



**Jussi Enkovaara**  
**Harri Hämäläinen**



## Python in High-performance Computing

**January 27-29, 2015**

**PRACE Advanced Training Centre**

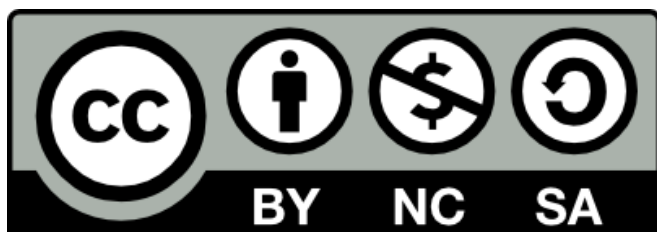
**CSC – IT Center for Science Ltd, Finland**

```
import sys, os
try:
    from Bio.PDB import PDBParser
    __biopython_installed__ = True
except ImportError:
    __biopython_installed__ = False

__default_bfactor__ = 0.0      # default B-factor
__default_occupancy__ = 1.0    # default occupancy level
__default_segid__ = ''        # empty segment ID

class EOF(Exception):
    def __init__(self): pass

class FileCrawler:
    """
    Crawl through a file reading back and forth without loading
    anything to memory.
    """
    def __init__(self, filename):
        try:
            self.__fp__ = open(filename)
        except IOError:
            raise ValueError, "Couldn't open file '%s' for reading." % filename
        self.tell = self.__fp__.tell
        self.seek = self.__fp__.seek
    def prevline(self):
        try:
            self.prev()
```



All material (C) 2015 by the authors.

This work is licensed under a **Creative Commons Attribution-NonCommercial-ShareAlike 3.0** Unported License, <http://creativecommons.org/licenses/by-nc-sa/3.0/>

# Agenda

## Tuesday

9:00-9:45	<b>Introduction to Python</b>
9:45-10:30	Exercises
10:30-10:45	Coffee Break
10:45-11:15	<b>Control structures</b>
11:15-12:15	Exercises
12:15-13:00	Lunch break
13:00-13:30	<b>Functions and modules</b>
13:30-14:30	Exercises
14:30-14:45	Coffee Break
14:45-15:15	<b>File I/O and text processing</b>
15:15-16:15	Exercises

## Wednesday

9.00-9.45	<b>Object oriented programming with Python</b>
9.45-10.30	Exercises
10.30-10.45	Coffee break
10:45-11:15	<b>NumPy – fast array interface to Python</b>
11:15-12:15	Exercises
12.15-13.00	Lunch break
13.00-13:30	<b>NumPy (continued)</b>
13:30-14:30	Exercises
14.30-14.45	Coffee break
14.45-15.15	<b>NumPy (continued)</b>
15:15-16:15	Exercises

## Thursday

9:00-9:45	<b>Visualization with Python</b>
9:45-10:30	Exercises
10:30-10:45	Coffee Break
10:45-11:30	<b>Scipy-package for scientific computing</b>
11:30-12:15	Exercises
12:15-13:00	Lunch break
13:00-13:30	<b>C extensions – integrating efficient C routines in Python</b>
13:30-14:30	Exercises
14:30-14:45	Coffee break
14:45-15:45	<b>MPI and Python – mpi4py</b>
15:45-16:15	Exercises





# INTRODUCTION TO PYTHON

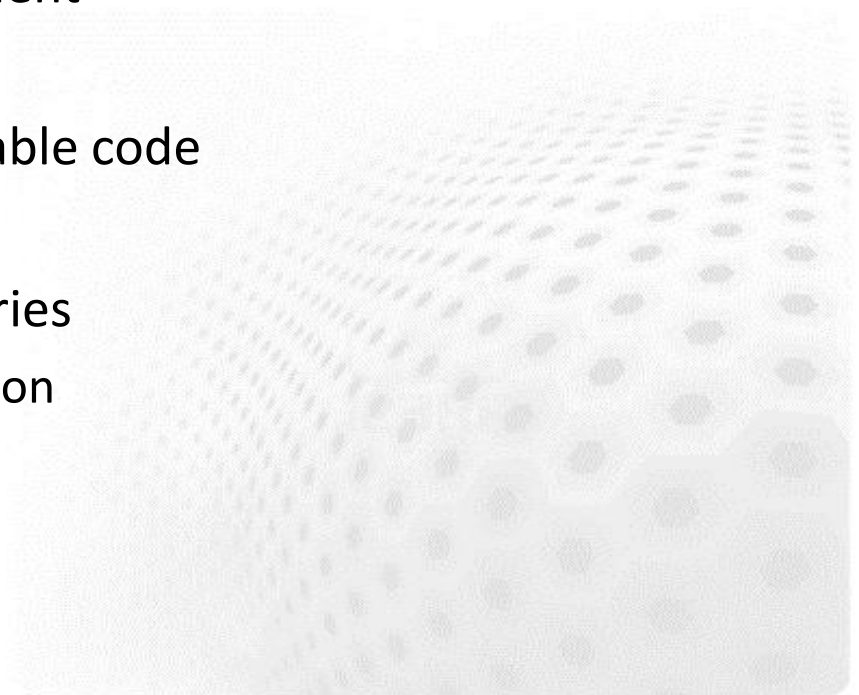


## What is Python?

- Modern, interpreted, object-oriented, full featured high level programming language
- Portable (Unix/Linux, Mac OS X, Windows)
- Open source, intellectual property rights held by the Python Software Foundation
- Python versions: 2.x and 3.x
  - 3.x is not backwards compatible with 2.x
  - This course uses 2.x version

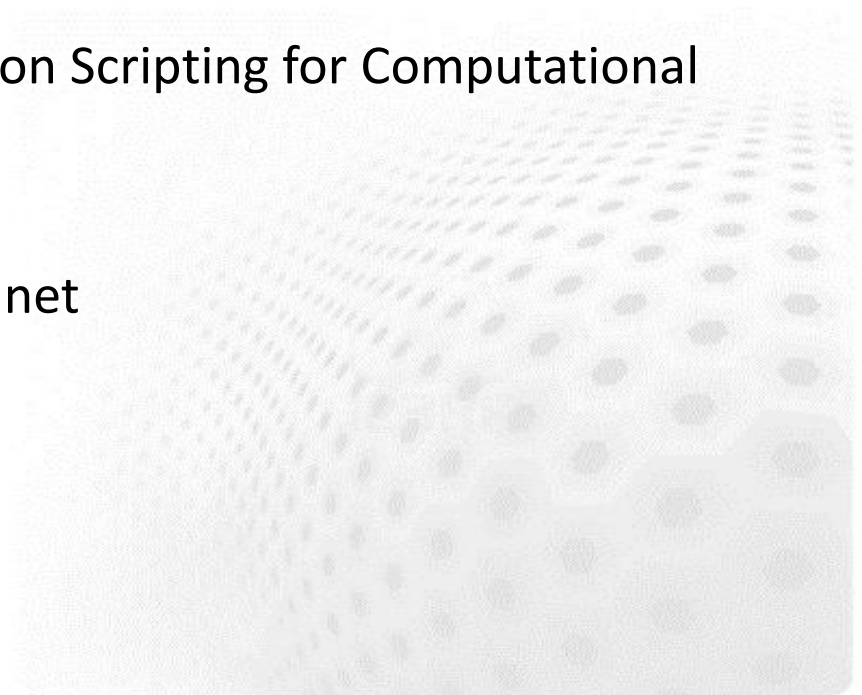
## Why Python?

