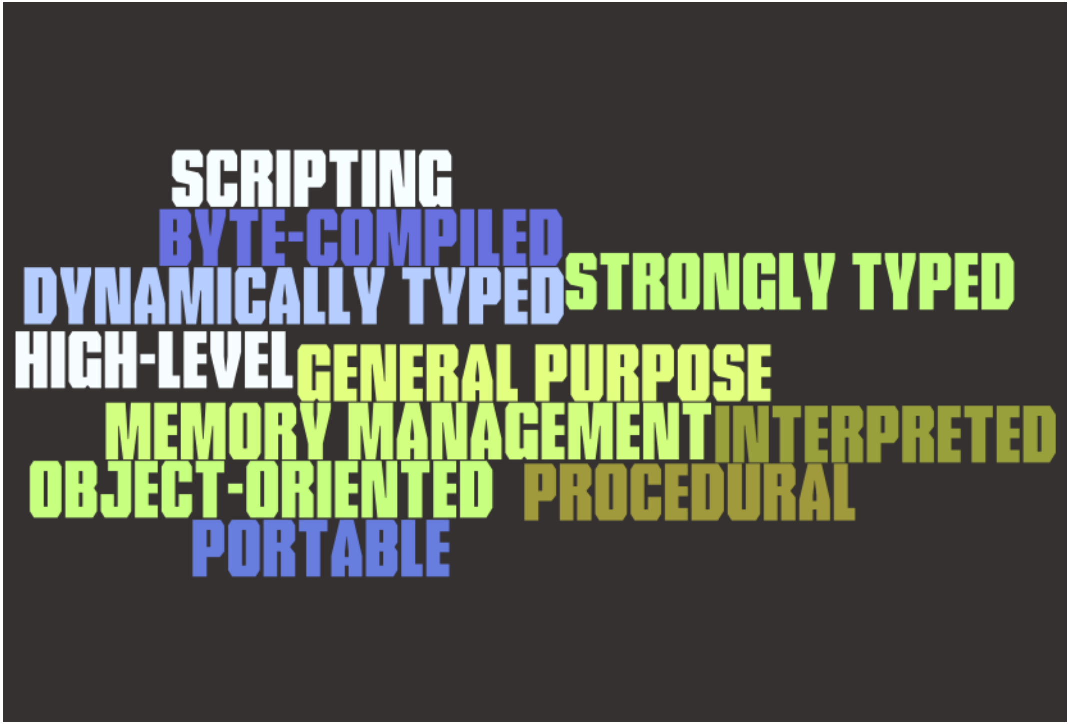
# Dive into Python

**Author:** Bartosz Telenczuk

# Why Python?

- high level
- easy to learn
- easy to read
- Open Source
- large library of users-contributed functions

# Python Words



# What Python is NOT?

- integrated development environment (IDE)
- scientific environment (but wait until Day 3 `numpy`)
- machine code (hence its slower performance)

# Your First Python Program

```
prices = {'milk': 1.00, 'wine': 2.50, 'apples': 0.6}

def sum_bill(purchase):
    """Calculate the total amount to pay"""
    total = 0
    for item, quantity in purchase:
        total += prices[item]*quantity
    return total

#Testing the code
if __name__=='__main__':
    my_purchase = [('milk', 2), ('wine', 1),
                   ('apples', 1.2)]
    bill = sum_bill(my_purchase)

    print 'I owe %.2f Euros' % bill
```

# Python types

Python does not require to provide a type of variables:

```
>>> a = 1 #integer
>>> b = 1.2 #floating point
>>> c = "test" #string
```

but values have types:

```
>>> a + 2
3
>>> a + c
Traceback (most recent call last):
   ...
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

# Lists

Python list is an ordered sequence of values of any type.

```
>>> a = []                          #an empty list
>>> b = ['eggs', 'butter'] #list of strings
>>> c = ['eggs', 4, []]    #mixed-type lists
```

List elements can be accessed by an index (staring with 0!)

