# SCRIPTING LANGUAGES

# -

# PYTHON

# TODAY

- How to invoke python interpreter
  - what does it actually do
  - useful options
  - a trivial use of python
- Language overview
  - types
  - basic operations
  - collection types
  - control
  - functions
  - modules
  - strings and files
- Python modules
  - Not exhaustive list of useful packages
- Next lecture
  - Invoking external programs
  - Using C/C++ code in python

# HOW TO INVOKE PYTHON

- **python** – yes, that's it
  - you get prompt (usually **>>>**)
  - now you can enter instructions
  - i.e. **print 2+2**
  - you can quit typing **Crtl-D** → python reads the input till the end of file
- Instruction you have entered by hand can be put it the file
  - say: test.py and can be invoked in one go:
    **python test.py**
- Instructions (quick&small scripts) can be given as command line:
  **python –c "print 2+2"**
- Or entire program can be read from command line (i.e. redirected from some other place):
  **cat code | python -**
- Or run the script and enter interactive mode afterwards:
  **python test.py -i**

# VERSIONS

- **python** command is usually a link to some version of the interpreter
  → it is the version of the language
- Now the python moved from the 2.X line to 3.0
  - This is major change! Mainly because it affects many lines of trivial code. Easy to fix … but still many lines to go through.
- Evolution of the python is (auto/demo)cratic, everyone can propose PEP (Python Enhancement Proposal) – this suggestion is discussed and often accepted, but the last word used to belong to Guido van Rossum
- The name of language is coming from the "Monthy Python" … series, but is also meant to mean it is very flexible

# STRUCTURE OF THE SCRIPT

- The spaces/tabs matter in python
  - but only the ones at the **very beginning** of the line
  - in the **hello.py** script there should be no space nor <tab> in front

**print “Hello world”**

- In case of conditionals (if/for/while) the code which needs to be invoked in the condition/loop must be indented with *<tab>*

```
if 2 == 2:
	print “magic, pure magic”
	if 3 == 2:
		print ”what? this is crazy”
	print ”uff, at least this is not true”
```

THAT MAKES LONG PROCEDURES VERY DIFFICULT TO COMPREHEND.

YES

WRITE SHORT ONES!

# VARIABLES

- Do not have type by themselves
- They do have assigned value which has type

Literals

- **"string"** can also use single quotes **'string'**
  use this two to avoid escaping "helo's"
  - there is triple """    """ for multi-line strings
    i.e. documentation
- 2 integer, **2L** long integer (big numbers)
- 3.14 - double precision
- True False – Boolean types
- You can check the type like this: **type(a)**

# SPECIAL VALUE NONE AND DISPOSAL OF THE VARIABLE

- In order to denote unassigned variable one can use special value: **None**
  **a=None**
  - The variable “a” exists. It references special value **None**.
- When you need to remove the variable do this:
  **del a**
  - referencing variable “a” from that point causes interpreter error

# BASIC OPERATIONS: ASSIGNMENT

Variable name on the left anything on the right

**a=2**

**a='hello'**

At he second assignment the "2" is lost!
Do not worry about memory leaks!

Each value is in fact reference with use counting (sort of smart pointer).
If given value is not pointed (assigned to) by any variable → no one is using it → it is disposed right away.
If two variables point to the same value they both can change the same value.

```
a=87
b=a
a=7
print a, b # what is the result
```

Hmm!
Simple thing,
like assignment,
is so complex!?

# DO NOT WORRY!

- It is in fact very natural what happens.
  Very rarely you would need to even think about it.

# BASIC OPERATIONS: CONTINUED

- "+" - works also for strings if the operand on the right is string
- "*" – works also for strings if the right operand is an integer
- "- /" – as you expect, no meaning for strings
- Rules of numerical conversions as in other languages:

  3/2 is 1, 2/3 is 0

String formatting operator: **%**

**"Alice has %d cats." % 4** – this language feature disappears from 3.0 standard

**"Alice {0:d} cats" .format(4)** – in 3.0 – there is much more here

# CONTAINERS

- Python is equipped essentially with 4 build-in types for storing multiple "objects".
- They are very efficient and should be used as much as possible.
- In fact more advanced functionalities are often based on these elementary "containers".