- Fast program development
- Simple syntax
- Easy to write well readable code
- Large standard library
- Lots of third party libraries
  - Numpy, Scipy, Biopython
  - Matplotlib
  - ...



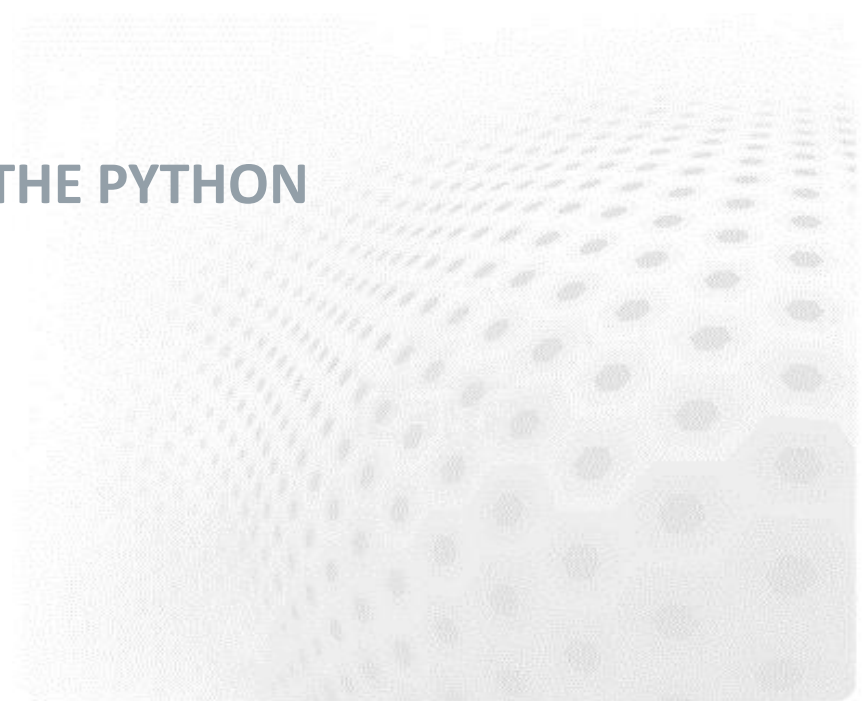
## Information about Python

- [www.python.org](http://www.python.org)
- H. P. Langtangen, “Python Scripting for Computational Science”, Springer
- [www.scipy.org](http://www.scipy.org)
- [matplotlib.sourceforge.net](http://matplotlib.sourceforge.net)
- [mpi4py.scipy.org](http://mpi4py.scipy.org)



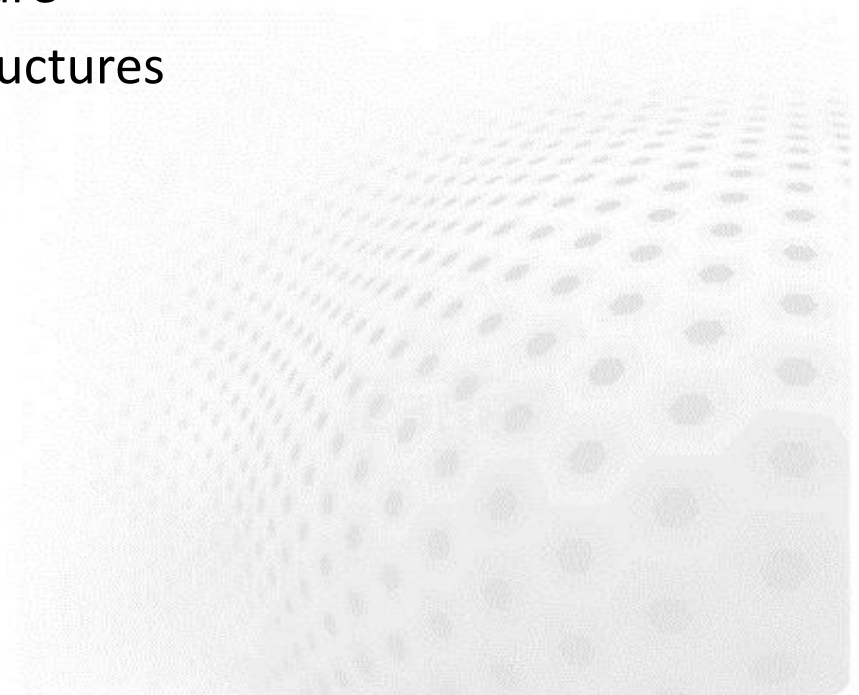


## FIRST GLIMPSE INTO THE PYTHON



## Python basics

- Syntax and code structure
- Data types and data structures
- Control structures
- Functions and modules
- Text processing and IO



## Python program

- Typically, a .py ending is used for Python scripts, e.g. *hello.py*:

```
hello.py  
print "Hello world!"
```

- Scripts can be executed by the *python* executable:

```
$ python hello.py  
Hello world!
```

## Interactive python interpreter

- The interactive interpreter can be started by executing python without arguments:

```
$ python
Python 2.4.3 (#1, Jul 16 2009, 06:20:46)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-44)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello"
Hello
>>>
```

- Useful for testing and learning

## Python syntax

- Variable and function names start with a letter and can contain also numbers and underscores, e.g “my\_var”, “my\_var2”
- Python is case sensitive
- Code blocks are defined by indentation
- Comments start by # sign

```
example.py
# example
if x > 0:
    x = x + 1 # increase x
    print("increasing x")
else:
    x = x - 1
    print "decreasing x"
print("x is processed")
```



## Data types

- Python is dynamically typed language
  - no type declarations for variables
- Variable does have a type
  - incompatible types cannot be combined

```
example.py
print "Starting example"
x = 1.0
for i in range(10):
    x += 1
y = 4 * x
s = "Result"
z = s + y # Error
```

## Numeric types

- Integers
- Floats
- Complex numbers
- Basic operations
  - $+$  and  $-$
  - $*$ ,  $/$  and  $**$
  - implicit type conversions
  - be careful with integer division !

```
>>> x = 2
>>> x = 3.0
>>> x = 4.0 + 5.0j
>>>
>>> 2.0 + 5 - 3
4.0
>>> 4.0**2 / 2.0 * (1.0 - 3j)
(8-24j)
>>> 1/2
0
>>> 1./2
0.5
```

