Python Tutorial

by OSOE Project.

▼ Details

Python Tutorial I

This guide will teach you:

- Python syntax & data types
- Python classes & modules
- Python iterators & functional programming

```
#!/usr/bin/python

print "Hello World
```

hello.py

▼ Details

The First Python Program: hello.py

```
#!/usr/bin/python

print "Hello World
```

hello.py

```
>chmod +x hello.py
>./hello.py
Hello World
```

▼ Details

Writing a script in Python to output “Hello World” on the screen can be as simple as 1,2,3. When, writing a Python program, it is useful to write it in a text editor and then save it with a .py extension. In python, there are three differents types of files ; the files with .py extension are the Python script files, the files ending .pyc represent the compiled files and the ones with .pyo extension represent the optimized byte code files. Usually, most people produce the .py files and the system handles the rest.

#!/usr/bin/python invocation tells the Operating System that the Python intepreter is needed to execute this file.

It is also important to change the executions rights by using the commands chmod +x hello.py. This command will now make the hello.py an executable file. Afterwards, ./hello.py automatically invokes the python interpreter which reads and interpretes the hello.py file. Unlike other languages, it is possible to write a simple python program with any variables or function definitions just type in the keyword *`print`* followed by the expression you want to print and then run your program.

Syntax

```
#!/usr/bin/python

statement1
statement2
...
...
```

▼ Details

The syntax of a python code is pretty straightforward. Generally, the syntax consists of #!/usr/bin/python invocation followed by a list of statements. These statements can be of different kinds: variables declarations, function declarations, control flow statements, loops, etc.

In Python, whenever a file is read, all the statements are executed contrary to other languages like C or Java where they only compiled. There are no variable declarations because the Python intepreter figures out what datatype the variable is. Function definitions are also a kind of statement in python. The way function are defined in python can be understood as the definition of a variable which name is the name of the function and which can be invoked by opening and closing parenthesis. It is even possible in python to assign to a variable a value which is a function.

Based Types

```
k = 1
print type(k)
k = 'a string'
print k
print type(k)
k = []
print type(k)
k=u'string as unicode'
print type(k)
```

```
1
<type 'int'>
a string
<type 'str'>
[]
<type 'list'>
<type 'unicode'>
```

```
K = ()
print k
print type(k)
k = {}
print k
print type(k)
k = 0.
print k
print type(k)
```

```
()
<type 'tuple'>
a string
<type 'dict'>
0.0
<type 'float'>
```

▼ Details

In Python, variables do not need to be declared. The first time they are used is when they are declared. Python use dynamics typing, variables can change their type according to last assigned value,

Here you can find differents Based Type in Python

Strongly Type Language

```
a = 9
b =  "9"
c = str(a) + b
d = a + int(b)
```

```
>>> print c
99
>>> print d
18
```

▼ Details

In Python, variables can not be coerced to unrelated types, an explicit conversion is required.

Strings

```
fname=  "mame"
```

```
>>> fname
mame
```

```
fname= fname +  "sall"
```

```
>>> fname
mamesall
>>> fname[2]
m
>>> len(fname)
7
```

▼ Details

One of Python built-in datatypes are strings.Python strings can be enclosed in either single quotes or double quotes. String supports many operations like + to concanate a value onto a string, * to repeat the string or slicing. You can also do indexing in a string. The first character of a string has index 0.

Integer, Float

```
>>> a = 11
>>> a
11
>>> a / 5 # Division with integer
2
>>> a % 5 # Modulo
1
>>> a+=1
12
>>> a = float(12)
>>> a
12.0
>>> a / 5
2.3999999999999999
```

▼ Details

Mathematical operation return same type as input

Lists

```
li= [ “baby” ,  “mom” ,  “dad” ]
```

```
>>> li
['baby', 'mom', 'dad']
>>> li[1]
'mom'
>>> li[0:1]
['baby', 'mom']
```

▼ Details

Lists are also another built-in datatype in Python. They are similar to arrays in other languages like Perl or Basic. They consist of a set of elements enclosed in brackets. Those elements can be of any datatype, Python keeps track of the datatype internally. You access each element like you would do in an array. The indexing in a list always starts at 0. In the example above, you can get a subset of the list, by slicing it. Slicing a list consists of accessing it by giving out two indices. The first index will be the starting point and it will take all the elements up to the second index.

