Python Course

USIT - UiO

2012

Miguel Oliveira

m.a.oliveira@usit.uio.no

17-04-2013

Outline

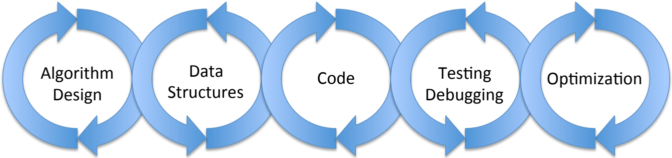
1. Introduction to Programming
2. What is Python?
3. Python as a calculator
4. Introduction to Python Programming

Outline

Outline

Outline

Introduction to Programming



What is Python?

- Python is a VHLL (Very High Level Language).
- Created by Guido van Rossum (Univ. of Amesterdam) in 1991.
- Named inspired in "Monty Python's Flying Circus"...
- On his own words:

  Python is an interpreted, interactive, object-oriented programming language. Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing.

What is Python?

- Python is still in development by a vast team of collaborators headed by GvR.
- Licensed under GPL.
- Available freely for most operating system on:
  http://www.python.org/
- Available currently in two versions: Python 2.x & Python 3.x
- What are the differences:
  - *Python 2.x is the status quo*
  - *Python 3.x is the present and the future of the language*
- Which version should I use:
  - If you are working on a new, fully independent project, you can use 3.x
  - If your work has dependencies it is safer to use 2.x...

Python as a calculator

- Python can be used as a simple calculator:

```
[miguel@MAO ~] > python
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Apple/clang-418.0.60)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2*2
4
>>> 2**3
8
```

Arithmetic expressions

- Common arithmetic expressions can be used with objects of a numerical type.
- Conversion integer→long integer→float functions in the usual way, except in the case of division of integers which is interpreted as integer division.

```
>>> n = 1
>>> z = 2*n
>>> print z
2
>>> x = 1.23456
>>> print 2*3.456+5*x
13.0848
>>> print z/3
0
```

Arithmetic expressions

The available arithmetic operators are:

Operator	Stands for	Precedence
+	sum	0
-	subtration	0
*	multiplication	1
/	division	1
//	integer division	1
%	remainder	1
**	power	2

Arithmetic expressions

- Python evaluates arithmetic expressions according to precedence:

```
>>> print 2**3+2*2
12
```

- Precedence can be overruled using parenthesis:

```
>>> print 2**(3+2*2)
128
```

Mathematical functions

- Python has a limited knowledge of mathematical functions.
- There is however a module, math, that extends that: ceil, floor, fabs, factorial, exp, log, pow, cos, sin, etc...
- For example:

```
>>> import math
>>> print math.sqrt(25)
5
```

or

```
>>> from math import sqrt
>>> print sqrt(25)
5
```

Bitwise operators

- Python also supports bitwise operators:

Operator	Stands for	Precedence
$\vert$	binary OR	0
$\wedge$	binary XOR	1
&	binary AND	2
<<	left shift	3
>>	right shift	3
$\sim$	binary not	4

Bitwise operators

- For example:

```
>>> x=1
>>> x<<2
4
```

- Arithmetic operations take precedence over bitwise operations:

```
>>> x=1
>>> x<<2+1
8
>>> (x<<2)+1
5
```

Complex Numbers

- Python also handles complex numbers
- Imaginary numbers are written with the suffix j or J.
- Complex number have the form (real+imag j) or with complex(real,imag)

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Complex Numbers

- A complex number z has a real part and an imaginary part, which are accessible with z.real and z.imag
- The absolute value of a complex number z can be calculated with abs(z)

```
>>> z = 3.0+4.0j
>>> z.real
3.0
>>> z.imag
4.0
>>> abs(z) # sqrt(z.real**2 + z.imag**2)
5.0
```

Complex Numbers

- It is not possible to convert a complex number to float even if its imaginary part is zero!

```
>>> a = complex(3,0)
>>> float(a)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use e.g. abs(z)
```

Complex Numbers

- Functions from the math module do not work with complex numbers.
- Appropriate mathematical functions for complex numbers are defined in the cmath module.

```
>>> import math,cmath
>>> print math.sqrt(-1)
Traceback (most recent call last):
File "<stdin>", line 1, in ? ValueError: math domain error
>>> print cmath.sqrt(-1)
1j
>>> z = cmath.exp(cmath.pi*1j)
>>> print z.real,z.imag,abs(z)
-1.0 1.22460635382e-16 1.0
```

Python Programming

- Structure of a Python program:
    - Programs are composed of **modules**.
    - Modules contain **statements** and **expressions**.
    - Instructions and expressions create and process **objects**.

What is an object?

- Difficult to say at this stage....
- But simplistically, objects correspond to a certain memory region to which a unique memory address is associated and in which we store:
  - data,
  - information about the data,
  - functions that act upon the data.

What is an object?

- Python's basic interaction is to create and name an object; this is called an (name) attribution.
- For example:

```
>>> x = 123
```

  creates the object 123, somewhere in memory and gives it the name `x`. We can also say that `x` is a reference to the object or even that it "points" to the object.

What is an object?

- After created objects will be referred to by name. For example, the instruction:

  ```
  >>> print x
  ```

  prints to the screen the value of the object whose name is `x`.
- Not all objects have a value, but all objects must have a type and a unique memory address.
- We can easily obtain the type and unique memory address for an object:

  ```
  >>> x = 1.23456
  >>> print x
  1.23456
  >>> type(x)
  <type 'float'>
  >>> id(x)
  135625436
  ```

What is an object?