# TUPLE

- Sequence (order matters)
  of immutable (you **can not** change them)
  objects (you can have tuples of whatever)

**a = (”Hello”, ”everyone”, ”in”, ”the”, ”room”, 206)**

**b=(1,)**

**c=(”hello”, (1,2,7) ) # nested tuple**

**q=4,5,7,6,”my pin code”, b # a quick way, and variable b used to make tuple**

You can access the content very easily:

**c[0]**

**c[1]**

**c[1][1]**

**~~c[0]=”bla”~~ # nope, this would mean c is mutable**

# LIST

- Sequence (order matters)
  of mutable (you **can change** them)
  objects (you can have lists of whatever)

**a=[] # an empty list**

**b=['hello', 2, 'nd','time']**

**b[0]='Hi' # this works!  Unlike for the tuples!**

# A BIT MORE ON INDEXING

- Square brackets [] can be used to address elements of tuple or list

```
s[0] # indexing starts always from 0

len(s) # gives length

s[len(s)-1] # is last element

s[-1] # is way more convenient way of accessing last
   element, s[-2] also works

s[0:3] # is a slice of elements: 0,1,2

s[0:10:2] # every second element from 0 to 9
```

# CONSEQUENCES OF LISTS MUTABILITY

- The API is richer as compared to tuples

```
a=[1,4,5]

a.append(9) # the . means on the list a invoke it's method append with the argument 9

a.extend([1,5])

a.insert(2, 7) # at position 2 insert 7

a.remove(4) # removes first occurrence of the 4

a.pop() # remove last element from the list and return it

a.reverse()

a.sort()
```

# DICTIONARY/MAP

- Set (order does not matter)
  of pairs key & value objects (can use anything as index and value)
  with the mutable value (can change the value )

- The most useful container type in python is dictionary

```
a={} # an empty dictionary

a[5] = 8 # under the index 5 is 8

a['hi']=8 # the same value is under the index "hi"

a[2.75] =8 # even the floating point number can be a key

a[4]={1:4, 8:88} # nesting is equally easy
```

# DICTIONARY API

- **[key] or get(key)** – access or assignment
- **has_key(key)** - True if key in the dict
- **values(), keys(), items()** – converts dicts to the lists of values, keys, and tuples (key, value)
- **iteritems(), itervalues(), iterkeys()** – we will get back to it when the for loop will be covered
- **a.update(otherdict)** – inject content of the other dict into the "a"
  - **fromkeys()** – similar but inserts only keys

# OTHER BASIC CONTAINERS

- **namedtuples** – like the (,,,) but with addressing using names
- **set** – like a dictionary but w/o value
- **deque** – "double ended queue" (deck) – like the list but optimised for insertions and removal from the ends
- **defaultdict** – dictionary w/o missing keys
- **OrderedDict** – as you might have guessed

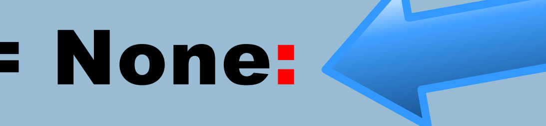
# INTERACTIVE HELP

- How do I learn about all this?
- As in many/all scripting languages "help" system is on the spot
  - **dir(obj)** – gives list of all attributes of the object
    - including methods
  - **help(obj.append)** – gives brief help message on usage of the method **append** of the object

# CONDITIONALS/LOOPS ETC.

- The if statement by example

**if a == None:**



**print ’a is unset’**

**pass # not required, useful when empty block is needed**

**else if a < -1:**

**print ’a is negative’**

**if a < -100:**

**print ’a is extremaly negative’**

**else:**

**print ’a is either positive or 0’**

# THE FOR LOOP

- Again let's start by example, assume the "s" is a list

```
for el in s:
    print el
else: # not required and rarely used, you can just write
    print after the for loop block
    print "finished"
```