List Operations

```
li.append( “mame” )
```

```
li.remove( “mame” )
```

```
>>> li
['baby', 'mom', 'dad', “ma
```

```
>>> li
['baby', 'mom', “dog” , “c
```

```
li.extend([ “dog” , “cat” ])
```

```
>>> li.index( “dog” )
2
```

```
>>> li
['baby', 'mom', 'dad', “mame” , “dog
```

```
>>> li + [ “newbaby” ]
['baby', 'mom', “dog” , “cat” ,
```

▼ Details

Many operations can be executed with lists. In fact, you can modify a list by adding elements into it, removing elements from the list, or simply searching through the list. The examples above demonstrate those operations. To add elements into a list, you can either use ***`append()`*** or ***`extend()`***. ***`append`*** only adds one single element to the end of a list. That element can be of any datatype. ***`extend()`*** extends the list with all the elements of another list which is provided as an argument. To remove an element from a list, you can use ***`remove()`***. But, ***`remove()`*** only removes the first occurrence of the value. You can also search a list, by using ***`index()`*** to determine the position of the first matching element into the list.

List Comprehension

```
>>> [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [str(i) for i in range(10)]
['0', '1', '2', '3', '4', '5', '6', '7', '8', '
>>> [ i for i in range(10) if i%2==0]
[0, 2, 4, 6, 8]
```

▼ Details

List comprehensions provide a more concise way to create lists in situations where map() and filter() and/or nested loops would currently be used.

Tuples

A tuple is an immutable list

```
tu = ( "baby" , "mom" , "dad" )
```

```
>>> tu
('baby', 'mom', 'dad')
>>> tu[0]
'baby'
>>> tu = tu + ["newbaby"]
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
TypeError: can only concatenate tuple (not "list") to
>>> " baby" in tu
True
```

▼ Details

A tuple is defined in the same way a list is defined. The only difference is the elements are in parenthesis and a tuple is immutable. Once, you create it, you cannot change it. You can't add or remove elements into it, you can't search through them. However, you can use the operator *in* to see if an element is in a tuple. This operator will return True if the element is in the tuple, False otherwise.

Because, they are more efficient than lists, tuples are often used to define a constant set values. The fact that tuple are immutable is also used to store persistent sequence values and make sure that some badly written program does not try to modify the sequence.

Dictionaries

```
Dict = {'user': 'mame',
        'computer': 'foo'}
```

```
>>> dict
{'user': 'mame', 'computer': 'foo'}
>>> dict['user']
'mame'
```