- It is possible to give more than one name to an object. This is called *aliasing*:

```
>>> x = 45
>>> y = 45
>>> id(x)
135363888
>>> id(y)
135363888
```

What is an object?

- We can even do aliasing in just one go:

```
>>> x = y = 45
>>> id(x)
135363888
>>> id(y)
135363888
```

What is an object?

- However one name can not be used by more that one object.

```
>>> x = 20
>>> x = 43
>>> print x
43
```

- The last attribution statement prevails and the object 20 no longer has a name.
- Python will automatically delete unnamed objects (*garbage collection*).

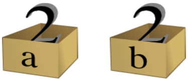
Other languages have variables...

- In many other languages, assigning to a variable puts a value into a box.
- *int* $a = 1; \leftarrow$ Box "a" now contains an integer 1.

- Assigning another value to the same variable replaces the contents of the box:
- *int* $a = 2; \leftarrow$ Now box "a" contains an integer 2.

Other languages have variables...

- Assigning one variable to another makes a copy of the value and puts it in the new box:
- *int* $b = a; \leftarrow$ "b" is a second box, with a copy of integer 2. Box "a" has a separate copy.



Python has objects...

- In Python, a “name” or “identifier” is like a parcel tag (or nametag) attached to an object.
- $a = 1; \leftarrow$ Here, an integer 1 object has a tag labelled “a”.

- If we reassign to “a”, we just move the tag to another object.
- $a = 2; \leftarrow$ Now the name "a" is attached to an integer 2 object.

- The original integer 1 object no longer has a tag “a”. It may live on, but we can’t get to it through the name “a”.
- When an object has no more references or tags, it is removed from memory - garbage collection.

Python has objects...

- If we assign one name to another, we're just attaching another nametag to an existing object.
- $b = a; \leftarrow$ The name "b" is just a second tag bound to the same object as "a".

Identifiers and reserved words

- Giving and object a name is one of the most basic instructions in Python.
- Names cannot begin by a number and cannot contain character with operational meaning (like *,+,−,%).
- Names can have arbitrary length.
- For instance `this-name` is illegal but `this_name` isn't.
- Names cannot coincide with the Python's reserved words:
  `and assert break close continue def del elif else except exec finally for from global if import in is lambda not or pass print raise return try else`
- Names shouldn't redefine common predefined identifiers: `True`, `False` e `None` or intrinsic functions like *float*.

Identifiers and reserved words

- As a rule short and suggestive names are the best option.
- For example:

```
x = 0.46
```

is a bad choice for the lack of clarity, but:

```
inflation = 0.46 # inflation rate in Norway
```

is a good choice (note the small comment after #), and is better than the unnecessarily verbose, although syntactically correct, version:

```
norwegian_inflation_rate = 0.46
```

Types of objects

There are several type of objects:

- *numbers*:
  - integers
  - long integers
  - floats
  - complexes
- *collections*:
  - *sequences*
    - strings
    - tuples
    - lists
  - *maps*
    - dictionaries.
- files, functions, classes, methods
- etc...

Categories of objects

- Each object belongs to one of two categories:
  - *mutable*
  - *immutable*
- **If and object is immutable it cannot be altered. Once created we can only change its name or destroy it.**
- **A mutable object can be altered.**
- **Numbers, strings and tuples are immutable objects.**

Types and categories of objects

- We can destroy and object with the instruction `del`.

```
>>> x = 123
>>> print x
123
>>> del(x)
>>> print x
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'x' is not defined
```

Input

- In Python interaction with the user can be done with the instructions:
    - `input("message")`
    - `raw_input("message")`
- `input` assumes that what is given is a valid Python expression.
- `raw_input` assumes that what is given is data and places it in a string.

Input

Exemplos:

```
>>>name = input("What is your name?")
What is your name? Peter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'Peter' is not defined

>>>name = input("What is your name?")
What is your name? "Peter"
>>>print "Hello, "+name+"!"
Hello, Peter!
```

Input

```
>>>name = raw_input("What is your name?")
What is your name? Peter
>>>print "Hello, "+name+"!"
Hello Peter!
```

Strings

- Strings are immutable sequences of characters
- To create such an object we enumerate all the characters enclosed by quotation marks

```
>>> S = 'Help! Python is killing me!'
```

- An empty string is represented by ''

Strings

- Strings can be indexed and sliced:

```
>>> S = 'Help! Python is killing me!'
>>> S[0]
'H'
>>> S[0:5]
'Help!'
>>> S[5:6]
' '
>>> S[:5]
'Help!'
>>> S[6:]
'Python is killing me!'
>>> S[:]
'Help! Python is killing me!'
```

Strings

- Strings can be concatenated and repeated with the operators + and *:

```
>>> S = 'Help'
>>> S+S
'HelpHelp'
>>> 2*S
'HelpHelp'
```

- Strings can be unpacked:

```
>>> S = 'ab'
>>> x,y=S
>>> print x
a
>>> print y
b
```

- The number of elements on both sides needs to be the same or else an error is generated.

Lists

- Lists are mutable and heterogeneous sequences of objects.
- To create such an object we enumerate all the elements of the list separated by commas enclosed by square brackets:

  `L = ['abc', 123, 'Python']`
- An empty list is represented by `[ ]`.