```
>>> c[0]                            #get first element
'eggs'
```

Attempt to access nonexisting element rises an error

```
>>> c[3]
Traceback (most recent call last):
    ...
IndexError: list index out of range
```

# Modifying Lists

You can assign a new value to an existing element of a list,

```
>>> c[2] = 3                        #modify 3rd element
```

append a new element:

```
>>> c.append('flower')          #add an element
>>> c
['eggs', 4, 3, 'flower']
```

or delete any element:

```
>>> del c[0]                        #remove an element
>>> c
[4, 3, 'flower']
```

Lists can be easily concatenated using an addition operator:

```
>>> c + ['new', 'list']                     #concatenate lists
[4, 3, 'flower', 'new', 'list']
```

# Slicing

You can take a subsequence of a list using so called **slices**:

```
>>> d = [1, 2, 3, 4, 5, 6]
>>> d[1:3]
[2, 3]
>>> d[:3]
[1, 2, 3]
>>> d[1::2]
[2, 4, 6]
```

Negative indices count elements starting from the end:

```
>>> d[-1]
6
>>> d[2:-2]
[3, 4]
```

# Tuples

Tuples are similar to lists:

```
>>> tup = ('a', 'b', 3)
>>> tup[1]
'b'
```

but they are **immutable**:

```
>>> tup[2]=5
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

Tuples support easy packing/unpacking:

```
>>> x, y, z = tup
>>> print x, y, z
a b 3
```

# Python Idiom 1: Swap variables

A common operation is to swap values of two variables:

```
>>> x, y = (3, 4)
>>> temp = x
>>> x = y
>>> y = temp
>>> print x, y
4 3
```

This can be done more elegant with tuples:

```
>>> x, y = (3, 4)
>>> y, x = x, y
>>> print x, y
4 3
```

# Dictionaries

Dictionary defines one-to-one relationships between keys and values (mapping):

```
>>> tel = {'jack': 4098, 'sape': 4139}
```

You can look up the value using a key:

```
>>> tel['sape']
4139
```

Assigning to a non-existent key creates a new entry:

```
>>> tel['jack'] = 4100
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'jack': 4100, 'guido': 4127}
```

# Python Idiom 2: Switch/Case statement

How to choose from a set of actions depending on a value of some variable? Using a chain of if clauses:

```
if n==1:
    print "Winner!"
elif n==2:
    print "First runner-up"
elif n==3:
    print "Second runner-up"
else:
    print "Work hard next time!"
```

or better using dictionaries:

```
comments = {1: "Winner!",
            2: "First runner-up.",
            3: "Second runner-up."
            }

print comments.get(n, "Work hard next time!")
```

# Strings

Use either single or double quotes to create strings:

```
>>> str1 = '"Hello", she said.'
>>> str2 = "'Hi', he replied."
>>> print str1, str2
"Hello", she said. 'Hi', he replied.
```

Use triple quotes (of either kind) to create multi-line string:

```
>>> str3 = """'Hello', she said.
... 'Hi', he replied."""
>>> print str3
'Hello', she said.
'Hi', he replied.
```

You can also use indexing and slicing with strings:

```
>>> str1[1]
'H'
>>> str1[1:6]
'Hello'
```

# Idiom 3: Building strings from substrings

If you want to join strings into one string you can use addition:

```
>>> a = 'Hello' + 'world'
>>> print a
Helloworld
```

but what if the number of substrings is large?

```
>>> colors = ['red', 'blue', 'yellow', 'green']
>>> print ''.join(colors)
redblueyellowgreen
```

You can also use spaces between your substrings:

```
>>> print ' '.join(colors)
red blue yellow green
```

# String Formatting

In order to get nicely formated output you can use % operator:

```
>>> name, messages = "Bartosz", 2
>>> text = ('Hello %s, you have %d messages.' % (name, messages))
>>> print text
Hello Bartosz, you have 2 messages.
```

You can have more control over the output with format specifiers

```
>>> print 'Real number with 2 digits after point %.2f' % (2/3.)
Real number with 2 digits after point 0.67
>>> print 'Integer padded with zeros %04d' % -1
Integer padded with zeros -001
```

You can also use named variables:

```
>>> entry = "%(name)s's phone number is %(phone)d"
>>> print entry % {'name': 'guido', 'phone': 4343}
guido's phone number is 4343
```

# Condtionals

Python uses `if`, `elif`, and `else` to define conditional clauses.