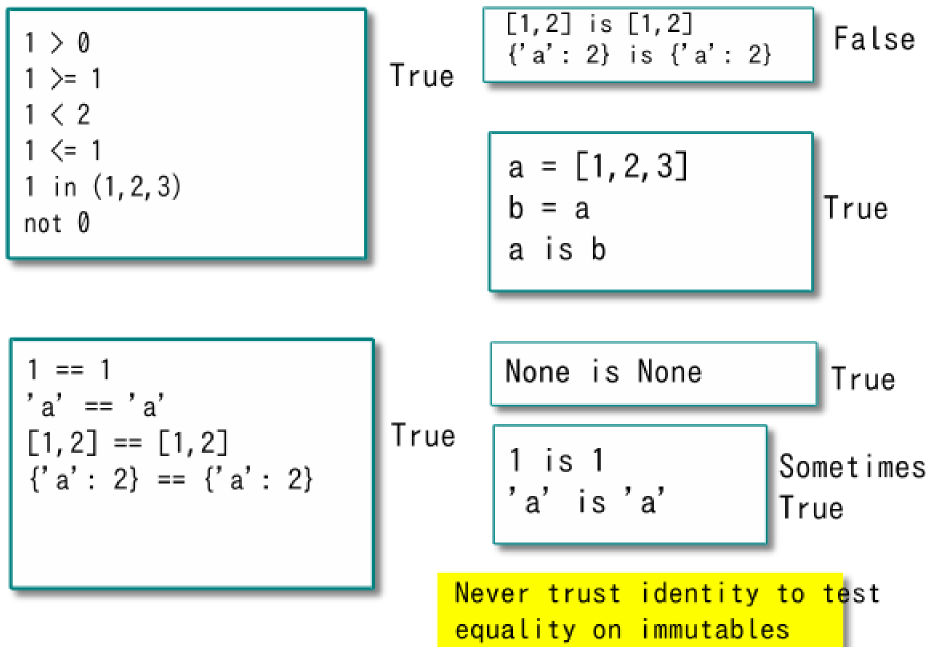
```
dict['computer'] = 'master'
```

```
>>> dict
{'user': 'mame', 'computer': 'master'}
>>> dict.update({'user': 'Toto', 'ip':'192.168.1.1
>>> dict
{'ip': '192.168.1.1', 'computer': 'master', 'user'
```

▼ Details

Another built-in datatype is the dictionary. A dictionary consists of a list of keys associated with some values. In the example above, the dictionary dict has two elements: "user" with its associated value "mame" and "computer" with its associated value "foo". You can access the values by doing dict["user"] or dict["computer"]. However, you can not reference the keys by doing dict["mame"] or dict["foo"]. You can assign a new value to a key by doing for example dict["computer"] = "master". The old value associated with the key "computer" is erased and the new value is set.

Identity vs. Equality



▼ Details

Conditions operators like >,<,<=, >=, in, not, etc. are used to produce boolean expressions which are useful for example for testing in *if* statements. In the first example, we use those operators to compare two integers. Since the values are compared, so if the expression is True, the answer is always True. Equality is represented by == and is used to test whether two values are the same. In the example above, since the values are the same, the answer is true.

However, there is another testing operator called *is* which is the identity testing. This operator tests whether two objects are the same, not their values. In analogy, to another language like C, this operator would test if two objects have the same pointers. In the example, [1,2] and [1,2] have the same values, but are not the same objects: they are two different list objects which are created independently. So, the test fails and returns False. However, a=[1,2,3] and b=a is always True because both variables point to the same object which is the list [1,2,3] which was created only once.

In the last example, 1 is 1 is sometimes true because there is no guarantee in Python that two integers or two strings are represented by the same object. In Python, immutable objects like integers or strings are not guaranteed to be unique unlike in some other object languages (ex. some Smalltalk implementations).

If Statements

```
for i in range(0,6):
  if i< 3:
    print i
  elif i > 4:
    print 3*i
  else:
    print 2*i
```

```
0
1
2
6
8
15
```

▼ Details

There are three types of if statements in Python: ***if*** statement which is used to test whether the expression is true, the ***elif*** statement used to test with another condition and the ***else*** statement which is called when the first two conditions fail. The if statements are a type of of group of statements. If the expression is true, the statements in the indented block are executed, otherwise it goes to the other blocks. The expression doesn't need to be in parenthesis. The ***if***, ***elif*** or ***else*** blocks can contain multiples lines as long as the indentation is consistent. In this example, the for loop with the ***range*** function assign the values 0,1,2,3,4,5 to i. Then, the first ***if*** statement tests that for all values of i less than 3 (i.e. 0,1,2), just print them to the screen, else for all values of i greater than 4 (i.e 5), print 3i to the screen and for the other values (i.e. 3, 4), print 2i to the screen.

