameters

```
# a and b parameters are mandatory
# the others are optional
def func(a, b, *args, **kwargs):
    print(args)
    print(kwargs)


func(5) # NOT ok, missing value for b
func(34, 'QWE', b=123) # NOT ok
func(34, 'QWE', 22, 33, b=123) # NOT ok
func(4, 5, a=7) # NOT ok, multiple value for a
```

Variable number of parameters

```
# a and b parameters are mandatory
# c is a mandatory keyword argument
def func(a, b, *args, c=8, **kwargs):
    print(args)
    print(kwargs)


func(2, 3, 'a', True, c=9, ab=12) # OK

# b must be passed as a keyword only
# every param that follows * must be passed as kw
def func(a, *, b):

func(3, 4) # not OK
func(3, b=4) # OK
func(b=3, a=4) # OK

def func(a=7, *, b): # definition OK
```

Functions - local variables

```
def func():
    x = 7  # x is a local variable
    print('Value of x:', x)  # 7


x = 5
func()
print('Value of x:', x)  # 5
```

Functions - global variables

```
def func():
    global x
    x = 7  # x is a global variable
    print('Value of x:', x)  # 7


x = 5
func()
print('Value of x:', x)  # 7
```

Functions - global variables

```
def func():
    print('Value of x:', x)  # 5
    x = 12  # not ok


x = 5
func()
print('Value of x:', x)  # 5
```

Functions - global variables

```
def func():
    li = [14]  # we made li local variable
    li.append(24)
    print('Value of li:', li)
    di['Q'] = 111


li = [4, 5]
di = {'ab': 123, 'd': 78}
func()
print('Value of li:', li)
print('Value of di:', di)
```

Functions - nonlocal variables

```
def func1():
    def func2():
        nonlocal x
        x = 6
        print(x)

    x = 7
    func2()
    print(x)


x = 5
func1()
print(x)
```

Functions returned from functions

```
def func(x):
    def func2(y):
        return x * y

    return func2

# x is a function
x = func(5)
print(x(7))
```

Recursive functions

```
def factorial(n):  # non-recursive
    t = 1
    for i in range(1, n + 1):
        t *= i

    return t

def factorial(n):  # recursive
    if n == 1:
        return 1
    return n * factorial(n - 1)

print(factorial(5))
```

Lambda functions

```
# short-lived, one expression, anonymous
functions
lambda x, y: x * y
lambda a: a ** 2

r = list(
   map(lambda x, y: x * y,
         ['a', 'b', 'c'],
         [2, 3, 5]
   )
)
print(r)


y = filter(lambda x: x >= 3, (1, 2, 3, 4))
z = filter(None, (-1, 0, 1, '', 2, 3, 4))
```

Formatting

```
x = 'bala'
print(f'ala {x} portocala {x * 2}')


print("{} {}, {}".format('ala', 'bala', 'portocala'))
print("{2} - {0} - {1}".format('ala', 'bala', 'portocala'))
print("{param2} - {param1}".format(param1='ala',
param2='bala'))
print("{1} - {param} - {0}".format('ala', 'bala', param =
'portocala'))
```

Formatting

```
print('%s # %s' % ('ala', 'bala'))
print('%s - %i - %f' % ('ala', 34, 6.8))

print('ala {:>15} portocala'.format('bala'))
print('ala {param:-^15} portocala'.format(param='bala'))
print('## {:7.4f} ##'.format(12.3))
```

Comprehensions

```
l = [x ** 2 for x in range(0, 10, 2)]

s = {x * 2 for x in list('abcde')}

t = tuple(10 / x for x in range(50, 30, -3) if x % 2 == 1)

d = {x : 10 / x for x in range(50, 30, -3) if x % 2 == 1}

d = {x: 'par' if x % 2 == 0 else 'impar' for x in range(10)}
```

Comprehensions

```
d = {x: 'par' if x % 2 == 0 else 'impar' for x in range(10)}

# equivalent

d3 = {}
for x in range(10):
    if x % 2 == 0:
        d3[x] = 'par'
    else:
        d3[x] = 'impar'
```

Regular expressions

```
import re

content = "some content"

print(re.split('pattern', content))
print(re.sub('pattern', 'replace', content))
result = re.search('pattern', content)
result = re.match('pattern', content)
result = re.fullmatch('pattern', content, flags)
if result: print(result.group(0))
result = re.findall('pattern', content)
flags: re.IGNORECASE, re.MULTILINE
```

