# Języki skryptowe

dr inż. Marcin Pietroń

# Plan

- Obiektowość w Pythonie
- Dostęp do baz danych
- Wielowątkowość
- Programowanie sieciowe
- Xml processing
- Popularne biblioteki i narzędzia (scikit learn, numpy, scipy)

# Object oriented programming

- Object languages:
  - Java (business applications, mobile software etc.)
  - C++ (embedded, real-time, finance-sector)
  - Python (prototyping, testing)
  - Smalltalk
  - Ruby (web)
  - Objective-C (mobile)
  - …

# Object oriented programming

- programming paradigm - data structures called "objects", contain data, in the form of fields, often known as *attributes;* and code, in the form of procedures, often known as *methods.*

# Object oriented programming

- a feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated,
- in OO programming, objects in a software system interact with one another.

# Object oriented programming

- There is significant diversity in object-oriented programming, but most popular languages are class-based, meaning that objects are instances of classes, which typically also determines their type.

# Object oriented programming

Languages that support object-oriented programming use inheritance for code reuse and extensibility in the form of either classes or prototypes. Those that use classes support two main concepts:

- Objects - structures that contain both data and procedures
- Classes - definitions for the data format and available procedures for a given type or class of object; may also contain data and procedures (class methods)

# Object oriented programming

- Objects sometimes correspond to things found in the real world (e.g.  a graphics program may have objects such as „figure ", "circle ", "square" or " cursor ", online shop may have objects such as "shopping cart" "customer" " basket " or "product ",
- Objects can represent more abstract things and entities, like an object that represents an open file, or an object which provides some service (e.g. xml parsing)

# Object oriented programming

Object is an instance of a particular class. Procedures in object-oriented programming are known as methods, variables are also known as fields, members, attributes or properties.

- **Class variables** - belong to the *class as a whole*; there is only one copy of each one
- **Instance variables or attributes** - data that belongs to individual *objects*; every object has its own copy of each one
- **Member variables** - refers to both the class and instance variables that are defined by a particular class
- **Class methods** - belong to the *class as a whole* and have access only to class variables and inputs from the procedure call
- **Instance methods** - belong to *individual objects*, and have access to instance variables for the specific object they are called on, inputs, and class variables

# Object oriented programming

- „Pure" OO languages, because everything in them is treated consistently as an object, from primitives such as characters and punctuation, all the way up to whole classes, prototypes, blocks, modules, etc.
  Examples: Eiffel, Obix, Ruby, Scala, Smalltalk, Self.
- Languages designed mainly for OO programming, but with some procedural elements. Examples: Delphi/Object
  Pascal, C++, Java, C#, VB.NET.
- Languages that are historically procedural languages, but have been extended with some OO features. Examples: Pascal, Visual Basic (derived from BASIC), Perl, COBOL 2002, PHP, Ada 95.
- Languages with abstract data type support but without all features of object-orientation. This includes object-based and prototype-based languages. Examples: Modula-2, JavaScript, Lua.
- Chameleon languages that support multiple paradigms, including OO, hybrid object system that supports both prototype-based programming and class-based OO (Tcl).

# Object oriented programming

- Dynamic dispatch/message passing
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

# Dynamic dispatch/message passing

- by definition, it is the responsibility of the object, to 'on-demand' select the procedural code to run/execute in response to a method call, by looking up the method at run time in a table associated with the object,
- multiple dispatch - if there are multiple methods that might be run for a given name (this process require some language support).

# Python object programming

- **Class:** A user-defined prototype for an object that defines a set of attributes that characterize any object of the class.
- **Class variable:** A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods.
- **Data member:** A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading:** The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.

# Python object programming

- **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance:** Reusing characteristics of a class to other classes that are derived from it.
- **Instance:** An individual object of a certain class.
- **Instantiation:** The creation of an instance of a class.
- **Method :** A special kind of function that is defined in a class definition.
- **Operator overloading:** The assignment of more than one function to a particular operator.