Indentation

Always use spaces for indentation, not tabs

```
def printValueList():
    for i in range(0,7):
        if i < 5:
            j = 2 * i
            print j
```

```
def printValueList():
    for i in range(0,7):
        if i < 5:
            j = 2 * i
        print j
```

```
>>> printValueList()
0
2
4
6
8
```

```
>>> printValueList()
0
2
4
6
8
8
8
```

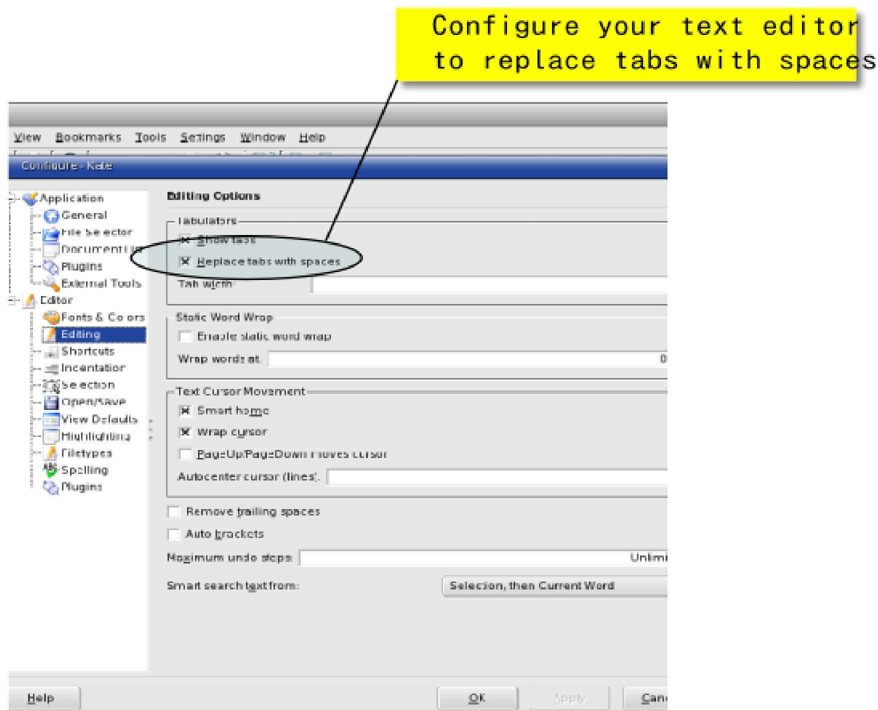
▼ Details

Indentation is very important in Python. Python groups statements with indentation. It replaces the use of brackets or keywords like ***begin*** or ***end*** in other languages. All groups of statements like functions, if statements, for loops, while loops, etc..., are defined by their indentation. Indenting starts a group of statements and un-indenting it ends it. It should be consistent throughout the whole code. A group of statements having the same number of spaces belong to the same

logical block.

In the example above, the two functions are the same except for the last line where the indentation is different. On the left side, the statement ***print j*** is executed only if the ***if*** condition is True while on the right side since the ***print j*** statement has the same indentation as the ***if*** statement, it will be executed for all the values defined by the loop.

Indentation: Tabs



▼ Details

It is important to never use tabs for indentation, always use spaces. In the text editors used to write the Python code, 1 tab can represent 4 spaces, while in others, it can represent 8 spaces. So, using tabs with different text editors can mess up your code. It is recommended to select the feature “replace tabs with spaces” present in most text editors and to use only spaces for indentation. In Python, the standard indentation which is recommended is four spaces. In ERP5, which is written in python, the recommended indentation is two spaces and thus does not follow the pure python recommendations.