Lists

- Lists being mutable we can alter, add or remove elements:

```
>>> stuff = ['123',123,1 +3j,'numbers']
>>> stuff
['123', 123, (1+3j), 'numbers']
>>> del stuff[2]
>>> stuff
['123', 123, 'numbers']
>>> stuff[0] = 2*123
>>> stuff
[246, 123, 'numbers']
```

Lists

- To add an element to a list we use the method *append*:

```
>>> clubs = ['Benfica', 'Sporting']
>>> clubs.append('Porto')
>>> clubs
['Benfica', 'Sporting', 'Porto']
```

- A list can contain other lists.
- A matrix can be represented by lists of lists.
- For example a 2×2 matrix can be:

```
>>> M = [[1,2,3],[4,5,6],[7,8,9]]
>>> print M[2][1]
8
```

Lists

- Lists can be indexed and sliced:

```
>>>L=[1,2,3,4]
>>>L[0]
1
>>>L[0:2]
[1,2]
>>>L[1:3]
[2,3]
>>>L[:2]
[1,2]
>>>L[2:]
[3,4]
>>>L[:]
[1,2,3,4]
```

Lists

- Lists can be concatenated and repeated with the operators + and *:

```
>>> L =['a','ab']
>>> L = L + L
>>> L
['a', 'ab', 'a', 'ab']
>>> 2*L
['a', 'ab', 'a', 'ab', 'a', 'ab', 'a', 'ab']
```

- Lists can be unpacked:

```
>>> x,y=['a','ab']
>>> print x
'a'
>>> print y
'ab'
```

- The number of elements on both sides needs to be the same or else an error is generated.

Lists

- Lists can contain references to other objects.

```
>>> x = 123
>>> L = [x,x*x]
>>> L
[123, 15129]
>>> L  = 2*L
>>> L
[123, 15129, 123, 15129]
>>> L = L[2:] + [2*L[0]]
>>> L
[123, 15129, 246]
>>> L.sort()
>>> L
[123, 246, 15129]
```

Tuples

- The major difference between tuples and lists is that the first are immutable objects.
- Tuples are constructed again by enumerating its elements separated by commas but enclosed by parenthesis:

```
>>> t = (1,'a',1j)
>>> type(t)
<type 'tuple'>
>>> print t
(1, 'a', 1j)
>>> type(t[2])
<type 'complex'>
```

```
>>> t = 1,'a',1j
>>> type(t)
<type 'tuple'>
>>> print t
(1, 'a', 1j)
>>> type(t[2])
<type 'complex'>
```

Tuples

- For a one element tuple we have a special notation:

  `('single',)`

  or:

  `'single',`

  but not:

  `('single')`

- Just like lists tuples can be indexed, sliced and concatenated.
- Unpacking a tuple is also easy:

```
>>> x,y,z = ('one','two','three')
>>> print x,y,z
one two three
```

- The number of elements on both sides needs to be the same or else an error is generated.

Tuples

- Tuples can contain other sequences, either other tuples or other lists.
- Tuples are immutable but mutable objects inside them can be changed...

```
>>> t = ('Python',['C','Pascal','Perl'])
>>> id(t)
1078912972
>>> lang = t[1]
>>> lang[2] = 'Python'
>>> print t
('Python', ['C', 'Pascal', 'Python'])
>>> id(t)
1078912972
```

Tuples

- Tuples may seem redundant compared with lists. There are however situations where you may need “immutable lists”.
- Tuples are also important in Python in several other contexts. For instance functions always return multiple values packed in tuples.

Dictionaries

- Dictionaries belong to the map category of Python objects.
- Dictionaries are the most powerful data structure in Python.
- Dictionaries can be indexed by any immutable object and not just by integers like sequences.
- Dictionaries are mutable objects to which we can change, add or delete elements.
- Dictionaries contain key:value pairs.

Dictionaries

```
>>> tel = {'pedro': 4098,'ana': 4139}
>>> tel['guida'] = 4127
>>> tel
{'pedro': 4098,'ana': 4139,'guida': 4127}
>>> tel['guida']
4127
>>> del tel['pedro']
>>> tel['berto'] = 5991
>>> tel
{'berto': 5991,'ana': 4139,'guida': 4127}
>>> tel.keys()
['berto','ana','guida']
>>> tel.has_key('ana')
True
>>> 'ana' in tel
True
```

Dictionaries

- It is possible to obtain a list of all keys and values in a dictionary with the methods `keys()` and `values()`.
- Ordering any of these lists can be done with `sort`.

```
>>> tel = {'berto': 5991,'ana': 4127,'guida': 4098}
>>> tel.keys()
['guida','berto','ana']
>>> tel.values()
[4098, 5991, 4127]
>>> keys = tel.keys()
>>> keys.sort()
>>> keys
['ana','berto','guida']
```

Dictionaries

- Trying to access an element of a dictionary that does not exist generates an error.
- We can always use the method `get(x,dft)` that allows to define a value by default to use if the element does not exist.

```
>>> print d.get('ana',0),d.get('fernando',0)
4127 0
```

- This method can easily be used to implement sparse matrices were most elements are zero...

Control Flow

- The flow of a program (sequence of instructions) is dictated by the *control flow* instructions, that in Python are:
  - `if ... elif ...else`
  - `while...else`
  - `for...else`
  - `break`
  - `continue`
  - `try...except...finally`
  - `raise`

Control Flow

- Compared with other languages Python has a reduced number of control flow instructions.
- Control flow instructions are designed to be powerful and generic.
- For example:

  `for ... else`,

  allows running over any iterable object.