# Definicja klasy

```
class ClassName:
    'class documentation string'
     class_suite
```

*documentation string* accessed by:
*ClassName.__doc__*

# Composition

- Objects can contain other objects in their instance variables,
- For example, an object in the Employee class might contain (point to) an object of the Address class, in addition to its other own instance variables,
- Object composition is used to represent "has-a" relationships: every employee has an address, so every Employee object has a place to store an Address object.

# Inheritance

- Languages that support classes almost always support inheritance.
- This allows classes to be arranged in a hierarchy that represents "is-a-type-of" relationships.
- For example, class Employee might inherit from class Person. All the data and methods available to the parent class also appear in the child class with the same names.
- For example, class Person might define variables "first_name" and "last_name" with method "make_full_name()". These will also be available in class Employee, and with additional variables "position" and "salary".
- This technique allows easy re-use of the same procedures and data definitions, with similar real-world relationships,

# Inheritance

- Subclasses can override the methods defined by superclasses.
- Multiple inheritance is allowed in some languages (this can make resolving overrides complicated),
- Some languages have special support for mixins, though in any language with multiple inheritance, a mixin is simply a class that does not represent an is-a-type-of relationship. Mixins are typically used to add the same methods to multiple classes. For example, class UnicodeConversionMixin might provide a method unicode_to_ascii() when included in class FileReader and class WebPageScraper, which don't share a common parent.

# Inheritance vs Composition

- The doctrine of composition over inheritance advocates implementing is-a-type-of relationships using composition instead of inheritance.
- For example, instead of inheriting from class Person, class Employee could give each Employee object an internal. Some languages, do not support inheritance at all.

# Python constructors

```
class Person:
   'Common base class for all employees'
    def __init__(self, firstName, lastName):
        self.fname = firstName
        self.lname = lastName
```

# __init__ method

class constructor or initialization method that Python calls when new instance of class is created.

person = Person(„Jan", „Kowalski")

# Implementacja klasy

```
class Person:
   'Common base class for all employees'
    count = 0
    def __init__(self, firstName, lastName):
        self.fname = firstName
        self.lname = lastName
    def printPerson(self):
        print „First name : ", self.fname, ",Last name: ", self.lname
```

# Inicjalizacja

person = Person(„Jan”, „Kowalski”)

person.printPerson()

# Usuwanie obiektów

- garbage collector

# Usuwanie obiektów

```
def __del__(self):
    class_name = self.__class__.__name__
    print class_name, "destroyed"
```

# Usuwanie obiektów

```
pt1 = Point()
pt2 = pt1
pt3 = pt1
del pt1
del pt2
del pt3
```

# Inheritance

```
class Parent:
   parentAttr = 100
   def __init__(self):
         print "Calling parent constructor"
   def parentMethod(self):
        print 'Calling parent method'
   def setAttr(self, attr):
         Parent.parentAttr = attr
    def getAttr(self):
          print "Parent attribute :", Parent.parentAttr

class Child(Parent): # define child class
     def __init__(self):
            print "Calling child constructor"
      def childMethod(self):
             print 'Calling child method'
             c = Child() # instance of child

c.childMethod() # child calls its method
c.parentMethod() # calls parent's method
c.setAttr(200) # again call parent's method
c.getAttr() # again call parent's method
```

# Inheritance

```
class A:
…
class B:
…
class C(A, B):
…
```

# Inheritance

- **issubclass(sub, sup)** boolean function returns true if the given subclass **sub** is indeed a subclass of the superclass **sup**.
- **isinstance(obj, Class)** boolean function returns true if *obj* is an instance of class *Class* or is an instance of a subclass of Class

# Overriding

```
class Parent: # define parent class
    def myMethod(self):
        print 'Calling parent method'
class Child(Parent): # define child class
    def myMethod(self):
        print 'Calling child method'

c = Child() # instance of child
c.myMethod() # child calls overridden method
```

