Jussi Enkovaara
Jyry Suvilehto

Introduction to Python

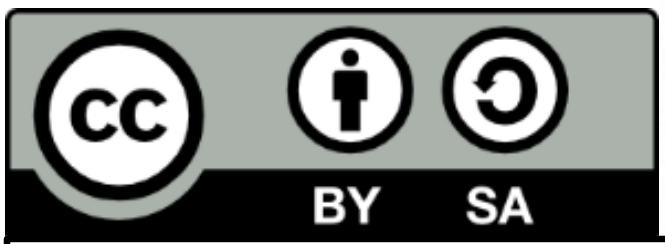
September 18-19, 2017
CSC – IT Center for Science Ltd, Finland

```
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()
```



Agenda

Monday

9:00-9:45	Introduction to Python
9:45-10:30	Exercises
10:30-10:45	Coffee Break
10:45-11:15	Control structures
11:15-12:15	Exercises
12:15-13:00	Lunch break
13:00-13:30	Functions and exceptions
13:30-14:30	Exercises
14:30-14:45	Coffee Break
14:45-15:15	Modules
15:15-16:15	Exercises

Tuesday

9.00-9.45	File I/O and string processing
9.45-10.30	Exercises
10.30-10.45	Coffee break
10:45-11:30	NumPy and simple plotting
11:30-12:15	Exercises
12.15-13.00	Lunch break
13.00-14:00	Object oriented programming
14:00-14:30	Exercises
14.30-14.45	Coffee break
14.45-16.15	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
  - Version 2 spread so wide that adopting version 3 has been a long struggle
  - This course uses 3.x version
  - It is possible and good practice to write programs compatible with both versions

Python Essentials

- Development started in 1989 by Guido van Rossum
  - 2.0 released in 2000, 3.0 in 2008
- Language development overseen by the Python Software Foundation (PSF)
- Most public Python packages available in the Python Package Index (PyPI)

image: Daniel Stroud
CC-BY-SA 4.0

python SOFTWARE FOUNDATION

Python language development

- There is a process for new language features called Python Enhancement Proposals
- Guido (a Dutchman) still retains final say over new features as the *Benevolent Dictator For Life*

```
There should be one-- and
preferably only one --obvious
way to do it.

Although that way may not be
obvious at first unless you're
Dutch.

- quote from Zen of Python
```

Python ecosystem

- Core Python language features and built-ins
  - Always available, very robust if used correctly
- The Python standard library
  - Available in most standard installations, need to be imported to use
  - Well tested and fairly stable
  - Covers a lot of basic needs, ”batteries included”
- The Python Package Index
  - A package exists for almost everything
  - May contain bugs, possibly not actively maintained
  - May solve your problem or sub-problem out of the box

Why Python?

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

Why not Python?

- Not energy-efficient
  - A factor in doing web-scale things
- Emphasizes programmer cognitive load and development over being close to hardware
  - mitigated by libraries that interface with lower level components
  - Not possible to use on some embedded systems
- Not backed by a single large commercial entity
- The programmer is generally trusted, so not very suitable for security-oriented systems
  - Or security arrangements have to be made at other levels

Information about Python

- www.python.org
- H. P. Langtangen, “Python Scripting for Computational Science”, Springer
- www.scipy.org

FIRST GLIMPSE INTO 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:

```
$ python3
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("hello")
hello
```

- Useful for testing and learning

Enhanced interactive Python interpreter

- The IPython shell is a more powerful tool for using Python from the command line
  - Autocompletion, details about objects
  - We mostly use the browser version on this course

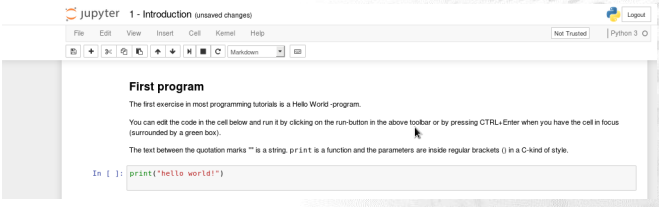
```
$ ipython
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: print("hello")
hello
```

Jupyter notebooks

- Python environment in a web application
- Notebook can contain progam code, explanatory text, and visualisations