For loops

```
for i in range(1,10):
  if i == 2:
    continue
  if i == 5:
    break
  print i
```

```
1
3
4
```

▼ Details

Usually, ***for*** loops are used to iterate the items of a sequence in order. In the example, a sequence of numbers of 1 through 9 are generated by a *range* function and assigned to i. First, the integer 1 is printed. Then, the first ***if*** statement tests that if the value of i is equal to 2, just continue by going to the next iteration of the loop and prints 3 and 4. The other if statement tests if the value of i is equal to 5, break out of the loop and the program ends.

Functions

```
def getValue1(i):
  return i
```

```
def getValue2(i, foo='default'):
  return i, foo
```

```
def getValue3(i, foo='default', **kw):
  return kw.get('bar', 'default2')
```

```
def getValue4(i, *args, **kw):
  return len(args), kw.keys()
```

```
>>> getValue1(2)
2
>>> getValue2(2)
(2, 'default')
>>> getValue2(2, foo='zozo')
(2, 'zozo')
>>> getValue3(2)
'default2'
>>> getValue3(2, bar=1)
1
>>> getValue4(2, 3, 4, bar=1
(2, ['bar'])
>>> a = getValue4
>>> a(2, 3, 4, bar=1)
(2, ['bar'])
```

▼ Details

Contrary to the syntax of functions in other languages like C++, Python functions don't need separate headers. When you need a function, just start it with the keyword ***def*** followed by the function name and the arguments in parenthesis. You can have multiple arguments, but they have to be separated by commas. Some arguments followed by an = assignment are called **named** parameters.

It is important to notice that functions in Python don't need to specify a return datatype or a datatype for their arguments; python functions can also return multiple values, of different type.

In the example above, the first function getValue1 simply returns the value of function parameter. In the second example, getValue2 has two parameters: i and another **named** parameter foo which is initialized to 'default'. When that function is called, if there is no second parameter, it will return the initial value 'default'. However, if the second parameter is set to another value, the function will now return the new value for that second parameter. In the third example, getValue3 the function will return the associated value with the key 'bar' and if there is no parameter it will return 'default2'. The last function getValue4 return, the number of parameters when the function is called and the keys of the names parameters stored in the **kw dictionary. The last example shows that it is possible to assign a function to a variable and later use the variable as if it were a function.

File Management

```
myfile= open( "input.txt" ,  "r" )
myfile.readlines()
myfile.close()
```

```
myfile= open( "output.txt" ,  "w"
for i in range(0,5):
 myfile.write( "good" )
myfile.close()
```

```
#cat output.txt
good
good
good
good
good
#
```

▼ Details

File Management allows you to read elements from a file and write them onto a file. To start manipulating a file, it is always important to first open it by using open(filename,mode). filename is a string representing the name of the file while the mode is either 'r' for reading a file or 'w' for writing in file. In the example above, myfile is created . To read the all the lines in the file myfile.readlines() has been used. readlines() reads all the lines of a file and puts them into a list. read(datasize) and readline() are also other methods to read the contents of a file. read(datasize) looks into the file and reads the quantity of lines specified by datasize. If no datasize is specified, it will read the entire file. readline() reads the file line by line.

The same syntax is for writing in a file. First, you need to open the file for writing, then use myfile.write(string). This will write the contents of the string on the output file. However, write() only writes a string, if something other than a string needs to be written, it needs to be converted to a string first. It is important to always close the file using close(), because it frees up the system ressources used by the file opened.

Classes & Methods

```
class Account:

   def getCurrentBalance(self, balance
      return balance + 1
```

```
>>> x= Account()
>>> x.getCurrentBalance(1000)
1001
```

```
class A:

    def public(self):
        pass

    def _semipublic(self):
        self.__private()

    def __private(self):
        pass
```

```
>>> a = A()
>>> A.public
<unbound method A.public>
>>> a.public
<bound method A.public of ....

>>> A.public(a)
>>> a.public()      Same meaning

>>> a._semipublic
<bound method A._semipublic of ......
>>> A.__private
AttributeError: class A has no attribute '__
```

▼ Details

Python is a class based object language. Classes are declared using the `class` keyword. Classes may contain any number

of attributes. Attributes may be of any time, including functions. A function attribute of a class is called a **`method`**. There are two kinds of methods in a python class. Those which first argument is named **`self`**. And the others, hich do not differ in terms of behaviour from any other python function.s

Methods which first argument is named **`self`** are sometimes called instance methods. They require as a first argument an instance of the class. Instantiation in python is obtained by invoking the class with an opening and closing parenthesis such as x = Account(). The result is an instance object. The getCurrentBalance method can then be called on x and returns an integer result.