# Generic function (to be overload)

| | |
|---|---|
| 1 | **__init__ ( self [,args...] )**<br>Constructor (with any optional arguments)<br>Sample Call : *obj = className(args)* |
| 2 | **__del__( self )**<br>Destructor, deletes an object<br>Sample Call : *del obj* |
| 3 | **__repr__( self )**<br>Evaluatable string representation<br>Sample Call : *repr(obj)* |
| 4 | **__str__( self )**<br>Printable string representation<br>Sample Call : *str(obj)* |
| 5 | **__cmp__ ( self, x )**<br>Object comparison<br>Sample Call : *cmp(obj, x)* |

# Overload operators

```
class Vector:
    def __init__(self, a, b):
        self.a = a self.b = b
    def __str__(self):
        return 'Vector (%d, %d)' % (self.a, self.b)
     def __add__(self,other):
        return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print v1 + v2
```

# Encapsulation

```
class JustCounter:
    __secretCount = 0
    def count(self):
        self.__secretCount += 1
        print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.__secretCount
```

# Static and class methods

```
@staticmethod
def the_static_method(x):
    body of method

@classmethod
def from_string(cls, date_as_string):
    body of method
```

In class method cls parameter is mandatory

# Mechanizm retrospekcji

- The **getattr(obj, name[, default])** : to access the attribute of object.
- The **hasattr(obj,name)** : to check if an attribute exists or not.
- The **setattr(obj,name,value)** : to set an attribute. If attribute does not exist, then it would be created.
- The **delattr(obj, name)** : to delete an attribute.

# Mechanizm retrospekcji

- **__dict__:** Dictionary containing the class's namespace.
- **__doc__:** Class documentation string or none, if undefined.
- **__name__:** Class name.
- **__module__:** Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- **__bases__:** A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

# Programowanie sieciowe

| | |
|---|---|
| domain | The family of protocols that is used as the transport mechanism. These values are constants such as AF_INET, PF_INET, PF_UNIX, PF_X25, and so on. |
| type | The type of communications between the two endpoints, typically SOCK_STREAM for connection-oriented protocols and SOCK_DGRAM for connectionless protocols. |
| protocol | Typically zero, this may be used to identify a variant of a protocol within a domain and type. |
| hostname | •The identifier of a network interface:A string, which can be a host name, a dotted-quad address, or an IPV6 address in colon (and possibly dot) notation<br>•A string "<broadcast>", which specifies an INADDR_BROADCAST address.<br>•A zero-length string, which specifies INADDR_ANY, or<br>•An Integer, interpreted as a binary address in host byte order. |
| port | Each server listens for clients calling on one or more ports. A port may be a Fixnum port number, a string containing a port number, or the name of a service. |

# Programowanie sieciowe

s = socket.socket (socket_family, socket_type, protocol=0)

- **socket_family:** This is either AF_UNIX or AF_INET, as explained earlier.
- **socket_type:** This is either SOCK_STREAM or SOCK_DGRAM.
- **protocol:** This is usually left out, defaulting to 0.

# Programowanie sieciowe

| | |
|---|---|
| s.bind() | This method binds address (hostname, port number pair) to socket. |
| s.listen() | This method sets up and start TCP listener. |
| s.accept() | This passively accept TCP client connection, waiting until connection arrives (blocking). |

# Programowanie sieciowe

| | |
|---|---|
| s.connect() | This method actively initiates TCP server connection. |

# Programowanie sieciowe

| | |
|---|---|
| s.recv() | This method receives TCP message |
| s.send() | This method transmits TCP message |
| s.recvfrom() | This method receives UDP message |
| s.sendto() | This method transmits UDP message |
| s.close() | This method closes socket |
| socket.gethostname() | Returns the hostname. |

# Programowanie sieciowe

```
import socket
s = socket.socket()
host = socket.gethostname()
port = 12345
s.bind((host, port))
s.listen(5)
while True:
    c, addr = s.accept()
    print 'Got connection from', addr
    c.send('Thank you for connecting')
    c.close() # Close the connection
```

# Programowanie sieciowe

```
import socket
s = socket.socket()
host = socket.gethostname()
port = 12345
s.connect((host, port))
print s.recv(1024)
s.close
```

# XML processing

**Simple API for XML (SAX) :** Here, you register callbacks for events of interest and then let the parser proceed through the document. This is useful when your documents are large or you have memory limitations, it parses the file as it reads it from disk and the entire file is never stored in memory.