Nested blocks are introduced by a colon and indentation (4 spaces!).

```
>>> n = -4
>>> if n > 0:
...     print 'greater than 0'
... elif n==0:
...     print 'equal to 0'
... else:
...     print 'less than 0'
less than 0
```

Python 2.5 introduces conditional expressions:

```
x = true_value if condition else false_value

>>> abs_n = -1*n if n<0 else n
>>> abs_n
4
```

# Python Idiom 4: Testing for Truth Values

Take advantage of intrinsic truth values when possible:

```
>>> items = ['ala', 'ma', 'kota']
>>> if items:
...     print 'ala has a cat'
... else:
...     print 'list is empty'
ala has a cat
```

| False | True |
|---|---|
| `False` (== 0) | `True` (== 1) |
| `""` (empty string) | any string but `""` (`" "`, `"anything"`) |
| `0`, `0.0` | any number but `0` (1, 0.1, -1, 3.14) |
| `[]`, `()`, `{}`, `set()` | any non-empty container (`[0]`, `(None,)`, `['']`) |
| `None` | almost any object that's not explicitly False |

# Looping Techniques: While and for loops

Do something repeatedly as long as some condition is true:

```
>>> num_moons = 3
>>> while num_moons > 0:
...     print num_moons,
...     num_moons -= 1
3 2 1
```

If you know number of iterations in advance use for loop:

```
>>> for i in xrange(3):
...     print i,
0 1 2
```

Note the colon and indentation for the nested blocks!

# Python Idiom 5: Iterators

Many data structures provide **iterators** which ease looping through their elements:

```
>>> clock = ['tic', 'tac', 'toe']
>>> for x in clock:
...     print x,
tic tac toe

>>> prices = {'apples': 1, 'grapes': 3}
>>> for key, value in prices.iteritems():
...     print '%s cost %d euro per kilo' % (key, value)
apples cost 1 euro per kilo
grapes cost 3 euro per kilo
```

If you also need indexes of the items use `enumerate`:

```
>>> for i, x in enumerate(clock):
...     print i, x,
0 tic 1 tac 2 toe
```

# List Comprehensions

List comprehension provides a compact way of mapping a list into another list by applying a function to each of its elements:

```
>>> [x**2 for x in xrange(5)]
[0, 1, 4, 9, 16]

>>> freshfruit = ['  banana',
...               '  loganberry ',
...              'passion fruit  ']
>>> [x.strip() for x in freshfruit]
['banana', 'loganberry', 'passion fruit']
```

It is also possible to nest list comprehensions:

```
>>> [[i*j for i in xrange(1,4)] for j in xrange(1,4)]
[[1, 2, 3], [2, 4, 6], [3, 6, 9]]
```

# Declaring Functions

Function definition = identifier + arguments + docstring + content

```
>>> def double(n):
...     """Double and return the input argument."""
...     return n*2
```

Now call the function we have just defined:

```
>>> a = double(5)
>>> b = double(['one', 'two'])
>>> print a, b
10 ['one', 'two', 'one', 'two']
```

Functions are objects:

```
>>> print double.__doc__
Double and return the input argument.
```

# Passing Arguments

It is possible to define default values for arguments:

```
>>> def bracket(value, lower=0, upper=None):
...     """Limit a value to a specific range (lower, upper)"""
...     if upper:
...         value = min(value, upper)
...     return max(value, lower)
>>> bracket(2)
2
>>> bracket(2, 3)
3
```

Functions can be also called using keyword arguments:

```
>>> bracket(2, upper=1)
1
```

# Python Idiom 6: Functions as arguments

Functions are first class objects and can be passed as functions arguments like any other object:

```
>>> def apply_to_list(func, target_list):
...     return [func(x) for x in target_list]

>>> a = range(-3, 5, 2)
>>> b = apply_to_list(bracket, a)
>>> print 'before:', a, 'after:', b
before: [-3, -1, 1, 3] after: [0, 0, 1, 3]
```

In Python there are several builtin functions operating on functions and lists. For example, `map` applies any function to each element of a list:

```
>>> print map(bracket, a)
[0, 0, 1, 3]
```

# Introspection

You can learn much about Python objects directly in Python interpreter.