Python syntax

- Variable and function names start with a letter and can contain also numbers and underscores, e.g "my_var", "my_var2"
- Variables may also begin with an underscore but these will be covered later
- Python is case sensitive
- Code blocks are defined by indentation
  - a block is started by a colon :
- Comments start with #

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")
```

Python style

variable_name

function_name

ClassName

modulename.submodule or
module_name.another_module

Data types

- Python is a dynamically typed language
  - no type declarations for variables
  - easy to make typing errors in variable names
- Variable does have a type
  - incompatible types cannot be combined
  - "In the face of ambiguity, refuse the temptation to guess." - zen of python

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)
```

Strings

- Strings are enclosed by " or '
- Multiline strings can be defined with three double quotes
  - this will include newlines
- Strings inside the same parentheses will be interpreted as one string

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."""
s6 = ("This string "
      "has no newlines "
      "in it")
```

Strings

- + and * operators with strings:

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

Data structures

- Sequence structures
  - Lists
  - Tuples
  - Sets
  - Frozen sets
  - Byte arrays
- Dictionaries

Mutable vs immutable data types

- Mutable data types can be modified after creation e.g.
  - a list can be added to
  - a byte array can be edited
  - a dict can be added to
- Immutable data types cannot be modified, modifying them returns a new object of the same type
  - The original is not changed
- There is no **++**-operator as numeric types are immutable

Immutable	Mutable
numeric types	
tuple	list
str	byte array
frozen set	set
	dict

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

```
>>> 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]
```

- Accessing list elements
- + and * operators with lists

```
>>> [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}
```

Converting between basic data types

- Python has built-in functions
  - **type()** for determining the type of a variable
  - **int(), float(), str(), list(), dict(), set(),** etc. for converting to each type
    - only if possible

```
>>> x = 5
>>> type(x)
<class 'int'>
>>> y = str(x)
>>> type(y)
<class 'str'>
>>> type(z)
<class 'float'>
>>> z
5.0
>>> x + z
10.0
>>> y+z
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
TypeError: Can't convert 'float'
object to str implicitly
```

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]
```

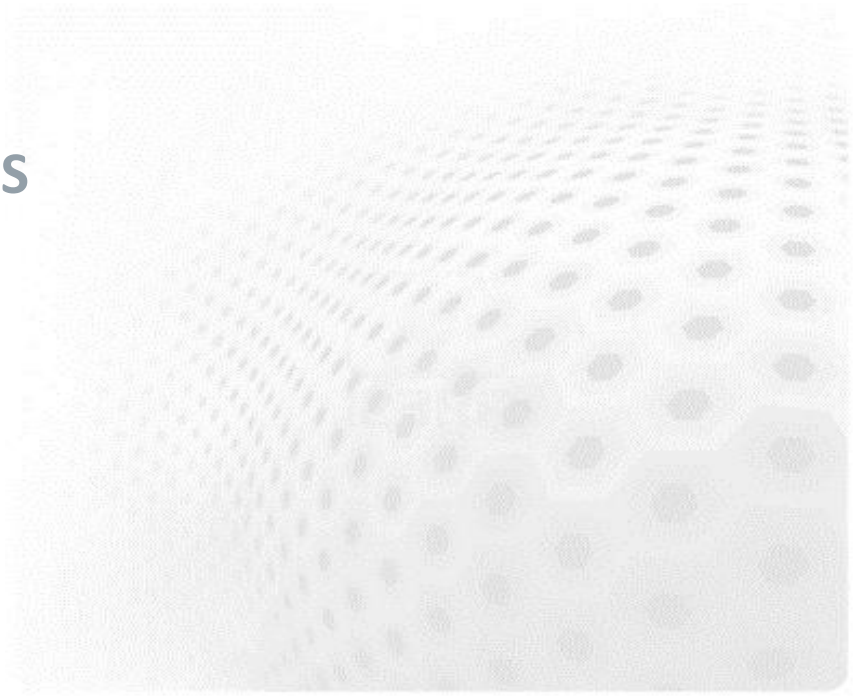
What is an object?