## Strings

- Strings are enclosed by " or '
- Multiline strings can be defined with three double quotes

strings.py

```
s1 = "very simple string"
s2 = 'same simple string'
s3 = "this isn't so simple string"
s4 = 'is this "complex" string?'
s5 = """This is a long string
expanding to multiple lines,
so it is enclosed by three '''s."""
```

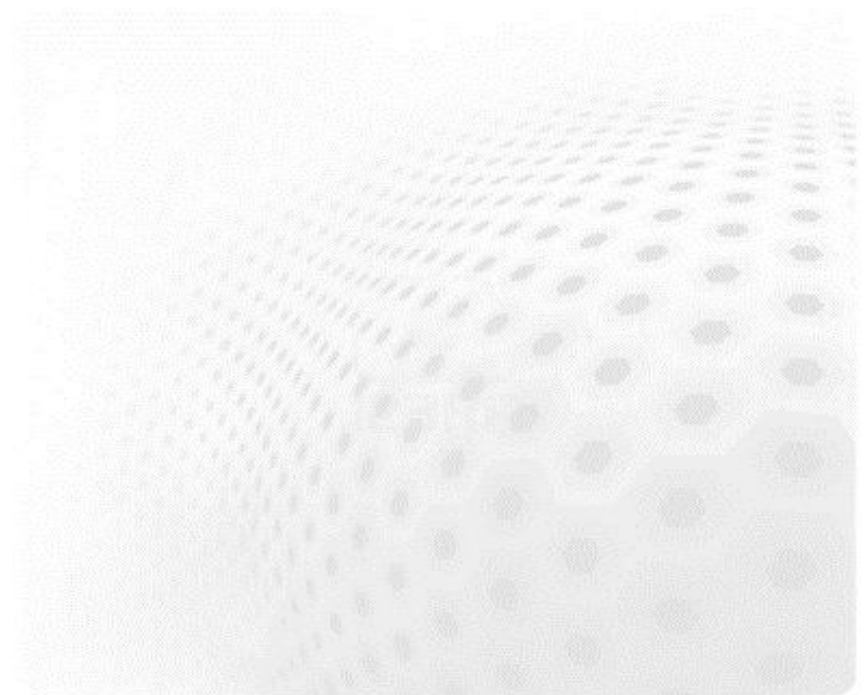
## Strings

🔴 + and \* operators with strings:

```
>>> "Strings can be " + "combined"
'Strings can be combined'
>>>
>>> "Repeat! " * 3
'Repeat! Repeat! Repeat!'
```

## Data structures

- Lists and tuples
- Dictionaries





## List

- Python lists are dynamic arrays
- List items are indexed (index starts from 0)
- List item can be any Python object, items can be of different type
- New items can be added to any place in the list
- Items can be removed from any place of the list

## Lists

- Defining lists

```
>>> my_list1 = [3, "egg", 6.2, 7]
>>> my_list2 = [12, [4, 5], 13, 1]
```

- Accessing list elements

```
>>> my_list1[0]
3
>>> my_list2[1]
[4, 5]
>>> my_list1[-1]
7
```

- Modifying list items

```
>>> my_list1[-2] = 4
>>> my_list1
[3, 'egg', 4, 7]
```

## Lists

- Adding items to list
- Accessing list elements
- **+** and **\*** operators with lists

```
>>> my_list1 = [9, 8, 7, 6]
>>> my_list1.append(11)
>>> my_list1
[9, 8, 7, 6, 11]
>>> my_list1.insert(1,16)
>>> my_list1
[9, 16, 8, 7, 6, 11]
>>> my_list2 = [5, 4]
>>> my_list1.extend(my_list2)
>>> my_list1
[9, 16, 8, 7, 6, 11, 5, 4]
```

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> [1, 2, 3] * 2
[1, 2, 3, 1, 2, 3]
```

## Lists

- It is possible to access slices of lists

```
>>> my_list1 = [0, 1, 2, 3, 4, 5]
>>> my_list1[0:2]
[0, 1]
>>> my_list1[:2]
[0, 1]
>>> my_list1[3:]
[3, 4, 5]
>>> my_list1[0:6:2]
[0, 2, 4]
>>> my_list1[::-1]
[5, 4, 3, 2, 1, 0]
```

- Removing list items

```
>>> second = my_list1.pop(2)
>>> my_list1
[0, 1, 3, 4, 5]
>>> second
2
```

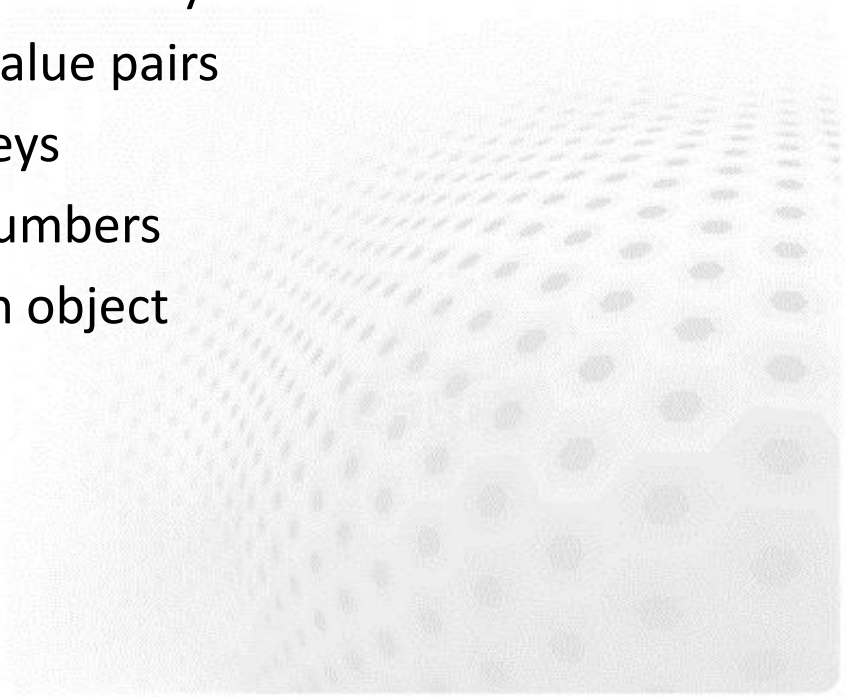
# Tuples

- Tuples are immutable lists
- Tuples are indexed and sliced like lists, but cannot be modified

```
>>> t1 = (1, 2, 3)
>>> t1[1] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
TypeError: 'tuple' object does not
support item assignment
```



## Dictionaries

- Dictionaries are associative arrays
  - Unordered list of key - value pairs
  - Values are indexed by keys
  - Keys can be strings or numbers
  - Value can be any Python object
- 

## Dictionaries

- Creating dictionaries

```
>>> grades = {'Alice' : 5, 'John' : 4, 'Carl' : 2}
>>> grades
{'John': 4, 'Alice': 5, 'Carl': 2}
```

- Accessing values

