Post Exam Python-Functions

Looking at what a module contains, and its documentation ¶

Once a module is imported, we can list the symbols it provides using the `dir` function:

In [1]:

```
import math

print(dir(math))
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'c
os', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'facto
rial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log1
0', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

And using the function `help` we can get a description of each function (almost .. not all functions have docstrings, as they are technically called, but the vast majority of functions are documented this way).

In [2]:

```
help(math.log)
```

```
Help on built-in function log in module math:

log(...)
    log(x, [base=math.e])
    Return the logarithm of x to the given base.

    If the base not specified, returns the natural logarithm (base e) of
x.
```

We can also use the `help` function directly on modules: Try

```
    help(math)
```

Some very useful modules form the Python standard library are `os`, `sys`, `math`.

A complete lists of standard modules for Python 2 and Python 3 are available at http://docs.python.org/2/library/ (http://docs.python.org/2/library/) and http://docs.python.org/3/library/ (http://docs.python.org/3/library/), respectively.

Variable Names

Variable names in Python can contain alphanumerical characters `a-z`, `A-Z`, `0-9` and some special characters such as `_`. Normal variable names must start with a letter.

By convention, variable names start with a lower-case letter, and Class names start with a capital letter.

In addition, there are a number of Python keywords that cannot be used as variable names. These keywords are:

```
and, as, assert, break, class, continue, def, del, elif, else, except,
exec, finally, for, from, global, if, import, in, is, lambda, not, or,
pass, print, raise, return, try, while, with, yield
```

Note: Be aware of the keyword `lambda`, which could easily be a natural variable name in a scientific program. But being a keyword, it cannot be used as a variable name.

Functions

A *function* is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like *print()*, etc. but you can also create your own functions. These functions are called **user-defined functions**.

Defining a Function

You can define functions to provide the desired functionality.

- Function blocks begin with the keyword `def` followed by the `function name` and parentheses `()` followed by a colon `:`.
- Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability. (this convention is not always followed and does not prevent operation, but it is good to develop a style and stick to it.
- Any input parameters or arguments should be placed within the parentheses. You can also define parameters inside these parentheses.
- The code block within every function starts is after the colon `:` and is indented.
- The first statement of a function can be an optional statement (although you will not gain full marks in this course if not included) - the documentation string of the function or docstring.
- The docstring is bounded by `"""...."""`
- The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A `return` statement with no arguments is the same as `return None`.

User defined functions: Example

In [3]:

```
def func0():
    print("test")
```

Calling a Function

Defining a *function* only gives it a *name*, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the *Python* prompt.

In [4]:

```
func0()
```

```
test
```

User defined functions: Example2

In [5]:

```
def func1(s):
    """
    Print a string 's' and tell how many characters it has
    """

    print(s + " has " + str(len(s)) + " characters")
```

In [6]:

```
func1("test")
```

```
test has 4 characters
```

Help(functionname) or functionname?

Using the function `help` will return a description of each function (almost .. not all functions have docstrings, as they are technically called, but the vast majority of functions are documented this way).

In [7]:

```
help(func1)
```

```
Help on function func1 in module __main__:

func1(s)
    Print a string 's' and tell how many characters it has
```

User defined functions: `return`

Functions that returns a value use the `return` keyword:

In [8]:

```
def square(x):
    """
    Return the square of x.
    """
    return x ** 2
```

In [9]:

```
square(4)
```

Out[9]:

```
16
```

User defined functions: Example 3

We can return multiple values from a function using tuples:

In [10]:

```
def powers(x):
    """
    Return a few powers of x.
    """
    return x ** 2, x ** 3, x ** 4
```

In [11]:

```
powers(3)
```

Out[11]:

```
(9, 27, 81)
```

In [12]:

```
x2, x3, x4 = powers(3)

print(x3)
```

```
27
```

Scope of a variable

A *local* variable is defined inside the Python function. *Local* variables are only accessible within their local scope. A *global* variable is defined outside the Python function. *Global* variables are accessible throughout the program.

Pass by reference vs value

All parameters (arguments) in the *Python* language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example:

User defined example 4

In [13]:

```
# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist.append([1,2,3,4]);
   print( "Values inside the function: ", mylist)
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print( "Values outside the function: ", mylist)
```