Conditions: `if...elif...else`

```
 if <test>:
      <statements>
[elif <test>:
       <statements>]
[else:
      <statements>]
```

Conditions: if...elif...else

Exemplo:

```
if choice == 'eggs':
    print 'Eggs for lunch.'
elif choice == 'ham'
    print 'Ham for lunch'
elif choice == 'spam'
    print 'Hum, spam for lunch'
else:
    print "Sorry, unavailable choice."
```

True or False

- Interpreting `True` or `False` in Python follows the rules:
  - An object is considered `False`:
    - number zero
    - empty object
    - object `None`
  - An object is considered `True` if it is not `False`:
    - in case of a number, if it is not zero
    - in any other cases if it is not empty
- The logical value of an object can be calculated with the function `bool`. It will return `True` or `False`, the two only possible `bool`(ean) values.

True or False

In Python the comparison operators are:

Operator	Description	Example
`<`	Smaller than	`i < 100`
`<=`	Smaller than or equal to	`i<=100`
`>`	Greater than	`i>100`
`>=`	Greater than or equal to	`i>=100`
`==`	Equal to	`i==100`
`!=`	Not equal to	`i!=100`
`is`	Are the same objects	`x is y`
`is not`	Are different objects	`x is not y`
`in`	Is a member of	`x in y`
`not in`	Is not a member of	`x not in y`

True or False

In Python the boolean operators are:

Operator	Description	Example
`not`	Negation	`not a`
`and`	Logical and	`(i<=100) and (b==True)`
`or`	Logical or	`(i>100) or (b>100.1)`

Loops: `while...else`

```
while <test>:
    <statements>
    [if <test>: break]
    [if <test>: continue]
    <statements>
[else:
    <statements>]
```

- Nota: The clauses `break`, `continue` and `else` are optional. If the `break` test is true execution jumps to the end of the `while`. If the `continue` test is true, execution abandons the current cicle and continue with the next. If the `else` clause is present and `break` is not called the program executes the `else` interactions at the end of the loop.

Loops: `while...else`

- Example 1:

```
a = 0; b = 10
while a < b:
    print a,
    a += 1  # a = a+1

Result: 0 1 2 3 4 5 6 7 8 9
```

- Example 2:

```
x = 10
while x:
    x -= 1    # x = x-1
    if x % 2 == 0:
        print x,

Result: 8 6 4 2 0
```

Loops: `while...else`

- Example 3:

```
name = 'Spam'
while name:
      print name,
      name = name[1:]
```

```
Result:  Spam pam am m
```

Loops: `while...else`

- Example 4:

```
# Guess a number game
mynumber = '123456'
while 1:
      n = input('Guess the number: ')
      if n == mynumber:
         print 'You guessed the number!'
         break;
      else:
         print 'Sorry, wrong guess.'
print 'Game is over'
```

Loops: `while...else`

Result:

```
Guess the number: 43465
Sorry, wrong guess.
Guess the number: 7161527
Sorry, wrong guess.
Guess the number: 999999
Sorry, wrong guess.
Guess the number: 123456
You guessed the number!
Game is over.
```

Loops: for...else

- In Python the instruction `for` is used (exclusively) to iterate over objects. In the current version of Python all sequences have an inbuilt iterator. It is also possible to define iterator for other objects.
- Syntax:

```
 for x in object:
    <statements>
    [if <test>: continue]
    [if <test>: break]
[else:
    <statements>]
```

Loops: for...else

- If the optional clause exists `else`, the enclosed instructions will be executed only after the loop is finished without the `break` condition having been met.
- Examples:

```
basket = ['orange','banana','apple']
for fruit in basket:
    print fruit
```

```
phrase = 'Oslo is a nice town.'
for c in phrase:
    print c,ord(c) # ord(c) = ASCII code
```

Loops: for...else

- Example (testing if there is at least one negative number in a list):

```
for x in L:
    if x < 0:
        print 'There are negative numbers'
        break
else:
    print 'All numbers are non-negative'
```

Range

- Python has the inbuilt function `range` to create integer lists that are often used to iterate `for` loops:

```
range(n)        →  [0,1,2,...,n-1]
range(i,j)      →  [i,i+1,i+2,...,j-1]
range(i,j,k)    →  [i,i+k,i+2k,...]
```

Range

- Examples:
  ```
  range(5) = [0,1,2,3,4]
  range(2,5) = [2,3,4]
  range(1,10,2) = [1,3,5,7,9]
  range(0,-10,-3) = [0,-3,-6,-9]
  ```
- In the case of Python 2.x if the lists generated are very long it might be useful to use instead `xrange`. This function is similar to the previous but is memory friendly only creating list elements as they are needed.

Loops for...else

- To iterate in a classic fashion, equivalent to Pascal, Fortran or C, we can use the construct:

```
>>> for i in range(len(lista)):
>>>     print lista[i]
```

which is totally equivalent to the more pythonic version:

```
>>> for x in lista:
>>>     print x
```

Loops `for...else`

- Warning: The iterating list should never be changed in the `for` loop otherwise wrong results may occur!
- Wrong example:

```
for x in lista:
    if x < 0: lista.remove(x)
```

- The right way to perform such a task is to iterate over a *copy* (*clone*) of the original list:

```
for x in lista[:]:
    if x < 0: lista.remove(x)
```

Loops `for...else`

- Sometimes when iterating over a sequence it is useful to access the element and its index. An option is:

```
for i in range(len(L)):
    print i,L[i]
```

  but it is more practical to use the predefined function `enumerate`:

```
for i,x in enumerate(L):
    print i,x
```

Loops for...else

- To iterate over multiple sequences there is also a very useful function:

  zip

- For example:

```
colors = ("red","green","blue")
clubs = ("Benfica","Sporting","Porto")

for club,color in zip(clubs,colors):
     print club,color
```

Functions: Definition

- Functions in Python are similar to functions in “C”and functions/procedures in “Pascal”.
  Syntax:

```
def nome([arg_1,arg_2,...,arg_n]):
    <statements>
    return [value_1,value_2,...value_n]
```