```
>>> grades['John']
4
```

- Adding items

```
>>> grades['Linda'] = 3
>>> grades
{'John': 4, 'Alice': 5, 'Carl': 2, 'Linda': 3}
>>> elements = {}
>>> elements['Fe'] = 26
>>> elements
{'Fe': 26}
```

## Variables

- Python variables are always references
- `my_list1` and `my_list2` are references to the same list
  - Modifying `my_list2` changes also `my_list1`!
- Copy can be made by slicing the whole list

```
>>> my_list1 = [1,2,3,4]
>>> my_list2 = my_list1
```

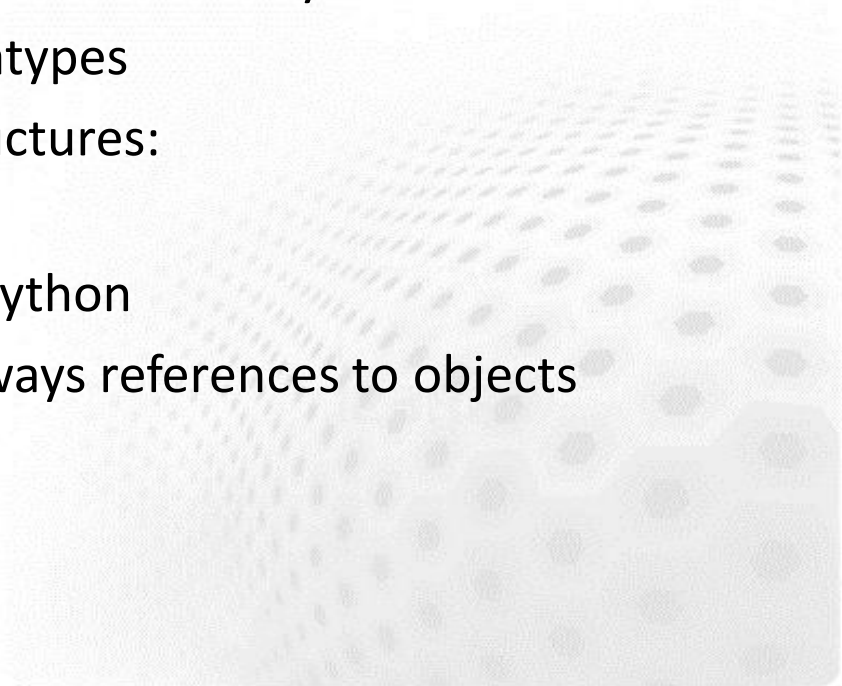
```
>>> my_list2[0] = 0
>>> my_list1
[0, 2, 3, 4]
```

```
>>> my_list3 = my_list1[:]
>>> my_list3[-1] = 66
>>> my_list1
[0, 2, 3, 4]
>>> my_list3
[0, 2, 3, 66]
```

## What is object?

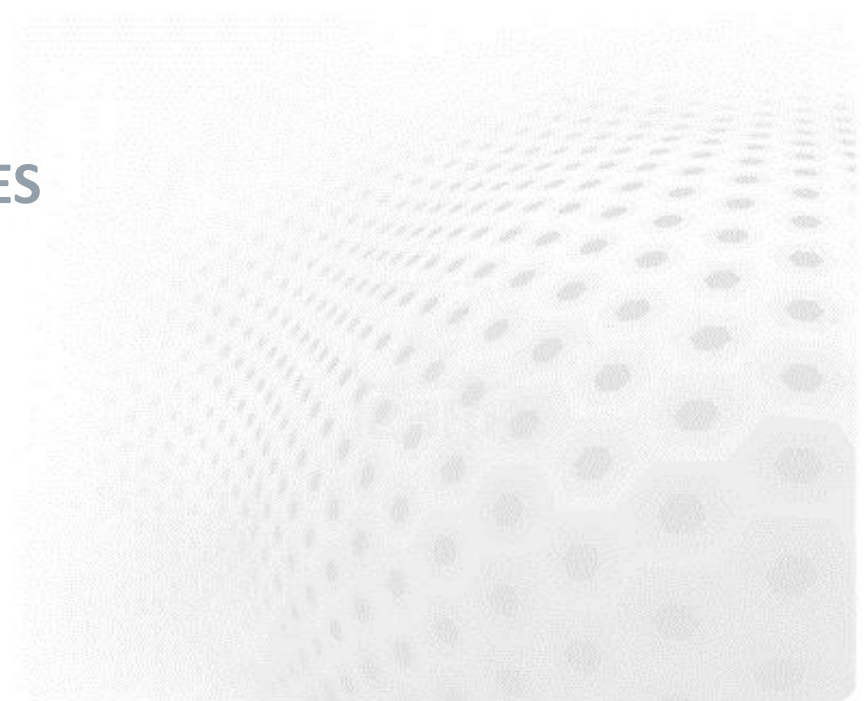
- Object is a software bundle of data (=variables) and related methods
- Data can be accessed directly or only via the methods (=functions) of the object
- In Python, **everything** is object
- Methods of object are called with the syntax:  
**obj.method**
- Methods can modify the data of object or return new objects

## Summary

- Python syntax: code blocks defined by indentation
  - Numeric and string datatypes
  - Powerful basic data structures:
    - Lists and dictionaries
  - Everything is object in Python
  - Python variables are always references to objects
- 

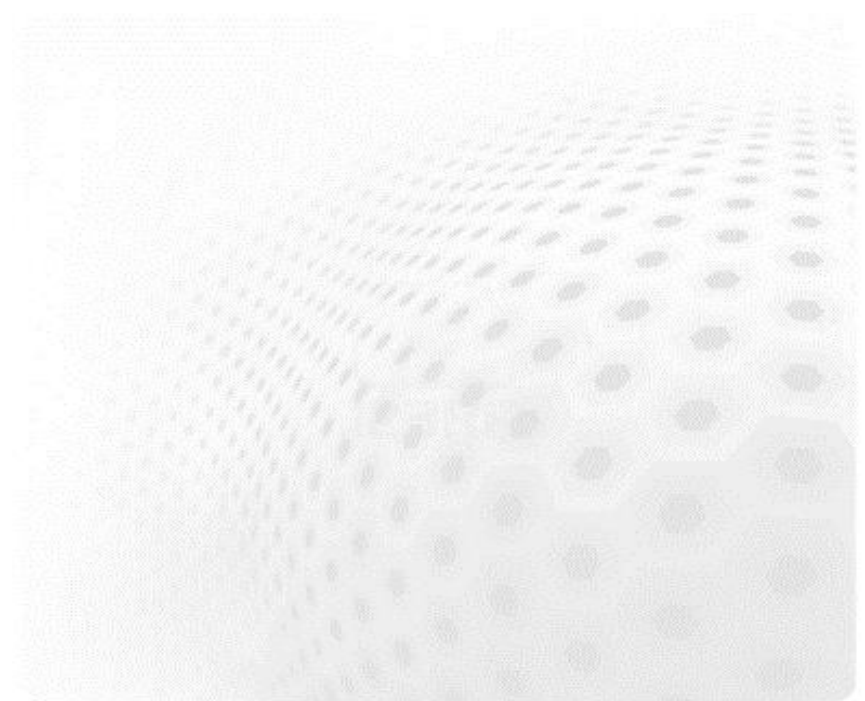


# CONTROL STRUCTURES



## Control structures

- **if – else** statements
- **while loops**
- **for** loops
- Exceptions



## if statement

- **if** statement allows one to execute code block depending on condition
- code blocks are defined by indentation, standard practice is to use four spaces for indentation

example.py