It is usual in python to use the underscore sign "_" to name a method which is intended for private use only, for example for implementation purpose. However, this is only a naming convention, unlike methods starting with a double underscore "__" which are really private and can only be used within code defined in another method of the same class.

Instance methods accessed either from their class or from an instance. In the first case, they have an **`unbound method`** type and require an instance as their first argument (self). In the second case they have a **`bound method`** type and do not require to provide the **`self`** argument.

Class Constructor

```
class NewAccount(Account):

   def __init__(self, original)
     self.original = original

   def getCurrentBalance(self, balance):
      return balance + self.original
```

```
>>> x= NewAccount(222)
>>> x.getCurrentBalance(1000)
1222
```

▼ Details

During instantiation, it may be useful to set initial properties of the instance. This is possible by using the __init__(self) method. . __init__() is known as a constructor. It is the first method and is called right after the first instance is created. **`self`** is the first argument for each method and represents the instance which has been created. In the example above, the argument **`original`** given to the class Account at creation time is passed to the __init__() method and stored as a property of the new object referenced by **`self`**. That property is later used when the method **`getCurrentBalance`** is called.

Modules

```
def setValue():
  for i in range(0,10):
    value= 2*i
    print value
  return value

def printValue(value):
  valuelist=[]
  for i in range(0,10):
    valuelist.append(value)
  return valuelist
```

```
def create():
  for x in range(0,5):
    y= 2*x +3
    print y
  return y
```

▼ Details

When you write long programs, it is always better to write them into multiple files for easier maintenance. You may also need a function for several programs without having to copy its definition into each program. Python's way of dealing with this issue is the use of modules. Modules in Python are files containing definitions used in a script. Modules can also contain executable statements. Definitions in a module can be imported into other modules. In the example above, three modules are created. The first module ValueModule.py has two functions definitions setValue() and printValue(), the second module CreateModule.py has one function create(). The third module Application.py(in the next slide) uses both modules for its function calls. For that, this third module needs to import them by doing import ValueModule and import CreateModule(). Then you can do a function call to use the first module by doing ValueModule.setValue() or ValueModule.printValue(10) or CreateValue.create() to use the second module.

Modules(2)

```
#!/usr/bin/python

import ValueModule
import CreateModule
from ValueModule import printValue

ValueModule.setValue()
printValue(10)
CreateModule.create()
```

```
#chmod +x Application.py
#./Application.py
0
2
4
6
8
10
12
14
16
18
[10, 10, 10, 10, 10, 10, 10, 10, 10,
3
5
7
9
11
11
```

▼ Details

This third module Application.py contains functions calls for the previous modules ValueModule.py and CreateModule.py. To be able to do those functions calls, we first need to import both modules. To import modules, we can use either ***`import module`*** or ***`from module import function`*** . This latter expression just keeps us from having to write

module.function() when we do the function call; we can just write ***function()*** .ValueModule.setValue() is the function call to the first function on the ValueModule.py. This function call will print out on the screen 2*i for all the values of i between 0 and 9. The next function call printValue(10) prints out a list containing the element 10 ten times. The last function call CreateModule.create() prints on the screen 2*x +3 for all x between 0 and 4.