- Notes:
  - The instruction `def` creates an object of type function and attributes it a name
  - `return` returns the results to the calling instruction
  - To call a function, after it is created, we just need to invoke it by name

Functions: Utility

- Code reutilization
- Decomposition of a complex task into a series of elementary procedures
- Eases readability of the code and future modifications

Functions: Examples

- Example 1:

```
>>> def prod(x,y):
>>>     return x*y
>>>
>>> print prod(2,3)
>>> 6
>>> z = 2
>>> y = prod(9,z)
>>> print y
>>> 18
>>> sentence = 'aa'
>>> z = prod(sentence,2)
>>> print z
>>> 'aaaa'
```

Functions: Examples

- Example 2:

```
>>> def intersect(seq1,seq2):
>>>     res = []
>>>     for x in seq1:
>>>         if x in seq2:
>>>            res.append(x)
>>>     return res
>>>
>>> intersect([1,2,3],[1,7,2])
>>> [1,2]
```

Functions: Documentation

- Functions defined in Python can contain documentation.
  Syntax:

```
def name([arg_1,arg_2,...,arg_n]):
    'Documentation and help.'
    <statements>
    return [value_1,value_2,...value_n]
```

- This text can be obtained with the help command:

```
>>>help(name)
```

Functions: Documentation

- This command can be used even with intrinsic functions:

```
>>>import math
>>>help(math.sqrt)
Help on built-in function sqrt in module math:

sqrt(...)
    sqrt(x)

    Return the square root of x.
```

Functions: parameter

- Arguments in Python are passed by value (object is referenced but is passed by value).
- This means that a local name in the function scope is created.
- More precisely immutable objects cannot be changed inside functions (or outside for that matter...).
- Mutable objects can however be changed inside functions.
- For example:

```
def try_to_change(n):
    n='A'
```

will give:

```
>>>name='B'
>>>try_to_change(name)
>>>name
'B'
```

Functions: parameters

- However:

```
def change(n):
    n[0]='A'
```

will give:

```
>>>name=['B','C']
>>>change(name)
>>>name
['A','C']
```

Functions: parameter types

- Positional parameters
- Keyword parameters
- Keyword parameters and defaults
- Collecting parameters

Functions: Positional Parameters

- Up to now all parameters we have used are positional.
- They are called positional because their order is crucial. For instance:

```
def op(x,y,z):
      return x/y+z
```

op(x,y,z) will certainly give a different result than op(y,x,z).

Functions: Keyword parameters

- Consider a function like:

```
def hello(greeting,name)
    print greeting+", "+name+"!"
```

- Using positional parameters we would in general call it like:

```
>>> hello('Hello','world')
Hello, world!
```

- It may however be useful to explicitly name the variables in the calling instruction:

```
>>> hello(greeting='Hello',name='world')
Hello, world!
```

  We say we are using keyword parameters in this call.

- In this case the order is unimportant:

```
>>> hello(name='world',greeting='Hello')
Hello, world!
```

Functions: Keyword parameters and defaults

- Keyword parameters can also be used in the function definition to specify default values. Example:

```
def add_tax(x,vat=20):
...     """Adds the VAT tax (given as percent)
...        to value x.
...     """
...     return x*(1+vat/100.)
...
print add_tax(100)
120.0
print add_tax(100,5)
105.0
print add_tax(100,iva=7)
107.0
```

Functions: Keyword parameters and defaults

- When omitting the argument `iva` its default value (20%) is assumed.
- Warning: If one argument is of type keyword then all that follow must also be keyword parameters. For instance, the following example is an illegal call:

```
print add_tax(buy=200,5)
SyntaxError: non-keyword arg after keyword arg
```

Functions: Keyword parameters and defaults

- Warning: Default values are only evaluated once at the time a function is defined.

```
taxa = 20
def add_tax(x,iva=taxa):
...     return x*(1+iva/100.)
...
print add_tax(100)
120.0
taxa = 21
print add_tax(100)
120.0
```

Functions: Keyword parameters and defaults

- Try to explain the unexpected result:

```
def save(x,L=[]):
...     L.append(x)
...     print L
save(1)
[1]
save(2)
[1, 2]
L =['a','b']
save(3,L)
['a', 'b', 3]
save(3)
[1, 2, 3]
```

Functions: Collecting parameters

- It is sometimes useful to write functions whose number of parameters is arbitrary. To this effect we can use the symbols `*args` e `**keyw` as function parameters.
- (`*args`) stands for an arbitrary number of positional parameters and is a tuple.
- (`**keyw`) stands for an arbitrary number of keyword parameters and is a dictionary.

Functions: Collecting parameters

```
def union(*args):
   res = []
   for seq in args:
       for x in seq:
           if x not in res:
                res.append(x)
   return res

a = [1,2,3]; b =[2,7,8]; c =[0,3]
print union(a,b,c)
[1, 2, 3, 7, 8, 0]
```

Name resolution: scoping rule

- Name resolution inside a function follows the rule: first the *local* namespace of the function is searched, i.e., all the names attributed to objects or those attributed by using the instructions `def` or `class`. If the name cannot be found it is searched in the *global* namespace, i.e., the names defined in the main module. If it still not found there then the *built-in* predefined namespace is searched. In the case the name cannot be found there then exception is raised "NameError: name is not defined."
- This simple rule is known as the scoping rule: **LGB (Local, Global, Built-in)**.