```
if x > 0:  
    x += 1  
    y = 4 * x  
numbers[2] = x
```

- boolean operators:

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

## if statement

- there can be multiple branches of conditions

```
example.py
if x == 0:
    print "x is zero"
elif x < 0:
    print "x is negative"
elif x > 100000:
    print "x is large"
else:
    print "x is something completely different"
```

- Python does not have switch statement

## while loop

- **while** loop executes a code block as long as an expression is True

example.py

```
x = 0
cubes = {}
cube = 0
while cube < 100:
    cubes[x] = cube
    x += 1
    cube = x**3
```

## for loop

- **for** statement iterates over the items of any sequence (e.g. list)

example.py

```
cars = ['Audi', 'BMW', 'Jaguar', 'Lada']
```

```
for car in cars:  
    print "Car is ", car
```

- In each pass, the loop variable **car** gets assigned next value from the sequence
  - Value of loop variable can be any Python object

## for loop

- Many sequence-like Python objects support iteration
  - Dictionary: "next" values are dictionary keys

example.py

```
prices = {'Audi' : 50, 'BMW' : 70, 'Lada' : 5}

for car in prices:
    print "Car is ", car
    print "Price is ", prices[car]
```

- (later on: file as sequence of lines, "next" value of file object is the next line in the file)



## for loop

- Items in the sequence can be lists themselves

example.py

```
coordinates = [[1.0, 0.0], [0.5, 0.5], [0.0, 1.0]]
for coord in coordinates:
    print "X=", coord[0], "Y=", coord[1]
```

- Values can be assigned to multiple loop variables

example.py

```
for x, y in coordinates:
    print "X=", x, "Y=", y
```

- Dictionary method **items()** returns list of key-value pairs

example.py

```
prices = {'Audi': 50, 'BMW' : 70, 'Lada' : 5}
for car, price in prices.items():
    print "Price of", car, "is", price
```

## break & continue

### ● break out of the loop

example.py

```
x = 0
while True:
    x += 1
    cube = x**3
    if cube > 100:
        break
```

example.py

```
sum = 0
for p in prices:
    sum += p
    if sum > 100:
        print "too much"
        break
```

### ● continue with the next iteration of loop

example.py

```
x = -5
cube = 0
while cube < 100:
    x += 1
    if x < 0:
        continue
    cube = x**3
```

example.py

```
sum = 0
for p in prices:
    if p > 100:
        continue
    sum += p
```

## exceptions

- Exceptions allow the program to handle errors and other "unusual" situations in a flexible and clean way
- Basic concepts:
  - Raising an exception. Exception can be raised by user code or by system
  - Handling an exception. Defines what to do when an exception is raised, typically in user code.
- There can be different exceptions and they can be handled by different code

## exceptions in Python

- Exception is caught and handled by **try - except** statements

```
example.py
my_list = [3, 4, 5]
try:
    fourth = my_list[4]
except IndexError:
    print "There is no fourth element"
```

- User code can also **raise** an exception

```
example.py
if solver not in ['exact', 'jacobi', 'cg']:
    raise RuntimeError('Unsupported solver')
```

## List comprehension

- useful Python idiom for creating lists from existing ones without explicit **for** loops
- creates a new list by performing operations for the elements of list:

**newlist = [op(x) for x in oldlist]**

```
>>> numbers = range(6)
>>> squares = [x**2 for x in numbers]
>>> squares
[0, 1, 4, 9, 16, 25]
```

- a conditional statement can be included

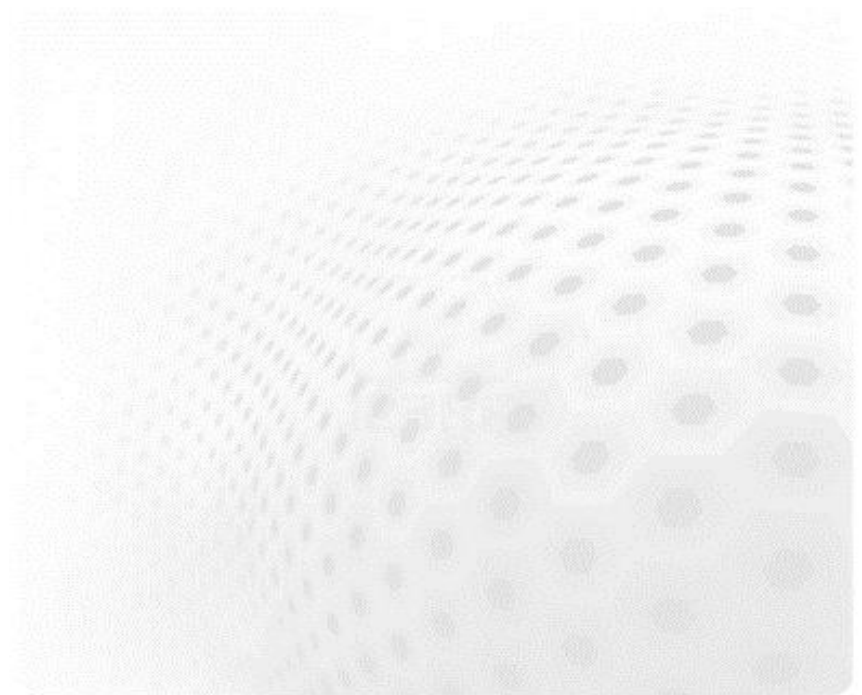
```
>>> odd_squares = [x**2 for x in numbers if x % 2 == 1]
>>> odd_squares
[1, 9, 25]
```

# FUNCTIONS AND MODULES



## Functions and modules

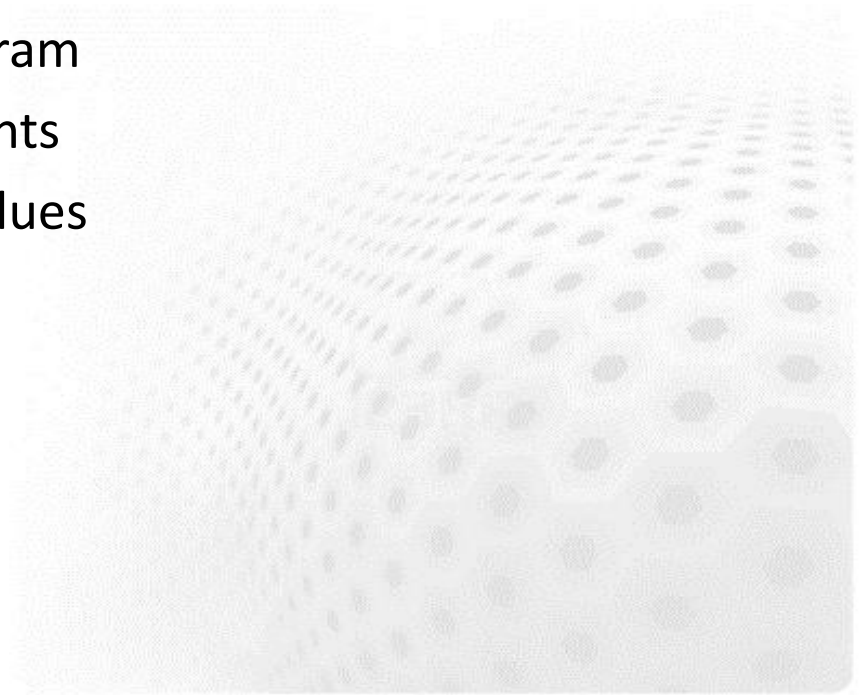
- defining functions
- calling functions
- importing modules





## Functions

- function is block of code that can be referenced from other parts of the program
- functions have arguments
- functions can return values



## Function definition