Expresii regulate (regexp)

- Șiruri de caractere cu o sintaxă specifică folosite pentru căutari complexe în texte

Expresie	Explicatie	Exemplu	Potrivire
*	Zero sau mai multe exemplare din caracterul anterior	hel*o	heo, helo, hellllo
+	Unul sau mai multe exemplare din caracterul anterior	hel+o	helllo heo
.	Orice caracter	h.lo	helo, halo heelo
[]	Orice caracter din setul respectiv	ca[lm]	cal, cam car
[^]	Orice caracter din afara setului	ca[^lm]	car, cap, caL cal. cam

Expresii regulate (regexp)

Expresie	Explicatie	Exemplu	Potrivire (match)
[a-z]	Oricare din intervalul a-z	[a-zT0-8]	f, T, 8 R, 9
x|y	Oricare dintre x sau y	ab|cd	ab, cd bc
?	Zero sau un caracter	x?	nimic, x xx
{2,4}	Intre 2 și 4 caractere	d{2,4}	dd, ddd, dddd d, ddddd
{2,}	Cel puțin 2 caractere	d{2,}	dd, ddd, dddd d
{,3}	Cel mult 3 caractere	r{,3}	, r, rr, rrr rrrr
{5}	Exact 5 caractere	w{5}	wwwww ww
\	Escape	\{|\?	{, ?

Type hinting

```
def func(a: str, b: int) -> str:
    """Mini documentation of func"""
    return a * b

def func2(a: int or list, b: int) -> int or list:
    return a * b

def func3(a: str = 'abc', b: int = 3) -> str:
    return a * b

x: bool = True
```

PEP 8

- PEP - Python Enhancement Proposal
- https://peps.python.org/pep-0008/
- Style guide for Python

```
#code not PEP compliant
def Func(a = 9):
    if (a>8):
     print('not OK')

# code PEP compliant
def func(a=9):
    if a > 8:
        print('OK')
```

Binary file operations

Serialization - the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file.

Deserialization - reverse process - a stream of bytes into an object in memory

Binary file operations

```
import pickle

data = {
            'a': ['hello', 'world'],
            'b' : 123
        }

with open('filename.pickle', 'wb') as file:
    pickle.dump(data, file)


with open('filename.pickle', 'ab') as file:
    pickle.dump(data, file)

with open('filename.pickle', 'rb') as file:
    data2 = pickle.load(file)
```

Pretty printer

```
import pprint

data = { 'a': ['hello', 'world'],
         'b' : 123
         }

pprint.pprint(data)
```

Modules

```
import mymodule, mymodule2
import folder.subfolder.module as alias
import mymodule as othermodule
from mymodule import my_func, my_list
from mymodule import my_func as other_func
from mymodule import *

def my_func():
    pass

mymodule.my_func()
my_func()
```

Modules

Additional modules on https://pypi.org/
install additional modules:

pip install <module-name>==version

pip install -r requirements.txt

Working with fractions

https://www.google.com/search?q=why+is+0.1%2B0.2+not+equal+to+0.3+in+most+programming+languages

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.551115123125783e-17
>>> 1.1 + 2.2
3.3000000000000003
```

```
from fractions import Fraction
a = Fraction(11, 10)
b = Fraction(22, 10)
c = a + b
print(c, float(c))
```

Working with the filesystem

```
import os, shutil

os.getcwd()
os.chdir()
os.mkdir()
os.rmdir()
shutil.rmtree()
shutil.copy('file1', 'some/path/file2')

os.system('os command')
os.walk()
```

Docstrings

```
# NumPy/SciPy docstring example
def func(a, b):
    """This the documentation title.