```
Values inside the function:  [10, 20, 30, [1, 2, 3, 4]]
Values outside the function:  [10, 20, 30, [1, 2, 3, 4]]
```

Here, we are maintaining reference of the passed object and appending values in the same object. Hence thr result obtained. However if instead we assign (using the assignment operator "=" a new reference to the passed object *mylist* it becomes a new *local* variable onside the function and the *mylist* outside the funcion is unchanged.

In [14]:

```
# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist = [1,2,3,4]; # This would assig new reference in mylist
   print( "Values inside the function: ", mylist)
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print ("Values outside the function: ", mylist)
```

```
Values inside the function:  [1, 2, 3, 4]
Values outside the function:  [10, 20, 30]
```

User defined example 5

Default argument and keyword arguments

In a definition of a function, we can give default values to the arguments the function takes:

In [15]:

```
def myfunc(x, p=2, debug=False):
    if debug:
        print("evaluating myfunc for x = " + str(x) + " using exponent p = " + str(p))
    return x**p
```

If we don't provide a value of the `debug` argument when calling the the function `myfunc` it defaults to the value provided in the function definition:

In [16]:

```
myfunc(5)
```

Out[16]:

```
25
```

In [17]:

```
myfunc(5, debug=True)
```

```
evaluating myfunc for x = 5 using exponent p = 2
```

Out[17]:

```
25
```

If we explicitly list the name of the arguments in the function calls, they do not need to come in the same order as in the function definition. This is called *keyword* arguments, and is often very useful in functions that takes a lot of optional arguments.

In [18]:

```
myfunc(p=3, debug=True, x=7)
```

```
evaluating myfunc for x = 7 using exponent p = 3
```

Out[18]:

```
343
```

Unnamed functions (lambda function)

In Python we can also create unnamed functions, using the `lambda` keyword:

In [19]:

```
f1 = lambda x: x**2

# is equivalent to

def f2(x):
    return x**2
```

In [20]:

```
f1(2), f2(2)
```

Out[20]:

```
(4, 4)
```

Unamed functions (lambda)

This technique is useful for example when we want to pass a simple function as an argument to another function, like this:

In [21]:

```
# map is a built-in python function
map(lambda x: x**2, range(-3,4))
```

Out[21]:

```
<map at 0x150443d4128>
```

In [22]:

```
# in python 3 we can use `list(...)` to convert the iterator to an explicit list
list(map(lambda x: x**2, range(-3,4)))
```

Out[22]:

```
[9, 4, 1, 0, 1, 4, 9]
```

Exceptions

In Python errors are managed with a special language construct called "Exceptions". When errors occur exceptions can be raised, which interrupts the normal program flow and fallback to somewhere else in the code where the closest try-except statement is defined.

To generate an exception we can use the `raise` statement, which takes an argument that must be an instance of the class `BaseException` or a class derived from it.

In [23]:

```
raise Exception("description of the error")
```

```
---------------------------------------------------------------------------
-
Exception                                 Traceback (most recent call las
t)
<ipython-input-23-c32f93e4dfa0> in <module>()
----> 1 raise Exception("description of the error")

Exception: description of the error
```

Exceptions

A typical use of exceptions is to abort functions when some error condition occurs, for example:

```
def my_function(arguments):

    if not verify(arguments):
        raise Exception("Invalid arguments")

    # rest of the code goes here
```

To gracefully catch errors that are generated by functions and class methods, or by the Python interpreter itself, use the `try` and `except` statements:

```
try:
    # normal code goes here
except:
    # code for error handling goes here
    # this code is not executed unless the code
    # above generated an error
```

For example:

In [24]:

```
try:
    print("test")
    # generate an error: the variable test is not defined
    print(test)
except:
    print("Caught an exception")
```

```
test
Caught an exception
```

To get information about the error, we can access the `Exception` class instance that describes the exception by using for example:

```
except Exception as e:
```

In [25]:

```
try:
    print("test")
    # generate an error: the variable test is not defined
    print(test)
except Exception as e:
    print("Caught an exception:" + str(e))
```

```
test
Caught an exception:name 'test' is not defined
```