Function: Returning results

- The instruction “return” can actually return any type of object that Python recognises and in whatever number.
- As a special case a function might not return anything. In this case the “return” instruction can even be omitted.

```
def test():
    print "This is a test"
    return
```

- Note: In `Python` all functions return at least one value. If it is not defined or omitted then they return `None`.

Function: Returning results

- When return more than one value these are enumerate in the "return" instruction separated by commas and packed in a tuple.

```
def multiples(x)
   return x,2*x,3*x,4*x

multiples(3)
(3,6,9,12)
```

- To unpack these we use the standard technique:

```
x,y,z,t = multiples(3)
print x,y,z,t
3 6 9 12
```

Function: Returning results

- Returning multiple values is extremely useful. Consider a function to exchange two variables. Consider the “canonical” procedure:

```
def swap(x,y)
   temp = x
   x = y
   y = temp

x = 2; y = 3
swap(x,y)
print x,y
2 3
```

- Can you explain why swap does not work?

Function: Returning results

- Don't worry Python has a way of exchanging values:

```
def swap(x,y)
   return y,x

x = 2; y = 3
x,y = swap(x,y)
print x,y
3 2
```

Function: Returning results

- In reality Python does not need to define a function to exchange values. It can simply be done as:

```
x = 2; y = 3
print x,y
2 3
x,y = y,x
print x,y
3 2
```

Functions: Recursion

- In Python, like in many other modern programming languages, it is possible to define a function in a recursive way, i.e., a function is allowed to call itself.
- For example the following function determines $n!$ in a recursive fashion.

```
def fact(n):
...     if n == 1:
...         return 1
...     else:
...         return n*fact(n-1)
...
fact(3)
6
```

Functions: Recursion

- The sequence of steps that takes place is:

```
fact(3) -> 3*fact(2) (stack memory)
fact(2) -> 2*fact(1) (stack memory)
fact(1) -> 1
retraces:
fact(2) <- 2*1
fact(3) <- 3*2*1 = 6
```

Functions: Recursion

- Defining a function recursively is most of the times clear but its execution and its memory usage may be too costly.
- Use at your own peril!!!

Files

- Files allow storage of objects in a permanent way.
- Python has a vast set of instructions to manipulate files.
- A file object is created with the instruction `open`:

```
f = open(filename,mode)
```

  where:
  - `filename` is a string containing the nome, and eventually the path, of the file.
  - `mode` is a string representing the mode in which the file is used.

  Frequent modes are:

`'r'`	-	Opening for reading.
`'w'`	-	Opening for writing.
`'a'`	-	Opening for apending.
`'r+'`	-	Opening for reading and writing.

- For example, to open the file `'report.txt'` in writing mode:

  f = open('report.txt','w')

Files

- After all operations on a file are concluded it should always be closed with the method `close`.

```
f = open('report.txt','r')
...
f.close()
```

- Closing a file allows Python to free all system resources related to that file, specially memory.

Files

- A file is an iterable object where the iterator returns, in sequence, every line of the file.
- A file can thus be printed to screen as:

```
f = open('report.txt','r')
for line in f:
    print line,
f.close()
```

Files

- Files have several predefined methods.
- It's possible to list all the methods associated with a given object `f` with `dir(f)`.
- The most important are:
  - `read`
  - `readline`
  - `readlines`
  - `write`
  - `writelines`.

Files

- `f.read(n)` reads `n` characters of the file and returns them in a *string*.

  If `n` is omitted all the file is read.

  If the end of the file is encountered while the `n` characters are being read the number of characters returned is less than `n`.

- Another way to print the file is then:

```
f = open('report.txt','r')
print f.read(),
f.close()
```

Files

- `f.readline()` reads one line of the file and returns it in a *string*.
- Yet another form to print a file would be:

```
f=open('exp.txt','r')
while True:
    line=f.readline()
    if line != '':
        print line,
    else:
        break
f.close()
```

Files

- `f.readlines(n)` reads `n` lines of the file and returns then in a list of *strings* (one for each line).
- To print a file we can then use:

```
f=open('report.txt','r')
lines = f.readlines()
for line in lines:
     print line,
f.close()
```

Files

- The methods `write` and `writelines` allow to write one or several lines to a file. The argument for the first one is a *string* constaining the line to write, while in the second case it's a list of *strings*.

```
f.write('Hello world!\n')
messages = ['How are you?','And your wife?']
f.writelines(messages)
```

Files

- Note that only *strings* can be stored in files. Any other kind of object has to be converted to *strings* before being stored to file.
- Simple objects can be converted with the function `str`, the quotation marks (`''`), that are interpreted as conversion to `string`, or using the formatting operator `(%)`.

```
a = str(123.2)
b = '1234'
f = 123.2; n = 189
s = "%10.2f %d %s %s" % (f,n,a,b)
```

Files

- More complicated objected may be *serialized* automatically using the `pickle` module:
  - `pickle.dump(x, f)`
    writes the x object into the f file open for writing.
  - `x = pickle.load(f)`
    reads the x object from the f file open for reading.

Output formatting

- Conversion of objects to *strings*, conveniently formated, is made easier by using the formatting operator %. A formatted *string* is produced with the expression:

  ```
  format-string % (o1,o2,o3...)
  ```

- Exemplo:

  ```
  >>> x = "This is %5.3f with %d decimals" % (math.pi,3)
  >>> print x
  This is 3.142 with 3 decimals
  ```

Output formatting

- The *format-string* is a *string* with the following generic form:
    - Character '%' marks the beginning of the formatting.
    - A mapping key (optional) consists in a sequence of characters inside parenthesis.
    - A conversion flag (optional) describes the data.
    - Minimal size of the field (opcional).
    - Precision (opcional), defined after a '.'.
    - Conversion type.