Functional programming

```
y=lambda i : 3*i + 5
```

```
>>> y(2)
11
```

```
def f(x):
    if x>3:
        return x
```

```
>>> filter(f, range(2,6))
[4,5]
```

```
def f (x):
    x*x
```

```
>>> map( f, range(1,4))
[1,4,9]
```

```
def f (x,y):
    y-x
```

```
>>> reduce( f, range(1,4))
2
```

```
>>> zip((1,2,3),[0,0,0])
[(1,0), (2,0),(3,0)]
```

▼ Details

There are a lot built-in functions(map,filter, lambda, reduce and zip) to handle functional programming in Python. lambda() is used to create a small function. In the example, by defining a lambda function y to compute 3i + 5 for all i. Calling then y(2) will directly produces the result. filter(function, sequence) takes two arguments: a function and a sequence and will return a sequence consisting of the elements for which the function is true. In the example above, the function f takes a parameter x and returns that parameter if x is greater than 3. So, doing filter(f,range(2,6)) will return a list containing only the values 4,5. map(function,sequence) also takes two arguments: a function and a sequence, but will return a list containing all the return values obtained by calling the function for each item in the sequence. In the example, map(f, range(1,4)) returns the list [1,4,9] which is the list of the square of the numbers 1,2,3 in range(1,4). reduce() also takes two arguments : the function and a sequence, and returns a single value which is the result obtained by calling the function on the first two items of the sequence, then on their result with the next item, and so on... reduce(f, range(1,4)) returns the value 2 obtained by doing 3-(2-1). zip() takes as arguments multiple sequences and combines them into one single sequence of tuples. Each tuple will contain the corresponding elements from the sequence arguments. In the example, zip([1,2,3],[0,0,0]) returns the sequence [(1,0),(1,0), (1,0)].

Iterators

```
class FactRange:

  def __init__(self, range_min, range_max):
    self.current = 0
    self.fact = 1
    self.range_min = range_min
    self.range_max = range_max

  def next(self):
    if self.current >= self.range_max:
      raise StopIteration
    if self.range_min >= self.range_max:
      raise StopIteration
    while self.current < self.range_min:
      self.fact = self.fact * max(1, self.curren
      self.current += 1
    self.fact = self.fact * self.current
    self.current += 1
    return self.fact

   def __iter__(self):
    return self
```

```
>f=FactRange(3,10)
>for i in f:
...   print i
6
24
120
720
5040
40320
362880
```

▼ Details

__iter__() and next() are the two methods used to define an iterator. __iter__() function returns an iterator object usually self that uses the method next() to access each element one at a time. In the example above, by using the methods __iter() and __next(), we are telling the intepreter that we want to use the class FactRange as an iterator. By creating the instance f of the class FactRange, f is also an iterator. So, when for i in f, print i is called, the intepreter keeps calling f.next(), it assigns also the factorials returned by that function to i and prints them on the screen. The use of iterators in Python is very elegant and doesn' t take up a lot of memory because the sequence is not computed all at once; each element in the sequence is only computed at the time when it is needed. This is very useful when we loop through an infinite number of iterations. The use of the exception StopIteration is important here because it allows to end the loop when the upper limit range_max is reached or in the case where the lower limit range_min is greater than the upper limit range_max.

Generators: yield

```
def facto(range_min, range_max):
    current = 0
    fact =1
    while current < range_max:
      while current < range_min:
        fact = fact * max(1,current)
        current += 1
      # Compute the current factorial
      fact = fact * current
      current += 1
      yield fact
```

```
>>> for f in facto(3,10):
  print f
6
24
120
720
5040
40320
362880
```

▼ Details

Generators are just another way to create iterators. With generators, __iter__() and next() don' t need to be specified, they are created automatically. Generators use the same syntax than regular functions, but instead of return, they have the keyword yield for whenever they want to return data. When the yield is executed, the function remembers all the previous

values and where the last statement was executed, so that when the generator function is called, it will start at the line following the yield. The generator function from the example above does the same thing that was done in the previous slide using a class. This generator function computes the factorial of all the numbers between two numbers given as parameters. Another interesting feature with generators is that they automatically raise StopIteration when they terminate.