**Document Object Model (DOM) API :** This is a World Wide Web Consortium recommendation wherein the entire file is read into memory and stored in a hierarchical (tree-based) form to represent all the features of an XML document.

# XML

```
<movie title=„James Bond, Skyfall">
    <type>Thriller</type>
    <format>DVD</format>
    <year>2012</year>
</movie>
```

# XML DOM

```
from xml.dom.minidom import parse
import xml.dom.minidom

def replaceText(node, newText):
    if node.firstChild.nodeType != node.TEXT_NODE:
        raise Exception("node does not contain text")
    node.firstChild.replaceWholeText(newText)
```

# XML DOM

```
DOMTree = xml.dom.minidom.parse("t.xml")
collection = DOMTree.documentElement
```

# XML DOM

```
movies = collection.getElementsByTagName("movie")

for movie in movies:
    print "*****Movie*****"
    if movie.hasAttribute("title"):
        print "Title: %s" % movie.getAttribute("title")

    type = movie.getElementsByTagName('type')[0]
    print "Type: %s" % type.childNodes[0].data
    format = movie.getElementsByTagName('format')[0]
```

# XML DOM

replaceText(type, „type test”)

file_handle = open("filename.xml","wb")
collection.writexml(file_handle)
file_handle.close()

# XML SAX

```
import xml.sax
class MovieHandler( xml.sax.ContentHandler ):
    def __init__(self):
        self.type = ""
        self.format = ""
        self.year = ""
```

# XML Sax

```
def startElement(self, tag, attributes):
    self.CurrentData = tag
    if tag == "movie":
        print "*****Movie*****"
        title = attributes["title"]
        print "Title:", title
```

# XML Sax

```
def endElement(self, tag):
    if self.CurrentData == "type":
        print "Type:", self.type
    elif self.CurrentData == "format":
        print "Format:", self.format
    elif self.CurrentData == "year":
        print "Year:", self.year
```

# XML Sax

```
def characters(self, content):
    if self.CurrentData == "type":
        self.type = content
    elif self.CurrentData == "format":
        self.format = content
    elif self.CurrentData == "year":
        self.year = content
```

# XML

```
<root>
    <doc>
        <field1 name="blah">some value1</field1>
        <field2 name="asdfasd">some vlaue2</field2>
    </doc>
</root>
```

# XML

```
import xml.etree.cElementTree as ET

root = ET.Element("root")

doc = ET.SubElement(root, "doc") ET.SubElement(doc, "field1", name="blah").text = "some value1"

ET.SubElement(doc, "field2", name="asdfasd").text = "some vlaue2"

tree = ET.ElementTree(root) tree.write("filename.xml")
```

# Multithreading

```
import thread
import time
def print_time( threadName, delay):
    count = 0
    while count < 5:
        time.sleep(delay)
        count += 1
        print "%s: %s" % ( threadName, time.ctime(time.time()) )
try:
    thread.start_new_thread( print_time, ("Thread-1", 2, ) )
    thread.start_new_thread( print_time, ("Thread-2", 4, ) )
except:
    print "Error: unable to start thread"

while 1: pass
```

# Multithreading

- **threading.activeCount():** Returns the number of thread objects that are active.
- **threading.currentThread():** Returns the number of thread objects in the caller's thread control.
- **threading.enumerate():** Returns a list of all thread objects that are currently active.

# Multithreading

- **run():** The run() method is the entry point for a thread.
- **start():** The start() method starts a thread by calling the run method.
- **join([time]):** The join() waits for threads to terminate.
- **isAlive():** The isAlive() method checks whether a thread is still executing.
- **getName():** The getName() method returns the name of a thread.
- **setName():** The setName() method sets the name of a thread.