    Parameters
    ----------
    a : str
        description of the first param
    b : int
        description of the second param

    Returns
    ----------
    str:
        some description for the return
    """

    return a * b


func('ab', 4)

help(func)
```

Docstrings

```
# reStructured Text docstring
def func2(a, b=1):
    """Some description

    :param a: first parameter
    :type a: str

    :param b: second parameter (default is 1)
    :type b: int

    :returns: some return
    :rtype: str
    """

    return a * b

# alternatives: Epytext, Google
```

Doctests

```
def factorial(n):
    """
    Calculates factorial of n

    Parameters:
        n (int): the input number

    Returns:
         int: the computed factorial

    Example usage:
    >>> factorial(5)
    120
    >>> factorial(7)
    50401
    """

    if n <= 1:
        return 1
    return n * factorial(n - 1)
```

Doctests

```
if __name__ == '__main__':
    from doctest import testmod
    testmod()
    testmod(name='factorial')
    testmod(name='factorial', verbose=True)
```

Command line parameters

```
# test.py
import sys

# print all arguments (parameters)
for p in sys.argv:
   print(p)
```

```
# command line

> python test.py param1 param2

# Linux
> "full/path/to/python" "full/path/to/test.py" 'Dana Popescu'

#Windows
> "C:\Program Files\Python3.9\bin\python"
"full/path/to/test.py" "Dana Popescu"
```

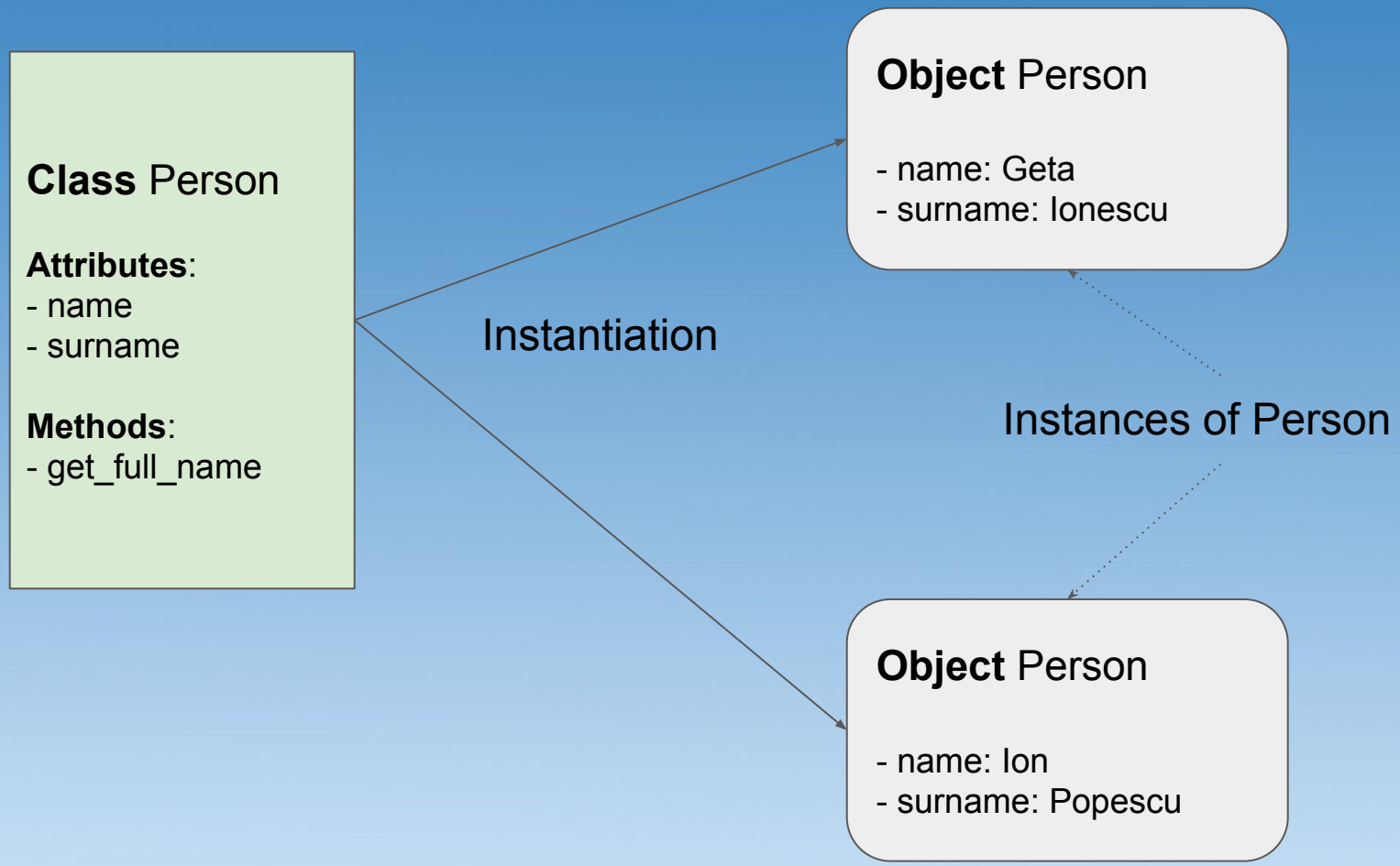
Unpacking operators: *, **