- There are also **break** and **continue** keywords

# ITERATING WITH THE FOR LOOP

- Very concise syntax

```
for k,v in d.iteritems():
    print ’key’, k,  ’has value’, v
```

- When the list of integers is needed the “range” built-in helps

```
for i in range(100):
    print i
    # do something else with i
```

# LISTS COMPREHENSION

- Very compressed way of scanning and transforming lists
- Imagine you have list of integers and want to have list of it’s squares

```
a=range(5,25,3)

b = [ x*x for x in a]
```

If you want to filter elements in addition:

```
b = [x*x for x in a if a != 8]
```

# OTHER GLOBAL LISTS OPERATIONS

- **len(a)** – size of the list
- **zip(a, b)** – makes list of tuples of elements from a and b
  - 0th elements from a and b, 1st elements from a and be ….
- **sum(a)** – sums all objects
- **map( lambda x:x<0, a)** – like lists comprehension
- **reduce(function_of_two_args, list, start)** – applies the function to the start and **list[0]**, then to the result and **list[1], …**
  - **reduce(lambda x,y: x+y, a, 0)** – is the same as the sum
  - **reduce(lambda x,y: x*y, a, 1)** – is the factorial
- **filter(function, a)** – leaves only this elements for which the function returns True
- **any(a), all(a)** – True if any or all elements are True
- **sorted(a), reversed(a)** – obvious functionality, the lists are not altered

If you care about the speed, this methods are to be used.

# UFF – WE ARE THROUGH THE BASICS

5 minutes break

# FUNCTIONS

- The indentation as a code block delimiter promotes short procedures → you need to have many of them → functions

```
def rmult(a, b):
    """ multiplies the two arguments with rounding
Note the arguments are first rounded and then multiplied.
    The rounding  is to the nearest max integer"""

    xa = int(a)

    xb = int(b)

    return xa*xb
```

# MORE "UNUSUAL" PROPERTIES OF THE FUNCTION "OBJECT"

- It can be defined anywhere:

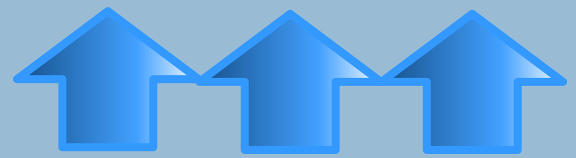
```
def a(arg):
    def smx(arg1, arg2):
        return max(arg1, arg2)
    return [smx(x, x*x) for x in range(0, arg)]
```

- Python function can even be returned

```
def generator(x):
    def gen():
        return x
    return gen
g=generator(5)
[g() for x in range(10)]
```

- If the name of the function is not needed one can define function on the spot with the lambda keyword:

```
map(lambda x: x*x, range(10))
```



# MODULES

- Language look nice but can anything useful be done with it?
  - as the c or c++ or Java, bash, … → the power comes from the external libraries, called modules in python
- How does one start using module
  **import math**
  **math.sin(2)**
- More selective way
  **from math import sin**
  **sin(2)**
  - if there is no risk of name conflicts
    **from math import ***
  - or if the better name is needed
    **from math import sin as sinus**

# IS MODULE COMPLICATED TO WRITE?

- Make a file: myfun.py and place it in the current working dir
- In this file define functions, lists, … objects in general

```
def get_file_content(fname):
    #... do something smart to get the file from somewhere
and then return it's content
```