- 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 an object
- Methods of object are called with the syntax: obj.method(parameter, parameter)
- 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 an object in Python
- Python variables are always references to objects

CONTROL STRUCTURES



Control structures

- if – else statements
- while loops
- for loops

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
```

- comparison operators: ==, !=, >, <, >=, <=
- boolean operators: and, or, not

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 a switch statement

if statement - oddities

- empty lists, empty strings and 0 evaluate to False for the truth statement

example.py
```
x = []
y = None
z = 0
if not x :
    print("x")
if not y :
    print("y")
if not z :
    print("z")
```

if statement - oddities

- collections and sequences support the **in** keyword

example.py
```
x = "01245"
if "6" in x :
    print("x contains '6'")
elif "5" < 0:
    print("x contains '5'")
else:
    print("x is something completely different")
```

- operator priority can be affected with parentheses
  - preferable to keep it simple

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
```

range

- Indices are not idiomatic when they can be avoided
- sometimes they can't be avoided

example.py
```
print("counting backwards from 10 in steps of 2")
for number in range(10, 1, -2) :
    print(number)
```

- range expresses a range of integers with configurable start, stop and step parameters
  - Counting up from 0 with single parameter (stop)
- In Python 3 **range** returns an iterator, in Python 2 a list

enumerate

- Built-in function enumerate returns tuples of index, [item]

example.py
```
my_list =  ["a", "b", "c", "d", "e"]
for index, str_ in enumerate(my_list):
    print("index " +  str(index) + " is str " + str(str_))
```

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 EXCEPTIONS

Functions

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

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 positional arguments
  - their order matters
- 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=0, y=0):
    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
- keyword arguments must always be after positional arguments
  - in function definition
  - in function call
- keyword arguments don't need to be in any particular order

Keyword 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 intentional or accidental

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

- Exceptions are catched 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')
```

Exceptions and functions

- Exceptions are a way to return information other than a return value from a function

example.py

```
def divide(first, second):
    if second == 0:
        raise ZeroDivisionError("user tried to divide by zero")
    return first/second

try:
    first= input("Give first number: ")
    second = input("Give second number")
    print(divide(first, second ))
except ZeroDivisionError:
    print("execution protected from user error")
except Exception:
    print("unknown error occurred!")
```

Error vs Exception

- Not all Exceptions are Errors
  - An error means something went wrong
  - There are exceptions that have other significance, e.g. StopIteration, KeyboardInterrupt or GeneratorExit
- Most of the time programmers are interested in the errors

Generator expressions

- It is often the case that each item of a return value can be processed independently from the rest of the items
- It may not be wise to create large data structures only to immediately destroy them
- Instead of returning a list or similar iterable, you can use the yield keyword
- Best used to create items one by one to be consumed one by one
- Sometimes the performance difference is significant

Generator expressions

- When the next value is requested, the execution continues from where it left off
- The use of yield instead of return makes this a generator function

function.py

```
def odd_numbers(n):
    counter = 0
    for i in range(n):
    if i % 2 == 1:
       yield result
       counter += 1
    else:
        pass
    print("yielded %d odd numbers" % counter )

sum(odd_numbers(10000000))
```

Summary

- functions help in reusing frequently used code blocks
- functions can have positional and keyword arguments
- functions often return values
- functions (and code in general) can raise exceptions, most common of which are errors

MODULES AND PACKAGES

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
- Third party modules can be installed via system package manager or pip
- User defined modules
  - Python searches for modules
    - in current working directory
    - $PYTHONPATH
    - relative to the python executable

Importing modules / from 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
- in fact, it is recommended to do so to give structure to your code
  - helps to write reusable code
  - forces you to thing about what functionality goes where

Creating modules

- it is possible to make imports from own modules
- every .py file in the directory you are in can be imported from
- define a function in file mymodule.py
- the function can now be imported in other .py files:

mymodule.py

```
def incx(x):
    return x+1
```

test.py

```
import mymodule

y = mymodule.incx(1)
```

test.py

```
from mymodule import incx

y = incx(1)
```

Module structure

- the presence of file __init__.py creates a module
  - file can be empty
- structure improves readability

test.py
project
    __init__.py
    module_alpha.py
    module_beta.py
    submodule
        __init__.py
        module_delta.py

test.py