```
def func(a, b, c):
   print(a, b, c)

t = (2, 3, 4)
func(t) # error
func(*t) # ok, like func(t[0], t[1], t[2])

d1 = {'b': 2, 'a': 3, 'c': 4}
d2 = {'b': 2, 'a': 3, 'd': 4}

func(*d1) # ok
func(**d1) # ok
func(b=2, a=3, c=4) # equivalent
func(*d2) # ok
func(**d2) # not ok
```

Unpacking operators: *, **

```
x, y, z = 3, 4, 5 # ok

x, y, z = 3, 4, 5, 6
# ValueError: too many values to unpack

x, y, z = 3, 4
# ValueError: not enough values to unpack

x, y, *z = 3, 4, 5, 6 # ok, z = [5, 6]
x, y, *z = 3, 4 # ok, z = []

x, *y, *z = 3, 4, 5, 6, 7, 8
# SyntaxError: multiple starred expressions in
assignment

(x, *y), *z = (3, 4, 5), 6, 7, 8 # ok
```

Object-Oriented Programming (OOP)



Object-Oriented Programming (OOP)

```
class Person:

    def __init__(self, name, surname):
        """Constructor of the class"""
        self.name = name
        self.surname = surname

    def get_fullname(self):
        return f'{self.name} {self.surname}'


p1 = Person('Geta', 'Ionescu')
p2 = Person('Ion', 'Popescu')
print(p1.get_fullname())
print(Person.get_fullname(p2))
```

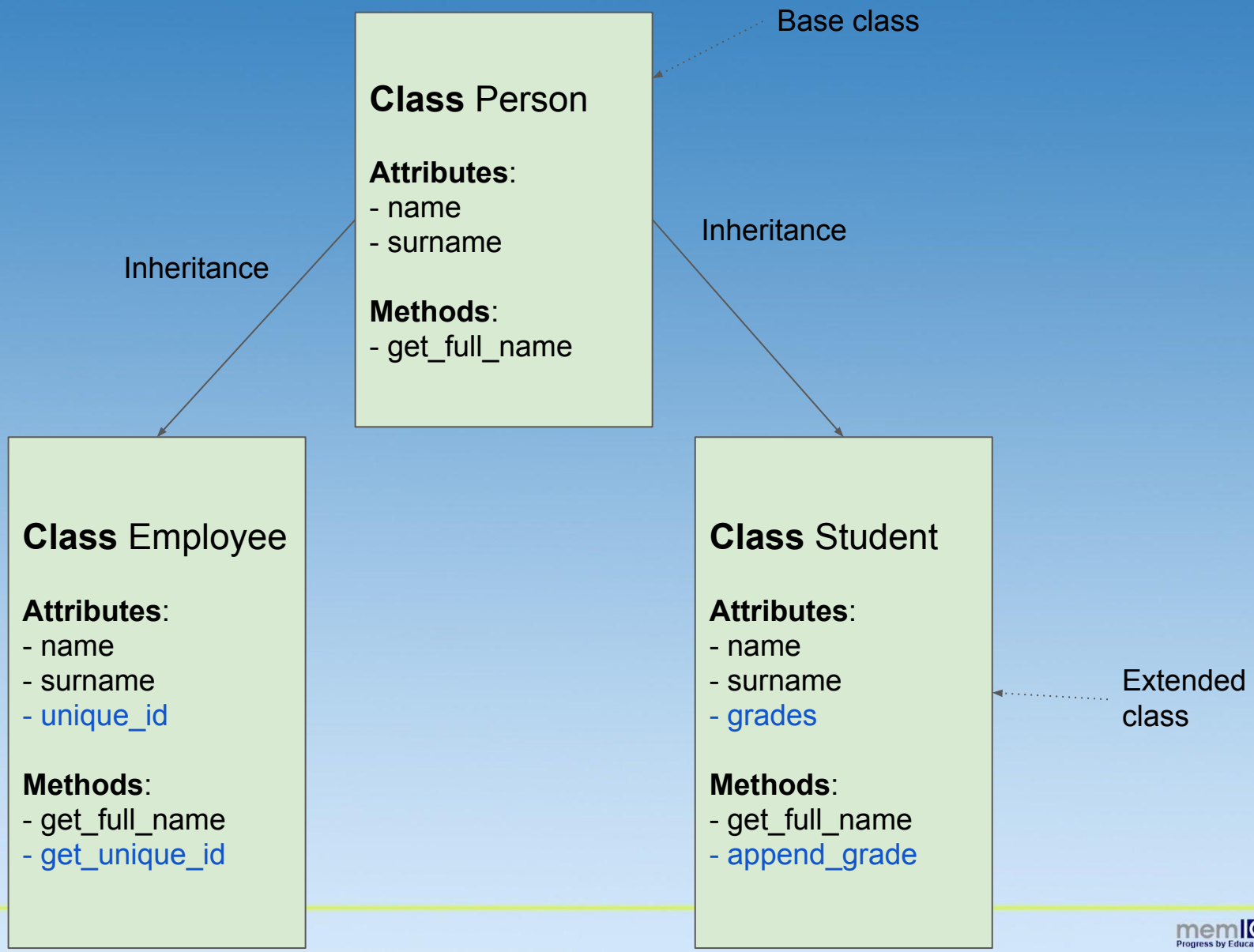
Object-Oriented Programming (OOP)