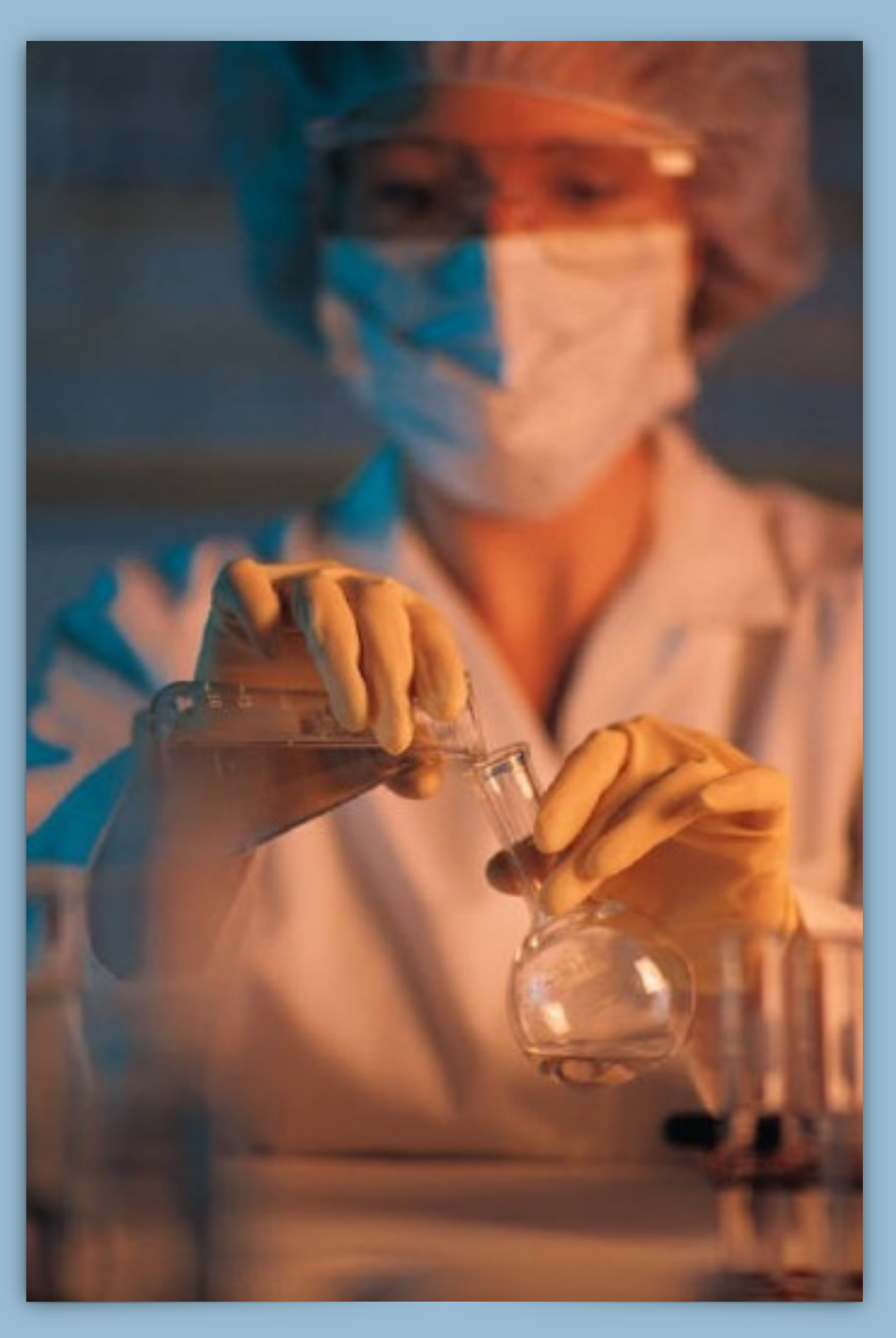
- In the script where you want to use it:

```
from myfun import get_file_content
print get_file_content('dane.txt')
```

  - Conclusion → this is rather trivial!

- If you have location where you want to keep your modules, add this location to the PYTHONPATH shell variable (use export).