```
import project.module_alpha

# the "as" keyword can be helpful when names collide or if name is long
from project.module_alpha import function_alpha as alpha

from project.submodule.module_delta import function_delta
```

Controlling imports (advanced)

- To control what is imported by "import *" you can expose the functions you want in __init__.py
- By convention functions and variables beginning with an underscore _ are not imported
  - This is a hint that someone using your code shouldn't use the function/variable

__init__.py

```
from alpha import first
from alpha import third
```

alpha.py

```
def first():
    ...
def second():
    ...
def third():
    ...
```

test.py

```
from mymodule.alpha import *
# first and third are defined but
# second is not
```

Packages

- Python code is often distributed in packages
- Most python packages are published in the Python Package Index (PyPI), https://pypi.python.org/pypi
- A tool called *pip* can be used to install packages

```
$ pip install --user hypothesis
Collecting hypothesis
  Downloading hypothesis-3.26.0.tar.gz (113kB)
    100% |████████████████████████████████| 122kB 3.3MB/s
Requirement already satisfied: enum34 in
/Users/suvileht/Library/Python/2.7/lib/python/site-packages (from hypothesis)
Installing collected packages: hypothesis
  Running setup.py install for hypothesis ... done
Successfully installed hypothesis-3.26.0
$
```

pip and virtualenv

- Usually only administrators can install packages for all system users
- You can use the *--user* handle of pip to install only to yourself
- Often one software requires library X to be < 1.0 and another for it to be > 1.5!
- It is often a good idea to isolate requirements for different software installations
- A tool called *virtualenv* is used to create virtual environments

Virtualenv example

```
$ virtualenv example
New python executable in /private/tmp/example/bin/python
Installing setuptools, pip, wheel...done.
$ source example/bin/activate
(example) $ pip install biopython
Collecting biopython
  Downloading biopython-1.70-cp27-cp27m-macosx_10_6_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl (2.1MB)
    100% |████████████████████████████████| 2.1MB 298kB/s
Collecting numpy (from biopython)
  Downloading numpy-1.13.1-cp27-cp27m-macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl (4.6MB)
    100% |████████████████████████████████| 4.6MB 158kB/s
Installing collected packages: numpy, biopython
Successfully installed biopython-1.70 numpy-1.13.1
(example) $ python
Python 2.7.10 (default, Feb  7 2017, 00:08:15)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.34)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from Bio.Seq import Seq
>>> my_seq = Seq("AGTACACTGGT")
>>> print(my_seq)
AGTACACTGGT
>>>
(example) $ deactivate
$
```

Summary

- Modules are used to structure code
- Modules can be single files or multiple structures
- The import keyword is used to activate code from modules

- The pip command is used to install packages
- It is recommended to isolate package installations to virtual environments

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 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
```

with

- special syntax for letting the interpreter take care of closing the file after use
- ensures that file is closed, even if errors occur inside the with-statement
- other resources can also be accessed using a with-statement

output.py

```
with open('out', 'r') as infile:
  for line in infile:
   print(line.strip().reverse())
```

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 lines 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
```

String formatting - old style

- Python also has an older style of string formatting, which is still valid but not recommended
- **printf**-style syntax
- Mentioned so you'll recognize it if you see it

```
>>> var = X is %d and Y is %s" % (5, "foo")
>>> print(var)
X is  5 and Y is foo
```

Writing to a file

- data can be written to a file with print statements
- file objects have also a write() function
- the `outfile.write()` does not automatically add a newline
- file should be closed after writing is finished or the with-keyword used

output.py

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

outfile = open('out_2', 'w')
outfile.write("Header\n")
outfile.write("{0:6.3f} {0:6.3f}".format(x, y))
outfile.close()
```