```
p1 = Person('Geta', 'Ionescu')

# access attributes
print(p1.name)
p1.name = 'Georgeta'

# add attribute on the fly; usually not ok
p1.new_something = 1234

# del attribute on the fly; not ok
del p1.surname
```

OOP - private vs public

```
class Person:

    def __init__(self, name, surname):
        # private attributes
        self.__name = name
        self.__surname = surname

    def get_name(self):
        return self.__name

    def set_name(self, name):
        self.__name = name

    def get_fullname(self):
        return f'{self.__name} {self.__surname}'


p1 = Person('Geta', 'Ionescu')
print(p1.__name)  # error
```

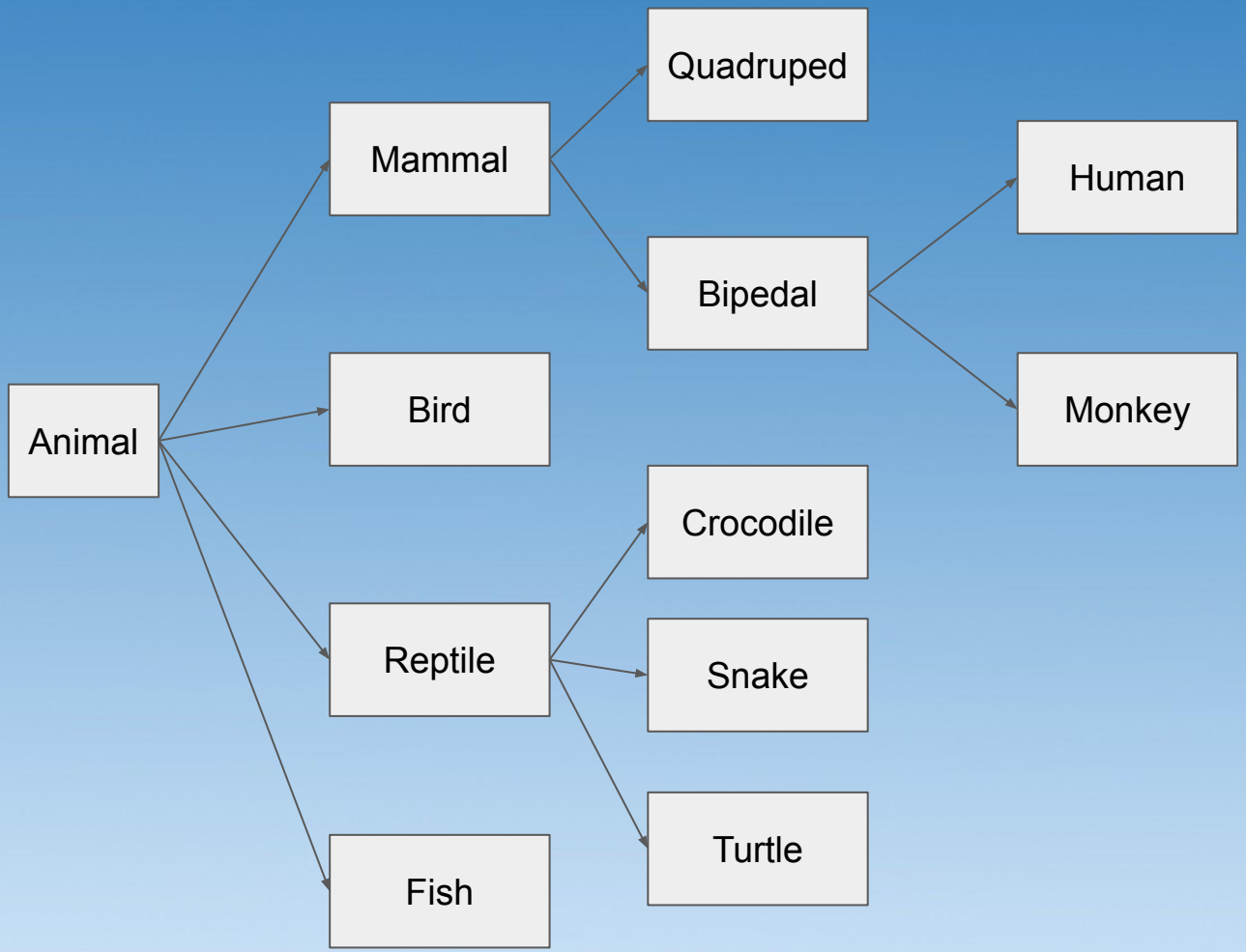
OOP - simple inheritance



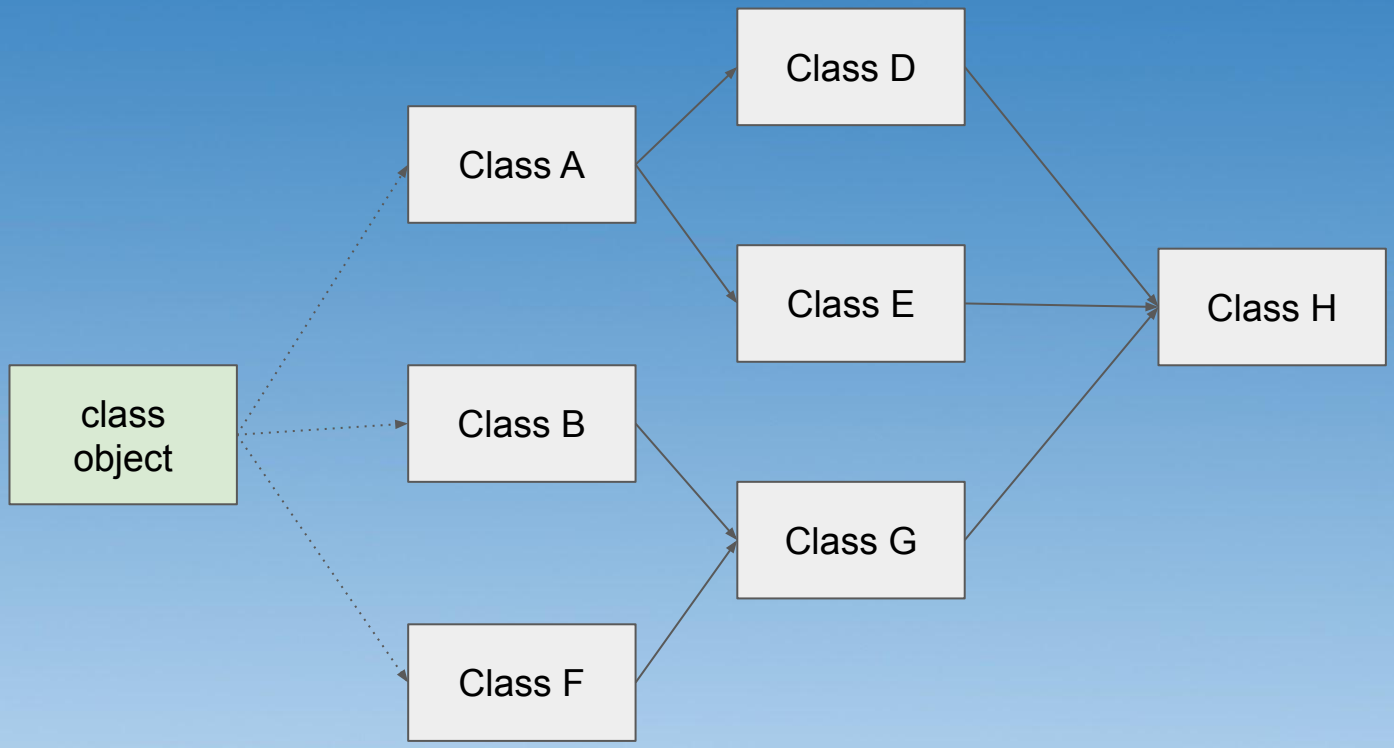
OOP - simple inheritance