```
function.py
def add(x, y):
    result = x + y
    return result

u = 3.0
v = 5.0
sum = add(u, v)
```

- name of function is **add**
- **x** and **y** are arguments
- there can be any number of arguments and arguments can be any Python objects
- return value can be any Python object

## Keyword arguments

- functions can also be called using keyword arguments

```
function.py
def sub(x, y):
    result = x - y
    return result

res1 = sub(3.0, 2.0)
res2 = sub(y=3.0, x=2.0)
```

- keyword arguments can improve readability of code

## Default arguments

- it is possible to have default values for arguments
- function can then be called with varying number of arguments

function.py

```
def add(x, y=1.0):  
    result = x + y  
    return result
```

```
sum1 = add(0.0, 2.0)  
sum2 = add(3.0)
```

## Modifying function arguments

- as Python variables are always references, function can modify the objects that arguments refer to

```
>>> def switch(mylist):  
...     tmp = mylist[-1]  
...     mylist[-1] = mylist[0]  
...     mylist[0] = tmp  
...  
>>> l1 = [1,2,3,4,5]  
>>> switch(l1)  
>>> l1  
[5, 2, 3, 4, 1]
```

- side effects can be wanted or unwanted

## Modules

- modules are extensions that can be imported to Python to provide additional functionality, e.g.
  - new data structures and data types
  - functions
- Python standard library includes several modules
- several third party modules
- user defined modules

# Importing modules

## 🔴 import statement

```
example.py
import math
x = math.exp(3.5)

import math as m
x = m.exp(3.5)

from math import exp, pi
x = exp(3.5) + pi

from math import *
x = exp(3.5) + sqrt(pi)

exp = 6.6
from math import *
x = exp + 3.2 # Won't work,
              # exp is now a function
```



## Creating modules

- it is possible to make imports from own modules
- define a function in file **mymodule.py**

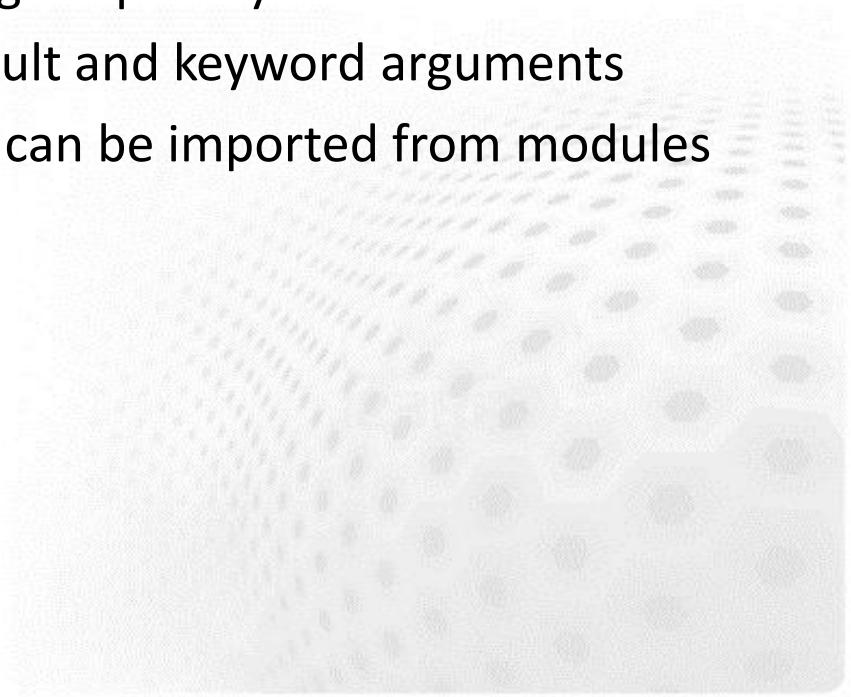
```
mymodule.py  
def incx(x):  
    return x+1
```

- the function can now be imported in other .py files:

```
test.py  
import mymodule  
  
y = mymodule.incx(1)
```

```
test.py  
from mymodule import incx  
  
y = incx(1)
```

## Summary

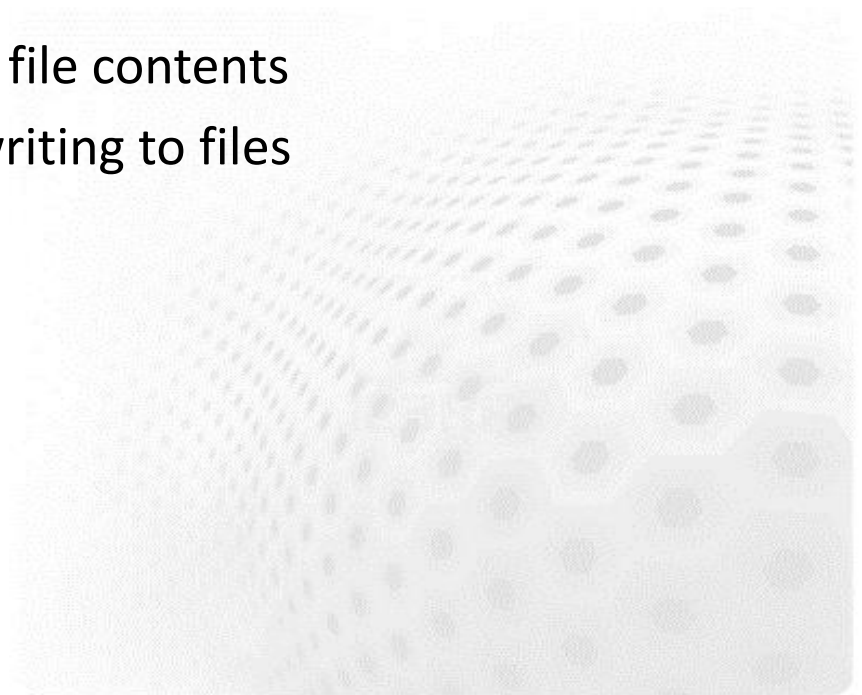
- functions help in reusing frequently used code blocks
  - functions can have default and keyword arguments
  - additional functionality can be imported from modules
- 

## FILE I/O AND TEXT PROCESSING



## File I/O and text processing

- working with files
- reading and processing file contents
- string formatting and writing to files



## Opening and closing files

- opening a file:  
`myfile = open(filename, mode)`
  - returns a handle to the file