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
```

Summary

- files are opened and closed with open() and close()
  - You can handle context using the with-statement
- 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
- json: JSON parsing
- + many more
  http://docs.python.org/library/

Useful external modules

- requests: HTTP request handling
- BeautifulSoup: HTML parsing and scraping
- Pillow: image handling
- OpenCV: Computer vision (e.g. for OCR)

NUMPY

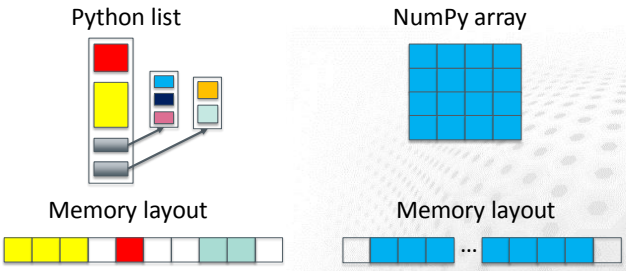
Numpy – fast array interface

- Standard Python is not well suitable for numerical computations
  - lists are very flexible but also slow to process in numerical computations
- Numpy adds a new **array** data type
  - static, multidimensional
  - fast processing of arrays
  - some linear algebra, random numbers

Numpy arrays

- All elements of an array have the same type
- Array can have multiple dimensions
- The number of elements in the array is fixed, shape can be changed

Python list vs. NumPy array



Creating numpy arrays

- From a list:

```
>>> import numpy as np
>>> a = np.array((1, 2, 3, 4), float)
>>> a
array([ 1.,  2.,  3.,  4.])
>>>
>>> list1 = [[1, 2, 3], [4,5,6]]
>>> mat = np.array(list1, complex)
>>> mat
array([[ 1.+0.j,  2.+0.j,  3.+0.j],
       [ 4.+0.j,  5.+0.j,  6.+0.j]])
>>> mat.shape
(2, 3)
>>> mat.size
6
```

Creating numpy arrays

- More ways for creating arrays:

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>>
>>> b = np.linspace(-4.5, 4.5, 5)
>>> b
array([-4.5 , -2.25,  0.  ,  2.25,  4.5 ])
>>>
>>> c = np.zeros((4, 6), float)
>>> c.shape
(4, 6)
>>>
>>> d = np.ones((2, 4))
>>> d
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

Indexing and slicing arrays

- Simple indexing:

```
>>> mat = np.array([[1, 2, 3], [4, 5, 6]])
>>> mat[0,2]
3
>>> mat[1,-2]
>>> 5
```

- Slicing:

```
>>> a = np.arange(5)
>>> a[2:]
array([2, 3, 4])
>>> a[:-1]
array([0, 1, 2, 3])
>>> a[1:3] = -1
>>> a
array([0, -1, -1, 3, 4])
```

Indexing and slicing arrays

- Slicing is possible over all dimensions:

```
>>> a = np.arange(10)
>>> a[1:7:2]
array([1, 3, 5])
>>>
>>> a = np.zeros((4, 4))
>>> a[1:3, 1:3] = 2.0
>>> a
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  2.,  2.,  0.],
       [ 0.,  2.,  2.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

Views and copies of arrays

- Simple assignment creates references to arrays
- Slicing creates “views” to the arrays
- Use copy() for real copying of arrays

example.py

```
a = np.arange(10)
b = a              # reference, changing values in b changes a
b = a.copy()       # true copy

c = a[1:4]         # view, changing c changes elements [1:4] of a
c = a[1:4].copy() # true copy of subarray
```

Array operations

- Most operations for numpy arrays are done element-wise
  - +, -, *, /, **

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a * b
array([ 2.,  4.,  6.])
>>> a + b
array([ 3.,  4.,  5.])
>>> a * a
array([ 1.,  4.,  9.])
```

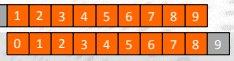
Array operations

- Numpy has special functions which can work with array arguments
  - sin, cos, exp, sqrt, log, ...

```
>>> import numpy, math
>>> a = numpy.linspace(-math.pi, math.pi, 8)
>>> a
array([-3.14159265, -2.24399475, -1.34639685, -0.44879895,
          0.44879895, 1.34639685,  2.24399475,  3.14159265])
>>> numpy.sin(a)
array([ -1.22464680e-16,  -7.81831482e-01,  -9.74927912e-01,
         -4.33883739e-01,   4.33883739e-01,   9.74927912e-01,
          7.81831482e-01,   1.22464680e-16])
>>>
>>> math.sin(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: only length-1 arrays can be converted to Python scalars
```

Vectorized operations

- for loops in Python are slow
- Use “vectorized” operations when possible
- Example: difference



example.py

```
# brute force using a for loop
arr = np.arange(1000)
dif = np.zeros(999, int)
for i in range(1, len(arr)):
    dif[i-1] = arr[i] - arr[i-1]