- `help`: prints help information including docstring
- `dir`: lists all methods and attributes of a class
- `type`: returns an object's type
- `str`: gives a string representation of an object

```
>>> type([])
<type 'list'>
>>> dir([])
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__delslice__', '__doc__', '__eq__', '__ge__', '__getattribute__', '__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__setslice__', '__str__', 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

# Coding Style

- Use 4-space indentation, and no tabs.
- Wrap lines so that they don't exceed 79 characters.
- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own.
- Use docstrings.
- Use spaces around operators and after commas, but not directly inside bracketing constructs: `a = f(1, 2) + g(3, 4)`.
- Name your classes and functions consistently; the convention is to use `CamelCase` for classes and `lower_case_with_underscores` for functions and methods.

Check PEP8 for complete list of coding conventions.

# Scope

Python looks for the variables in the following order:

- local scope (function)
- module scope
- global scope

```
>>> a, b = "global A", "global B"

>>> def foo():
...     b = "local B"
...     print "Function Scope: a=%s, b=%s" % (a, b)

>>> print "Global Scope: a=%s, b=%s" % (a, b)
Global Scope: a=global A, b=global B
>>> foo()
Function Scope: a=global A, b=local B
```

# Function Libraries = Modules

Python allows to organize functions into **modules**. Every Python file is automatically a module:

```
# mymodule.py
def bracket(value, lower=0, upper=None):
    """Limit a value to a specific range (lower, upper)"""
    if upper:
        value = min(value, upper)
    return max(value, lower)

 def apply_to_list(func, target_list):
     """Apply function func to each element of the target list)"""
     return [func(x) for x in target_list]
```

You can import the module into your current scope:

```
>>> import mymodule
>>> x = range(-2, 4)
>>> mymodule.apply_to_list(mymodule.bracket, x)
[0, 0, 0, 1, 2, 3]
```

# Imports

You can define an alias for your module when importing:

```
>>> import mymodule as m
>>> m.bracket.__doc__
'Limit a value to a specific range (lower, upper)'
```

or you can import only specific functions:

```
>>> from mymodule import bracket
>>> bracket(-5)
0
```

It is possible to import all definitions from a module:

```
>>> from mymodule import *
>>> apply_to_list(bracket, [-1, 2, -3])
[0, 2, 0]
```

but it is **NOT** recommended!

# Python Idiom 7: Testing a module

Often you want to include some sample code or tests with your module which should be executed only when it is run as a script but not when it is imported.

```
#mymodule.py
...
if __name__=='__main__':
    x = [-1, 2, 3]
    x_bracket = apply_to_list(bracket, x)
    print "Original List: %s, Bracketed List: %s" % (x, x_bracket)
```

If you run it from a shell:

```
> python mymodule.py
Original List: [-1, 2, 3], Bracketed List:  [0, 2, 3]
```

but when you import it:

```
>>> import mymodule
```

# Simulating Ecosystem

Suppose you want to simulate a small ecosystem of different organisms:

- Plants (don't move)
- Fish (swim)
- Dogs (walk)

You could implement it in a procedural way:

```
for time in simulation_period:
    for organism in world:
        if type(organism) is plant:
            pass
        elif type(organism) is fish:
            swim(organism, time)
        elif type(organism) is dog:
            walk(organism, time)
```

but it is not easy to extend it with new organisms.

# Objects to the Rescue

In order to solve the problem we define custom types called objects. Each object defines a way it moves:

```
for time in simulation_period:
    for organism in world:
        organism.update(time)
```

Such approach is called **object-oriented programming**:

- we don't have to remember how each organism moves
- it is easy to add new organisms - no need to change the existing code
- small change, but it allows programmers to think at a higher level

# Python Classes

**Class** is a definition that specifies the properties of a set of objects.

Defining a class in Python:

```
>>> class Organism(object):
...     pass
```

Creating a class **instance** (object):