```
>>> fp = open('example.txt', 'r')  
>>>
```

## Opening and closing files

- file can be opened for
  - reading: `mode='r'`  
(file has to exist)
  - writing: `mode='w'`  
(existing file is truncated)
  - appending: `mode='a'`
- closing a file
  - `myfile.close()`

example.py

```
# open file for reading
infile = open('input.dat', 'r')

# open file for writing
outfile = open('output.dat', 'w')

# open file for appending
appfile = open('output.dat', 'a')

# close files
infile.close()
```

## Reading from files

- a single line can be read from a file with the `readline()` - function

```
>>> infile = open('inp', 'r')
>>> line = infile.readline()
```

- it is often convenient to iterate over all the lines in a file

```
>>> infile = open('inp', 'r')
>>> for line in infile:
...     # process lines
```



## Processing lines

- generally, a line read from a file is just a string
- a string can be split into a list of strings:

```
>>> infile = open('inp', 'r')
>>> for line in infile:
...     line = line.split()
```

- fields in a line can be assigned to variables and added to e.g. lists or dictionaries

```
>>> for line in infile:
...     line = line.split()
...     x, y = float(line[1]), float(line[3])
...     coords.append((x,y))
```

## Processing lines

- sometimes one wants to process only files containing specific tags or substrings

```
>>> for line in infile:
...     if "Force" in line:
...         line = line.split()
...         x, y, z = float(line[1]), float(line[2]), float(line[3])
...         forces.append((x,y,z))
```

- other way to check for substrings:
  - `str.startswith()`, `str.endswith()`
- Python has also an extensive support for regular expressions in `re` -module

## String formatting

- Output is often wanted in certain format
- The string object has **.format** method for placing variables within string
- Replacement fields surrounded by **{}** within the string

```
>>> x, y = 1.6666, 2.33333
print "X is {0} and Y is {1}".format(x, y)
X is 1.6666 and Y is 2.3333
>>> print "Y is {1} and X is {0}".format(x, y)
Y is 2.3333 and X is 1.6666
```

- Possible to use also keywords:

```
>>> print "Y is {val_y} and X is {val_x}".format(val_x=x, val_y=y)
Y is 2.3333 and X is 1.6666
```

## String formatting

- Presentation of field can be specified with `{i:[w][.p][t]}`
  - `w` is optional minimum width
  - `.p` gives optional precision (=number of decimals)
  - `t` is the presentation type
- some presentation types
  - `s` string (normally omitted)
  - `d` integer decimal
  - `f` floating point decimal
  - `e` floating point exponential

```
>>> print "X is {0:6.3f} and Y is {1:6.2f}".format(x, y)  
X is  1.667 and Y is  2.33
```

## Writing to a file

- data can be written to a file with **print** statements
- file objects have also a **write()** function
- the **write()** does not automatically add a newline

output.py

```
outfile = open('out', 'w')
print >> outfile, "Header"
print >> outfile, "{0:6.3f} {0:6.3f}".format(x, y)

outfile = open('out', 'w')
outfile.write("Header\n")
outfile.write("{0:6.3f} {0:6.3f}".format(x, y))
```

- file should be closed after writing is finished

## Differences between Python 2.X and 3.X

- print is a function in 3.X

differences.py