```
class Student(Person):

    def __init__(self, name, surname, grades):
        super().__init__(name, surname)
        self.grades = grades

    def get_fullname(self):
        return f'{self.name} {self.surname}'.upper()

    def get_average(self):
        if len(self.grades) == 0:
            return 0.0
        return round(sum(self.grades) / len(self.grades), 2)


p1 = Student('Geta', 'Ionescu', [3, 5, 6, 7])
print(p1.get_fullname())
print(p1.get_average())

print(type(p1))  # Student
print(isinstance(p1, Person))  # True. A Student is also a
Person
```

OOP - simple inheritance



OOP - multiple inheritance



OOP - multiple inheritance

```
class A:

    def generic(self):
        print("AAAA")


class B:

    def generic(self):
        print("BBBB")


class C(A, B):

    def generic2(self):
        B.generic(self)


c = C()
c.generic()
c.generic2()
print(C.mro())  # Method Resolution Order(MRO)
```

OOP - composition

```
class A:
   pass


class B:
   pass


class C:
   def __init__(self, atr1, atr2):
      self.atr1 = atr1
      self.atr2 = atr2

a = A()
b = B()

c = C(a, b)
```

OOP - __repr__ and __str__

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

   def __repr__(self)
      return f'A({self.x}, {self.y})'

   def __str__(self)
      return f'This is object A({self.x},
{self.y})'


a = A(4, 5)
print(a)
```

OOP - destructor

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

   def __del__(self):
      """
      This is the destructor.
      Gets called when the object is destroyed
      """
      pass


a = A(4, 7)
a = 1234
```

OOP - operator overloading

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

   def __lt__(self, other):
      return self.x < other.x


a1 = A(4)
a2 = A(7)

if a1 < a2:  # normally error here
   print('a1 lower than a2')
else:
   print('a2 lower than a1')
```

Exceptions

- an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions
- triggered automatically on errors or by the code

```
l = ['a', 'b']
print(l[4])
# IndexError: list index out of range

n = input('Enter the number:')
print(n ** 2)
# TypeError: unsupported operand type(s) for **
or pow(): 'str' and 'int'
```

- exceptions hierarchy: https://docs.python.org/3/library/exceptions.html#exception-hierarchy

Exceptions handling

```
try:
   l = ['a', 'b']
   print(l[4])
   print('one more line..')
except:
    print("Exception caught!")
else:
    print("No exception caught!")
finally:
    print("Finally done!")

print('Program continues..')
```

Exceptions handling

```
try:
   # some code that might generate exceptions
except LookupError:
   print('LookupError caught')
except IndexError as ex:
   print(type(ex), ex)
except (IndexError, KeyError) as ex:
    print(type(ex), ex)
except Exception:
    print('Everything..')
else:
    print("No exception caught!")
finally:
    print("Finally...")

print('Will this print anything?')
```

Custom exceptions

```
class MyCustomException(LookupError):
   pass

# raise - raises (throws) an exception

raise MyCustomException('some text')

raise IndexError
```

Static class members

```
class A:

    instances = 0

    def __init__(self, x):
        self.x = x
        A.instances += 1

    def __del__(self):
        A.instances -= 1

    @staticmethod
    def show_instances():
        print(f'There are {A.instances} instances so
far..')


a = A(7)
b = A(8)
A.instances
A.show_instances()
```