Output formatting

- The conversion flag can be of the type:

`'0'`	For numeric values the field will be padded with zeros.
`'-'`	The converted value is left aligned.
`' '`	A space is left empty before any positive value.
`'+'`	Sign ('+' or '−') will precede the converted value.

Output formatting

- A *format-string* is a *string* that contains formatting codes:

`%s`	*string*
`%c`	character
`%d`	integer - decimal
`%o`	integer - octal
`%x`	integer - hexadecimal
`%f`	float
`%e`	float in scientific notation
`%g`	float in alternate format

Modules

- Python code can be divided into functional units that can be stored in different files.
- These units are called modules.
- Each module is just a regular Python code.
- It can be as simple as:

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

- You can now tell Python to interpret this module with `import`:

```
>>> import hello
Hello world!
```

Modules in the right place

- Where does Python look for modules?

```
>>> import sys
>>> print sys.path
['', '/System/Library/...]
```

Modules in the right place

- If that is not too easy to read:

```
>>> import sys, pprint
>>> pprint.pprint(sys.path)
['',
 '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python27.zip',
 '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7',
 '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-darwin',
 '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac',
 '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac/lib-scriptpackages',
 '/System/Library/Frameworks/Python.framework/Versions/2.7/Extras/lib/python',
 '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib-tk',
 '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib-old',
 '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib-dynload',
 '/System/Library/Frameworks/Python.framework/Versions/2.7/Extras/lib/python/PyObjC',
 '/Library/Python/2.7/site-packages']
```

Modules in the right place

- What if I want to put my modules in another directory?
  - Solution 1:

    ```
    >>> import sys
    >>> sys.path.append('my_python_module_dir')
    ```
  - Solution 2:
    - At the operating system level redefine the variable PYTHONPATH to include your directory
    - For bash in UNIX system:
      export PYTHONPATH=$PYTHONPATH:my_python_module_dir

Modules

- Modules are mainly used to define functions and classes that can be reused.
- It is useful however to include some test code on every module that is only run if they are called (run) directly (not imported).

```
#hello.py
def hello():
        print "Hello world!"
def test():
        hello()
if __name__ == '__main__': test()
```

Errors and Exceptions

- To represent exceptional conditions, Python uses exception objects.
- If these exception objects are not handled the program terminates with an error message:

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Errors and Exceptions: Raising your own

- When your code encounters an error situation for which it cannot recover you can raise an exception:

```
>>> raise Exception
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception
```

- This raises a generic exception

Errors and Exceptions: Raising your own

- Or in a more documented way:

```
>>> raise Exception, 'Too many arguments...'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: Too many arguments...

>>> raise Exception('Too many arguments...')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: Too many arguments...
```

- These just add an error message
- They are equivalent

Errors and Exceptions

- There are many built-in exceptions that you can raise:

```
>>> import exceptions
>>> dir(exceptions)
['ArithmeticError', 'AssertionError',...]
```

- You can raise any of them:

```
>>> raise ArithmeticError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArithmeticError
```

Errors and Exceptions

- The advantage of having exceptions is that we can handle them
- In programming language this is called trapping or catching an exception
- To catch an exception and perform some error handling we use `try...except`

```
>>> try:
...     x=input('Enter the first number: ')
...     y=input('Enter the second number: ')
...     print x/y
... except ZeroDivisionError:
...     print "The second number can't be zero!"
...
Enter the first number: 1
Enter the second number: 0
The second number can't be zero!
```

Errors and Exceptions

- We can catch as many exceptions as we need:

```
>>> try:
...     x=input('Enter the first number: ')
...     y=input('Enter the second number: ')
...     print x/y
... except ZeroDivisionError:
...     print "The second number can't be zero!"
... except NameError:
...     print "That wasn't a number, was it?!"
...
Enter the first number: a
That wasn't a number, was it?!
```

Errors and Exceptions

- Or in one block:

```
>>> try:
...     x=input('Enter the first number: ')
...     y=input('Enter the second number: ')
...     print x/y
... except (ZeroDivisionError,NameError):
...     print "Something is fishy here!"
```

Errors and Exceptions

- The exception object can also be caught
- Useful if you want to check it or if you want to print the error but still want to go on...

```
>>> try:
...     x=input('Enter the first number: ')
...     y=input('Enter the second number: ')
...     print x/y
... except (ZeroDivisionError,NameError), e:
...     print e
```

Errors and Exceptions

- We can catch all possible exceptions with:

```
>>> try:
...     x=input('Enter the first number: ')
...     y=input('Enter the second number: ')
...     print x/y
... except:
...     print "Something fishy here!"
```

- This is however risky. It might catch a situation we hadn't thought and it will also catch Crtl-C attempts to terminate.

Errors and Exceptions

- It is a safer option to do:

```
>>> try:
...     x=input('Enter the first number: ')
...     y=input('Enter the second number: ')
...     print x/y
... except Exception,e:
...     print e
```

- We can now so some checking on the exception object e.
- You can also consider `try...finally`...