```
print "The answer is", 2*2    # 2.X  
print("The answer is", 2*2)  # 3.X
```

```
print >>sys.stderr, "fatal error"    # 2.X  
print("fatal error", file=sys.stderr) # 3.X
```

- in 3.X some dictionary methods return “views” instead of lists.
  - e.g. `k = d.keys(); k.sort()` does not work,  
use `k = sorted(d)` instead
- for more details, see  
<http://docs.python.org/release/3.1/whatsnew/3.0.html>

## Summary

- files are opened and closed with `open()` and `close()`
- lines can be read by iterating over the file object
- lines can be split into lists and check for existence of specific substrings
- string formatting operators can be used for obtaining specific output
- file output can be done with `print` or `write()`

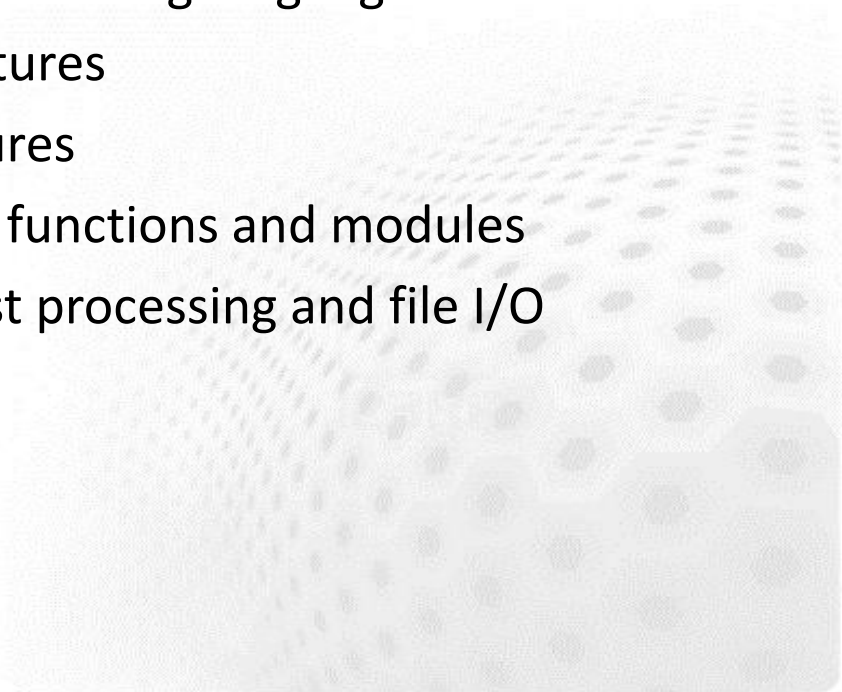


## Useful modules in Python standard library

- math : “non-basic” mathematical operations
- os : operating system services
- glob : Unix-style pathname expansion
- random : generate pseudorandom numbers
- pickle : dump/load Python objects to/from file
- time : timing information and conversions
- xml.dom / xml.sax : XML parsing
- + many more

<http://docs.python.org/library/>

## Summary

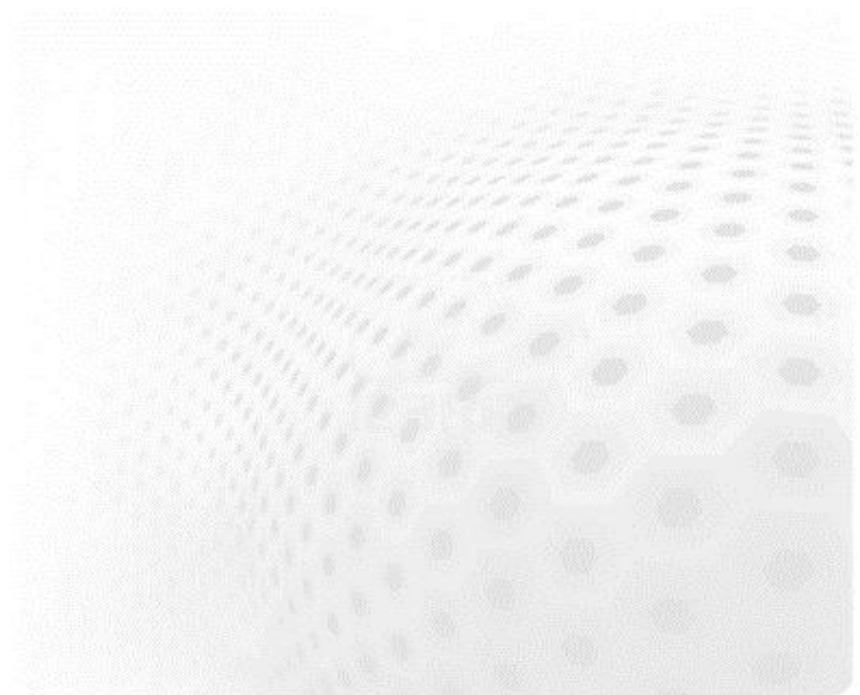
- Python is dynamic programming language
  - flexible basic data structures
  - standard control structures
  - modular programs with functions and modules
  - simple and powerful text processing and file I/O
  - rich standard library
- 

# **OBJECT ORIENTED PROGRAMMING WITH PYTHON**



# Object oriented programming with Python

- Basic concepts
- Classes in Python
- Inheritance
- Special methods



## OOP concepts

- OOP is programming paradigm
  - data and functionality are wrapped inside of an “object”
  - Objects provide methods which operate on (the data of) the object
- Encapsulation
  - User accesses objects only through methods
  - Organization of data inside the object is hidden from the user

## Examples

- String as an object
  - Data is the contents of string
  - Methods could be lower/uppercasing the string
- Two dimensional vector
  - Data is the x and y components
  - Method could be the norm of vector

## OOP in Python

- In Python everything is a object
- Example: **open** function returns a file object
  - data includes e.g. the name of the file

```
>>> f = open('foo', 'w')  
>>> f.name  
'foo'
```

- methods of the file object referred by **f** are **f.read()**, **f.readlines()**, **f.close()**, ...
- Also lists and dictionaries are objects (with some special syntax)



## OOP concepts

### ● class

- defines the object, i.e. the data and the methods belonging to the object
- there is only single definition for given object type

### ● instance

- there can be several instances of the object
- each instance can have different data, but the methods are the same

## Class definition in Python

- When defining class methods in Python the first argument to method is always **self**
- **self** refers to the particular instance of the class
- **self** is not included when calling the class method
- Data of the particular instance is handled with **self**

students.py

```
class Student:
    def set_name(self, name):
        self.name = name

    def say_hello(self):
        print "Hello, my name is ", self.name
```

## Class definition in Python

students.py

```
class Student:
    def set_name(self, name):
        self.name = name
    def say_hello(self):
        print "Hello, my name is ", self.name

# creating an instance of student
stu = Student()
# calling a method of class
stu.set_name('Jussi')
# creating another instance of student
stu2 = Student()
stu2.set_name('Martti')
# the two instances contain different data
stu.say_hello()
stu2.say_hello()
```

## Passing data to object

- Data can be passed to an object at the point of creation by defining a special method `__init__`
- `__init__` is always called when creating the instance

students.py

```
class Student:
    def __init__(self, name):
        self.name = name
    ...
```

- In Python, one can also refer directly to data attributes

```
>>> from students import Student
>>> stu1 = Student('Jussi')
>>> stu2 = Student('Martti')
>>> print stu1.name, stu2.name
'Jussi', 'Martti'
```

## Python classes as data containers

- classes can be used for C-struct or Fortran-Type like data structures

students.py

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

- instances can be used as items in e.g. lists

```
>>> stu1 = Student('Jussi', 27)
>>> stu2 = Student('Martti', 25)
>>> student_list = [stu1, stu2]
>>> print student_list[1].age
```

## Encapsulation in Python

- Generally, OOP favours separation of internal data structures and implementation from the interface
- In some programming languages attributes and methods can be defined to be accessible only from other methods of the object.
- In Python, everything is public. Leading underscore in a method name can be used to suggest “privacy” for the user

## Inheritance

- New classes can be derived from existing ones by inheritance
- The derived class “inherits” the attributes and methods of parent
- The derived class can define new methods
- The derived class can override existing methods



# Inheriting classes in Python

```
inherit.py
class Student:
    ...

class PhDStudent(Student):
    # override __init__ but use __init__ of base class!
    def __init__(self, name, age, thesis_project):
        self.thesis = thesis_project
        Student.__init__(self, name, age)

    # define a new method
    def get_thesis_project(self):
        return self.thesis

stu = PhDStudent('Pekka', 20, 'Theory of everything')
# use a method from the base class
stu.say_hello()
# use a new method
proj = stu.get_thesis_project()
```



## Special methods

- class can define methods with special names to implement operations by special syntax (operator overloading)
- Examples
  - `__add__`, `__sub__`, `__mul__`, `__div__`
    - for arithmetic operations (+, -, \*, /)
  - `__cmp__` for comparisons, e.g. sorting
  - `__setitem__`, `__getitem__` for list/dictionary like syntax using []

## Special methods

special.py

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

    def __add__(self, other):
        new_x = self.x + other.x
        new_y = self.y + other.y
        return Vector(new_x, new_y)

v1 = Vector(2, 4)
v2 = Vector(-3, 6)
v3 = v1 + v2
```

special.py

```
class Student:
    ...
    def __cmp__(self, other):
        return cmp(self.name,
                    other.name)

students = [Student('Jussi', 27),
            Student('Aaron', 29)]
students.sort()
```

## Summary

- Objects contain both data and functionality
- class is the definition of the object
- instance is a particular realization of object
- class can be inherited from other class
- Python provides a comprehensive support for object oriented programming (“Everything is an object”)