# vectorized operation
arr = np.arange(1000)
dif = arr[1:] - arr[:-1]
```

  - for loop is ~80 times slower!

I/O with Numpy

- Numpy provides functions for reading data from file and for writing data into the files
- Simple text files
  - numpy.loadtxt
  - numpy.savetxt
  - Data in regular column layout
  - Can deal with comments and different column delimiters

Linear algebra

- Numpy can calculate matrix and vector products efficiently: dot, vdot, ...
- Eigenproblems: linalg.eig, linalg.eigvals, ...
- Linear systems and matrix inversion: linalg.solve, linalg.inv

```
>>> A = np.array(((2, 1), (1, 3)))
>>> B = np.array(((-2, 4.2), (4.2, 6)))
>>> C = np.dot(A, B)
>>>
>>> b = np.array((1, 2))
>>> np.linalg.solve(C, b)  # solve C x = b
array([ 0.04453441,  0.06882591])
```

Numpy performance

- Matrix multiplication
  C = A * B
  matrix dimension 200
- pure python: 5.30 s
- naive C: 0.09 s
- numpy.dot: 0.01 s

Summary

- Numpy provides a static array data structure
- Multidimensional arrays
- Fast mathematical operations for arrays
- Arrays can be broadcasted into same shapes
- Tools for linear algebra and random numbers

SIMPLE PLOTTING WITH MATPLOTLIB

Matplotlib

- 2D plotting library for python
- Can be used in scripts and in interactive shell
- Publication quality in various hardcopy formats
- “Easy things easy, hard things possible”
- Some 3D functionality

Matplotlib interfaces

- Simple command style functions similar to Matlab

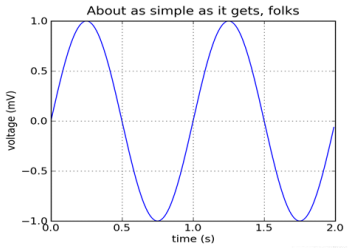
plot.py
```
import matplotlib.pyplot as plt
...
plt.plot(x, y)
```

- Powerful object oriented API for full control of plotting

Basic concepts

- Figure: the main container of a plot
- Axes: the “plotting” area, a figure can contain multiple Axes
- graphical objects: lines, rectangles, text
- Command style functions are used for creating and manipulating figures, axes, lines, ...
- The command style interface is stateful:
  - track is kept about current figure and plotting area

Simple plot



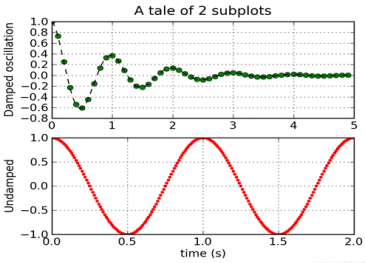
plot.py
```
import matplotlib.pyplot as plt
...
plt.plot(x, y)
plt.title(‘About as simple’)
plt.xlabel(‘time (s)’)
```

- plot : create a simple plot. Figure and axes are created if needed

Interactive vs. batch mode

- In many installations batch mode is default
  - Figures do not show up without show() function
  - Batch mode is useful e.g. for writing out files during simulation and for heavy rendering
- Mode can be controlled as:
  - ion() : turn on interactive mode
  - ioff() : turn on interactive mode
- IPython has more extensive support for interactive usage
  - %matplotlib magic command
  - Start as “ipython - -matplotlib”

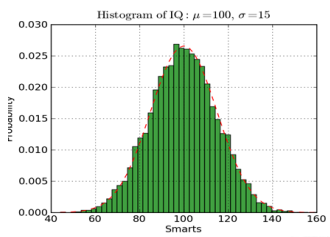
Multiple subplots



subplot.py
```
plt.subplot(211) #2x1 plot, use 1st
plt.plot(x, y1)
…
plt.subplot(212) #use 2nd
plt.plot(x, y2)
```

- subplot : create multiple axes in the figure and switch between subplots

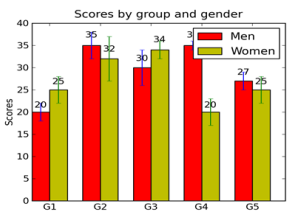
Histograms

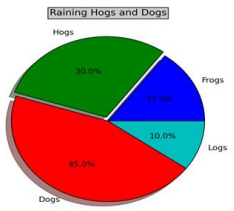


histogram.py
```
mu, sigma = 100, 15
x = mu + sigma*np.random.rand(1000)
plt.hist(x, 50)
#use raw strings for Latex
plt.title(r’Distribution, $\mu$)
```

- hist : create histogram
- Latex can be used with matplotlib

Bar and pie charts





- bar : bar charts
- pie : pie charts

Summary of basic functions

- Simple plot: plot
- Interactive vs. batch mode: ion / ioff
- Hardcopies: savefig
- Multiple plots: subplot
- Histograms: hist
- Bar charts: bar
- Pie charts: pie
- Switch plotting on top of existing figure: hold
- Contour plots: contour, contourf

Summary

- Matplotlib provides a simple command style interface for creating publication quality figures
- Interactive plotting and different output formats (.png, .pdf, .eps)
- Simple plots, multiplot figures, decorations
- Possible to use Latex in text

OBJECT ORIENTED PROGRAMMING WITH PYTHON

Object oriented programming with Python

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

Programming paradigms

- There is no one right way to structure programs
- Some ways have been found to be better than others, especially because of
  - reusability
  - testability