```
>>> first = Organism()
>>> second = Organism()
```

# Methods

Objects have behaviors and states. Behaviors are defined in methods:

```
>>> class Organism(object):
...     def speak(self, name):
...         print "Hi, %s. I'm an organism." % name
```

The object itself is always passed to the method as its first argument (called `self`).

Object methods are called using **dot notation**:

```
>>> some_organism = Organism()
>>> some_organism.speak('Edgy')
Hi, Edgy. I'm an organism.
```

# Attributes

Current state of the object is defined by **attributes.** You can access object attributes using dot notation:

```
>>> some_organism.species = "unknown"
>>> print some_organism.species
unknown
```

Attributes can be initialized in a special method called `__init__` (constructor):

```
>>> class Organism(object):
...     def __init__(self, species):
...         self.species = species
```

You can pass arguments to the constructor when creating new instance:

```
>>> some_organism = Organism("amoeba")
>>> print some_organism.species
amoeba
```

# Encapsulation

Methods can access attributes of the object they belong to by referring to `self`:

```
>>> class MotileOrganism(object):
...     def __init__(self):
...         self.position = 0
...     def move(self):
...         speed = 1
...         self.position += speed
...     def where(self):
...         print "Current position is", self.position
```

```
>>> motile_organism = MotileOrganism()
>>> motile_organism.move()
>>> motile_organism.where()
Current position is 1
```

Any function or method can see and modify any object's internals using its instance variable.

```
>>> motile_organism.position = 10
```

# Inheritance

**Problem**:

Only some organisms can move and other don't, but all of them have names and can speak (sic!).

**Solutions**:

- define separate classes for each type of organisms and copy common methods (WRONG!)
- extend classes with new abilities using **inheritance** (BETTER!)

# Inheritance Example

```
>>> class Organism(object):
...     def __init__(self, species="unknown"):
...         self.species = species
...     def speak(self):
...         print "Hi. I'm a %s." % (self.species)
>>> class MotileOrganism(Organism):
...    def __init__(self, species="unknown"):
...         self.species = species
...         self.position = 0
...    def move(self):
...        self.position += 1
...    def where(self):
...        print "Current position is", self.position
>>> algae = Organism("algae")
>>> amoeba = MotileOrganism("amoeba")
>>> amoeba.speak()
Hi. I'm a amoeba.
>>> amoeba.move()
>>> amoeba.where()
Current position is 1
```

# Reading and Writing Files

```
>>> f = open('workfile', 'r') #Open a file in a readonly mode
>>> f.read() #Read entire file
'This is the first line.\nThis is the second line.\n'
>>> f.seek(0)
>>> f.readline() #Read one line
'This is the first line.\n'

#Use iterator to loop over the lines
>>> f.seek(0)
>>> for line in f:
...     print line,
This is the first line.
This is the second line.

#Write a string to a file
>>> f = open('savefile', 'w')
>>> f.write('This is a test\n')
>>> f.close()
```

# Regular Expressions

**Regular expressions** provide simple means to identify strings of text of interest.

First define a pattern to be matched:

```
>>> import re
>>> p = re.compile('name=([a-z]+)')
```

Now try if a string "tempo" matches it:

```
>>> m = p.match('name=bartosz')
>>> m.group()
'name=bartosz'
```

or search for the matching substring and :

```
>>> m = p.search('id=1;name=bartosz;status=student')
>>> m.group()
'name=bartosz'
```

You can also parse the string for some specific information:

```
>>> m.group(1)
'bartosz'
```

Learn more about **regexp** in the short HOWTO

# Exceptions

Python exceptions are cought the `try` block and handled in `except` block:

```
>>> filename = 'nonexisting.file'
>>> try:
...     f = open(filename, 'r')
... except IOError:
...     print 'cannot open:', filename
cannot open: nonexisting.file
```

To trigger exception processing use `raise`:

```
>>> for i in range(4):
...     try:
...         if (i % 2) == 1:
...             raise ValueError('index is odd')
...     except ValueError, e:
...         print 'caught exception for %d' % i, e
caught exception for 1 index is odd
caught exception for 3 index is odd
```

Built-in exceptions lists the built-in exceptions and their meaning.