Exceptions

```
5/ 0
```

```
Traceback (most recent call last)
File "<stdin>" line 1 in ?
ZeroDivisionError: integer division or modulo by
```

```
P= 2*a
```

```
Traceback (most recent call last)
File "<stdin>", line 1, in ?
NameError: name "a" is not defin
```

```
def f(x):
  try:
   result = 1/x
  except ZeroDivisionError:
   result = "N.A."
  return result
```

```
f(1)
1
f(0)
N.A.
```

▼ Details

Exceptions are the errors detected at execution time. Exceptions are syntatically correct, but produce an error message when you try to execute them. There are different types of exceptions. The First shown in the example is the ZeroDivisionError exception obtained because we tried to divide an integer by 0. The second one NameError is obtained because we tried to assign to a variable another variable that has not been defined before. The example on the right shows a way to handle the ZeroDivisionError exception. In the example, try followed by result = 1/x computes the inverse of each parameter of the function as long as there is no ZeroDivisionError exception. The except keyword allows the program to print a N/A on the screen there is a ZeroDivisionError exception.

The Python Standard Library

- **re**: regular expressions
- **os**: platform -independent interaction with the operating system (files, directories, etc.)
- **sys**: access to interpreter internals
- **popen2**: creation of subprocesses
- **math**: standard C library functions for floating point maths
- **urllib2** and **smtplib**: access to internet protocols (HTTP, SMTP)
- **datetime**: date and time access
- **threading**: platform - independent threads
- etc.

▼ Details

Python standard library includes many modules useful in your interactions with the interpreter. ***os*** provides numerous functions to interact with the operating system while ***sys*** is another module that gives access to the Python's interpreter internals. With its argv attribute, ***sys*** can store command line arguments and with its attributes like stdin, stdout, stderr, ***sys*** is able to emit warnings and error messages. ***Re*** is used to provide regular expressions for advanced string processing. ***math*** gives you access to the standard C library functions for floating point maths. ***urlib2*** and ***smtplib*** are two modules for accessing the internet and processing internet protocols. ***urlib2*** retrieves data from URLs while ***smtplib*** is used to send mails. ***datetime*** is used to manipulate dates and times. ***threading*** is also another module used for platform independent threads. With this library, we can run tasks on the background while the main program is running. Other modules in the standard library are ***random*** for making random selection , ***timeit*** for performance measurement, ***gzip, zlib*** , ***zipfile*** and ***tarfile*** for data compression and archiving, etc.

More Python Libraries

- **Numpy / Scipy**: high performance scientific computing
- **PyQt**: cross platform rapid GUI application development
- **Twisted**: high performance networking
- **SIP**: C/C++ bindings
- **PIL**: imaging
- **PyGame, PyOpenAL, PySDL, Soya3D**: high performance graphics
- **PyMedia**: video processing
- **SimPy**: system simulation
- **BioPython**: bioinformatics
- **Psyco**: just -in -time compiler
- **Jython**: Java integration
- etc.

▼ Details

The python community has created numerous libraries to turn python into a universal development environment.

`NumPy` and `SciPy` provide a complete collection of libraries for scientific computing and in particular for high speed matrix operations. `PyQt` is a great toolkit to create applications for Mac, Windows, Linux and even PDAs.

`Twisted` is an innovative library to create high performance networking applications which support most Internet

protocols.

In the are of gaming and graphics, python includes a wide range of high performance libraries. Games such as `slune` demonstrate how python can be used for real time interactive applications. Python can even support hundreds of multimedia file formats through the integration of ffmpeg codes in the `PyMedia` library.

Python is used in bioinformatics (`BioPython`), system simulation (`SimPy`).

Integration with other languages can be performed through binding libraries such as `SIP` for `C` or `C++` code. `Java` integration is possible through `Jython`, the Java based implementation of Python.