# THE MOST IMPORTANT TOPIC OF TODAY!



# TESTING MODULES

- The python code is easy and pleasant to write → you are going to write your modules quickly
- But in the use-case at hand you may not be able to really see if you write correct code
- So you need to test a bit: ***problem*** → change the code → test → change → test → change → …
- Later when you get back to your code, just to modify it a bit so it solves another similar problem → you **may break** what already worked well
- This is bad thing to do to yourself!

You need take care about testing your script.

# THE SIMPLEST APPROACH

- The module “myfun.py” is a file which can be used twofold:

```
import myfun
```

or/and:

```
python myfun.py
```

- In the second case you can invoke self-tests

```
if __name__ == '__main__':
    c = get_file_content('testfile')
    assert(c == 'test content')

    # and so on
```

# WE ARE THROUGH ANOTHER BIG PIECE

- Get the advise to your heart, write self-tests whenever code seems useful enough to be moved to the module
- What is left for today, going through the list of modules
- The modules usually contain objects → so we need to tackle OO in python
- Objects are created similarly to plain types we mentioned

**a = dict()**

**b = list()**

**h = file(name, ”r”)**

- To call object methods type **obj.method()**
  w/o the () you can access the members:

**b.index(7)**

# STRINGS AS EXAMPLE OBJECT

- You already know how to define string literals: 'bla' or "bla"
- Important is to remember that strings are immutable in python → you can never change it's content, (all the code below is useless w/o assignment)

```
a="bla"

a[1]

a[2]="e" #nope, strings are immutable

len(a)

a.center(25); a.rjust(25); a.ljust(25); a.strip(); a.rstrip(); a.lstrip() # adds removes spaces (or other character)

a.endswith('xd'); a.startswith('yes')

a.find('la'); a.index('a'); a.replace('a', 'rrr')  # there are rindex, rfind versions which search from the end of the string

a.split('l'); ' '.join(list) # splits a , joins using space as the delimiter
```

# READING TEXT FILES

- Opening and reading text file can be one-liner

```
list_of_words = []
[ list_of_words.extend(line.split()) for line in file('data.txt', 'r')]
```

- Usually though this is done is less "compressed" code

```
words={}
datafile = file('test', 'r')
for line in datafile:
    for w in line.split():
        clean = w.strip(',.;?!') # remove junk
        words.setdefault(clean, 0) # if the word is not in the dict yet
        words[clean] += 1 # increment word count by one
data.close()

for word,count in words.iteritems(): #nice print
    print word.ljust(25), ':', count
```

# WRITING TO THE TEXT FILES

- The file can be read using several techniques
  content = f.read() # entire file as a string

- There are similar methods, like readline, readlines etc.

- For writing the usual way is to use: write

  **f.write(2)**
  **f.write(some_string)**
  - There is writelines variant

- If you need formatting use this:
  **f.write('%10d %4f %25s' % (lineno, frac, desc) )**

# MODULES
# TOURS DE FORCE

- string – covered
- re – regular expressions – very powerful
- difflib – for diffing objects (files) and nice output (i.e. as HTML)
- datetime – date and time utilities
- pprint – nice printout (specially useful for nested collection types)
- copy – for making deep copies of objects
- math,cmath – math function ($2^{nd}$ one for complex numbers)

# OS INTERFACE MODULES

- os – unix system interface (doc says it it universal)
  - environ – list of env variables
  - processes execution
- sys – python env
  - settings
  - file descriptors, exit()

```
import sys

sys.exit()
```

# INTERNET RELATED MODULES

- urllib2 – reading from the internet resources

```
import urllib2
f = urllib2.urlopen('http://...')
content = f.read()
```

- httplib – more interactive data retrieval
- ftplib – guess
- smtplib, smtpd, pop3 – e-mail stuff

# AND MANY MANY OTHER ….

- scipy – for serious math …
- matplotlib – nice plotting library (probably best quality plots can be made with it)
- Here we finish, next 2/3 labs and then we will have another lecture (possibly short) on gluing python with C/C++ external programs etc.