Errors and Exceptions

- Sometimes it is useful to have some code for when the exception is not raised:

```
>>> try:
...     x=input('Enter the first number: ')
...     y=input('Enter the second number: ')
...     print x/y
... except Exception,e:
...     print e
... else:
...     print "Looks like it went smoothly!"
```

Errors and Exceptions

- If you need to do something after a possible exception regardless of what that exception is you should consider:

```
>>> x=None
>>> try:
...     x=1/0
... finally:
...     print "Cleaning up..."
...     del(x)
```

Classes: Definition

- All data types in Python are objects.
- It's the fact that objects know what they are and which functions act on them that makes Python so robust and so simple.
- It's possible to add classes of objects using the instruction `class`.
- For example to create a class of vector objects:

```
>>> class Vec:
...     def __init__(self,x,y,z):
...       self.x=x
...       self.y=y
...       self.z=z
...     def norm(self):
...       return (self.x**2+self.y**2+self.z**2)**0.5
```

Classes: Instances

- The instruction `class` only defines a class. It does not create any object.
- To create an object we just invoke the class as if it were a function. For example for the `Vec` class:

  ```
  >>> u=Vec(1,1,1)
  ```

- In programming terminology we say we created an instance of the class.

Classes: Instances

- In this example, `u` is now an instance of the class `Vec`:

```
>>> type(u)
<type 'instance'>
>>> type(Vec)
<type 'classobj'>
```

- Classes are thus “object factories”.

Classes: Methods

- Functions defined inside a class are named methods of the class.
- Methods are invoked by qualifying the object:

  `object.method()`

- For example:

```
>>> u.x
1
>>> u.norm()
1.7320508075688772
```

Classes: Special Methods - `__init__`

- `__init__` is a special method used to initialise instances of the class.
- In this example it gives values to the `x`, `y` and `z` atributes of the object.
- This initialisation can be done during creation:

```
>>> u=Vec(1,1,1)
```

- Or at any other time:

```
>>> u.__init__(1,1,1)
```

- The first argument `self` represents the object own instance and is always omitted when a method is invoked.

```
>>> u.__init__(1,1,1,1)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: __init__() takes exactly 4 arguments (5 given)
```

Classes: Special Methods - `__repr__`

- One of the most useful methods that classes can define is `__repr__`.
- This method defines how the object is represented.

```
>>> class Vec:
...     def __init__(self,x,y,z):
...             self.x=x
...             self.y=y
...             self.z=z
...
>>> U=Vec(1,2,3)
>>> U
<__main__.Vec instance at 0x249b98>
```

Classes: Special Methods - __repr__

```
>>> class Vec:
...     def __init__(self,x,y,z):
...             self.x=x
...             self.y=y
...             self.z=z
...     def __repr__(self):
...             return "(%f,%f,%f)"%(self.x,self.y,self.z)
...
>>> U=Vec(1,2,3)
>>> U
(1.000000,2.000000,3.000000)
```

Classes: Special Methods - __doc__

- The method __doc__ allows to obtain the class documentation.

```
>>> class Vec:
        """Class for 3D vectors"""
...     def __init__(self,x,y,z):
...             self.x=x
...             self.y=y
...             self.z=z
...
```

```
>>>Vec.__doc__
'Class for 3D vectors'
>>> help(Vec)
...
```

```
>>> U=Vec(1,2,3)
>>> U.__doc__
Class for 3D vectors
>>>help(U)
...
```

Classes: Attributions

- It's always possible to add more names to a class. We just need to make an attribution:

```
>>> dir(u)
['__doc__', '__init__', '__module__', 'x', 'y', 'z']
>>> u.new_attribute=100
>>> dir(u)
['__doc__', '__init__', '__module__', 'new_attribute', 'x', 'y', 'z']
>>> u.new_attribute
100
```

Classes: Operator overloading

- There is a number of special names for methods that allow operator overloading. For example, if we define a method using the special name `__or__` this method can be invoked using the operator |:

```
>>> class Vec:
...   def prod(self,other):
...     return Vec(self.y*other.z-self.z*other.y,\
...                self.z*other.x-self.x*other.z,\
...                self.x*other.y-self.y*other.x)
...   def __or__(self,other):
...     return self.prod(other)
...
```

Classes: Operator overloading

```
>>> u=Vec(1,0,0)
>>> v=Vec(0,1,0)
>>> a=u.prod(v)
>>> print a.x, a.y, a.z
0 0 1
>>> b=u|v
>>> print b.x, b.y, b.z
0 0 1
```

Classes: Operator overloading

- Some of the special names for operator overload are:
  `__init__`, `__del__`, `__add__`, `__or__`,
  `__repr__`, `__len__`, `__cmp__`, etc.

  http://www.python.org/doc/2.4.4/ref/customization.html

  http://www.python.org/doc/2.4.4/ref/numeric-types.html

Classes: Subclasses

- We can define subclasses of pre-existent classes:

```
>>> class UnitVec(Vec):
...     def __init__(self,x,y,z):
...         norm=(x**2+y**2+z**2)**0.5
...         self.x=x/norm
...         self.y=y/norm
...         self.z=z/norm
```

Classes: Subclasses & Inheritance

- Subclasses inherite all the attribute of their parent class.

```
>>> v=UnitVec(3,1,2)
>>> dir(v)
['__doc__', '__init__', '__module__', 'norm', 'x', 'y', 'z']
>>> v.x
0.80178372573727319
>>> v.norm()
1.0
```

Classes: Summary

- Classes allow to incorporate in a single structure all the attributes of an object and the functions that are capable of acting over them.
- Classes allow to overload operators.
- Inheritance is mechanism that allows to define specialised subclasses that inherit all the attributes of their parent class.
- Subclasses are allowed to have new methods or different versions of a parent's method.

Bibliography and credits

- The material in this presentation is based on the lectures notes for the course "Computers & Programming" lectured at the Department of Physics of the University of Coimbra, Portugal, which grew out of on an original presentation by Fernando Nogueira, José António Paixão e António José Silva.
- Beginning Python: From novice to Professional by Magnus Lie Heltland
- Code like a Pythonista: Idiomatic Python