# Multithreading

- synchronizacja (Lock class)
- aquire and release method
- funkcja join

# Multithreading

```
class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name
        print_time(self.name, self.counter, 5)
        print "Exiting " + self.name
def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))
        counter -= 1
```

# Multithreading

```
 # Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)
# Start new Threads
thread1.start()
thread2.start()
print "Exiting Main Thread"
```

# Lock

- lock = Lock()
- lock.acquire()
- *# will block if lock is already held* ... access shared resource
- lock.release()

# Event

event = threading.Event()

*# a client thread can wait for the flag to be set*

event.wait()

*# a server thread can set or reset it*

event.set()

event.clear()

# Semaphore

```
semaphore = threading.BoundedSemaphore()
semaphore.acquire() # decrements the counter
... access the shared resource
semaphore.release() # increments the counter
```

```
max_connections = 10
semaphore =
threading.BoundedSemaphore(max_connection
s)
```

# Condition

```
# represents the addition of an item to a resource condition = threading.Condition()
# producer thread ... generate item
condition.acquire() .
.. add item to resource
condition.notify() # signal that a new item is available
condition.release()

condition.acquire()
while True: ... get item from resource
        if item:
                break
condition.wait() # sleep until item becomes available
condition.release()
... process item
```

# Multithreading

```
#!/usr/bin/python
import threading
import time
    class myThread (threading.Thread):
        def __init__(self, threadID, name, counter):
            threading.Thread.__init__(self)
            self.threadID = threadID
            self.name = name
            self.counter = counter
        def run(self):
            print "Starting " + self.name
            # Get lock to synchronize threads
            threadLock.acquire()
            print_time(self.name, self.counter, 3)
            # Free lock to release next thread
            threadLock.release()
```

# Multithreading

```
def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        print "%s: %s" % (threadName,
time.ctime(time.time())) counter -= 1
```

# Multithreading

```
threadLock = threading.Lock()
threads = []
# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)
# Start new Threads
thread1.start()
thread2.start()
# Add threads to thread list
threads.append(thread1)
threads.append(thread2)
# Wait for all threads to complete
for t in threads:
    t.join()
print "Exiting Main Thread"
```

# Multiprocessing

https://docs.python.org/2/library/multiprocessing.html - mechanizmy synchronizacji i sekcji krytycznej podobne jak w module multithreading, możliwość definiowania pamięci shared z wbudowanymi na niej operacjami atomowymi

# Dostęp do bazy danych

```
db = MySQLdb.connect("localhost","testuser","test1" ,"TESTDB" )
method cursor = db.cursor()
cursor.execute("SELECT VERSION()")
data = cursor.fetchone()
print "Database version : %s " % data
server db.close()
```

# Dostęp do bazy danych

```
import MySQLdb
db = MySQLdb.connect("localhost","testuser","test1","TESTDB" )
cursor = db.cursor()
sql = "INSERT INTO EMPLOYEE(FIRST_NAME,  LAST_NAME, AGE, SEX, INCOME) VALUES ('%s', '%s', '%d', '%c', '%d' )" %  ('Mac', 'Mohan', 20, 'M', 2000)
try:
    cursor.execute(sql)
    db.commit()
except:
    db.rollback()


db.close()
```

# Dostęp do baz danych

- **fetchone():** It fetches the next row of a query result set. A result set is an object that is returned when a cursor object is used to query a table.
- **fetchall():** It fetches all the rows in a result set. If some rows have already been extracted from the result set, then it retrieves the remaining rows from the result set.
- **rowcount:** This is a read-only attribute and returns the number of rows that were affected by an execute() method.

# Regular expressions

```
#!/usr/bin/python
import re line = "Cats are smarter than dogs"
matchObj = re.match( r'(.*) are (.*?) .*', line,
re.M|re.I)
if matchObj:
    print "matchObj.group() : ", matchObj.group()
    print "matchObj.group(1) : ", matchObj.group(1)
    print "matchObj.group(2) : ", matchObj.group(2)
else:
    print "No match!!"
```

# Regular expressions

- Match
- Search
- Sub