  - readability
  - programmer efficiency
- Multiple programming paradigms exist and Python actually supports several

OOP concepts

- Object Oriented Programming (OOP) is programming paradigm
  - data and functionality are wrapped inside of an “object”
  - Objects provide methods which operate on (the data of) the object
    - Method is a function that is tied to the data of an 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

Examples

- An object in a 3D game
  - Data are the location of the object and it's shape, color, state, etc.
  - Methods could be interacting with the object
  - Multiple objects may be parts of a larger object
- A bank account in banking software
  - Data are the account number, account balance and a list of account transactions
  - A transaction needs to be it's own type of object
  - A likely method is recording a new transaction on an account

OOP in Python

- In Python ***everything*** is an 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)
- Even functions are objects!

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
  - e.g. the string-class
- instance
  - there can be several instances of the object
  - each instance can have different data, but the methods are the same
  - e.g. string instances "example" and "foobar"

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(object):
    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(object):
    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(object):
    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(object):
    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.
- Python does not enforce encapsulation
- Leading underscore in a method or data attribute name can be used to hint that it is not intended for external use

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
- All classes should explicitly inherit the "object" class

Inheritance - example

- Suppose you are making a mobile game with multiple objects that are all drawn in 2D → class WorldObject
- WorldObjects have a location and a shape and they can move (have speed and direction) and they can collide with other world objects, they can also draw themselves on a display using a method draw()
- You then create object types Bird and Pig, which inherit WorldObject but are drawn differently and a Bird colliding with a Pig destroys the Pig, they override the method collide() and draw()
- Then you create multiple subclasses of Bird which perform differently
- Through all this you only had to handle movement and other basic stuff once

Inheriting classes in Python

inherit.py

```
class Student(object):
...

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

Multiple inheritance

- In Python classes can inherit from multiple classes
- Very powerful when used correctly
  - Possible to create mixin classes
- Method Resolution Order determines in which order the calls to super() are processed

inherit.py

```
class Vehicle(object):
...
class Car(object):
...

class Volvo(Vehicle, Car):
...
```

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(object):
    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(object):
    ...
    def __lt__(self, other):
        return self.age < other.age

    def __eq__(self, other):
        return self.age == other.age


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

Exceptions

- Exceptions are just classes that inherit from Exception
- Creating new exceptions is easy, because most of the things are handled in the parent classes

exceptions.py

```
def addition(a, b):
    return a + b

try:
    addition("5", 10)
except TypeError as ex:
    print("Error: ", str(ex))


class BadDayException(Exception):
    pass

raise BadDayException("programmer "
    "is having a bad day")
```

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(es)
- Python provides a comprehensive support for object oriented programming (“Everything is an object”)
- Exceptions inherit from the base class Exception
- Python is a *multiparadigm* programming language, you don't have to use classes
  - Most programmers know OOP so it may be the path of least resistance to follow it in larger projects