Chapter Five
Functions, Modules and packages

Kibret Zewdu (MSc.)

Department of Computer Science
College of Informatics
University of Gondar

February 2, 2023

Outline

Introduction

- Python provides several ways to modularize the source code: functions, modules, packages, and classes.
- are important for re-usability, readability, maintenance and improves performance.
- This chapter covers functions, mudules and packages

Functions

- Functions are the traditional way to modularize code.
- Functions helps you to organize your code into nice, tidy blocks that make it easy to maintain and to understand.
- A function takes values (called arguments or parameters), executes some operation based on them and returns a value.
- We have already seen several Python built-in functions such as `len()`, `print()`, `input()`, `max()`, `min()` and soon.
- The general syntax of a function is:

```
def function_name(argument1, argument2, ...):
        """ Optional Function description (Docstring)
        ... FUNCTION CODE ...
        return DATA
```

Functions

```
        # A simple function to calculate molar volumes
        def calculateMolarVolume(mass,density):
                volume = mass/density
                return volume
```

- In the given function definition includes two arguments, mass and density, which are used within the function to calculate volume, the molar volume.
- The return statement at the end of the function specifies what (if anything) the function returns when it ends.
- A function with no return statement is the same as a function with an empty return statement, that is, it returns nothing.

Functions

NOTE: The indent in Python code is syntactically meaningful. In Python, the indent is used to show where a particular block of code begins and ends. All of the code inside a Python function must be indented to define it as belonging to the function.

```
def explainIndentedCode():
        print('This indented line is part of the function')
        print('So is this one')
print('This unindented line is not a part of the function')
```

Functions

Function to calculate the net charge of a protein:

```
def protcharge(aa_seq):
        """Returns the net charge of a protein sequence"""
        protseq = aa_seq.upper()
        charge = -0.002
        aa_charge = {'C':-.045, 'D':-.999, 'E':-.998, 'H':.091,
                'K':1, 'R':1, 'Y':-.001}
        for aa in protseq:
        charge += aa_charge.get(aa,0)
        return charge
```

To "use" the function, it must be called with the parameter:

```
>>> protcharge('EEARGPLRGKGDQKSAVSQKPRSRGILH')
4.094
```

Function Scope

- Variables declared inside a function are valid only inside the function. That means, if you try to access to a variable from outside the function, Python won't find it.
- To access the contents of a function variable from outside the function, the variable must be returned to the main program by using the ***return*** statement.

```
def duplicate(x):
        y = 1
        print('y = {0}.format(y))
        return (2*x)
```

The scope of y is inside the **duplicate** function. We can say that the function provides a namespace where the name y "lives."

Function Scope

- Order of preference when looking for names
  - First in the scope it was called, and then outside until reaching the global scope.
  - If the name is defined in the namespace provided by the function and outside, Python will use the first available name, that is, the one inside the function

Function Parameter Options

Placement of Arguments

- Up to this point the arguments were put in the same order as originally defined. i.e If you slip the order of arguments, you will get an error message.
- By using variable names the order of parameters is irrelevant.

Function Parameter Options

Arguments with Default Values

- The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
        while True:
                ok = input(prompt)
                if ok in ('y', 'ye', 'yes'):
                        return True
                if ok in ('n', 'no', 'nop', 'nope'):
                        return False
                retries = retries - 1
                if retries < 0:
                        raise ValueError('invalid user response')
                print(reminder)
```

Now the function can be called with only one parameter:

Function Parameter Options

Arguments with Default Values

This function can be called in several ways:

- giving only the mandatory argument:
  `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments:
  `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments:
  `ask_ok('Overwrite the file?', 2, 'Only yes or no!')`

Function Parameter Options

Arguments with Default Values

The default values are evaluated at the point of function definition in the defining scope

```
1           i = 5
2           def f(arg=i):
3                   print(arg)
4           i = 6
5           f()
```

will print 5.

Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes.

Function Parameter Options

Keyword Arguments

Functions can also be called using keyword arguments of the form kwarg=value.

```
def parrot(voltage, state='a stiff', action='voom',
 type='Norwegian Blue'):
        print("-- This parrot wouldn't", action, end=' ')
        print("if you put", voltage, "volts through it.")
        print("-- Lovely plumage, the", type)
        print("-- It's", state, "!")
```

For the above function it accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`).

Function Parameter Options

Keyword Arguments

The above function can be called in any of the following ways:

```
parrot(1000)                    # 1 positional argument
parrot(voltage=1000)            # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')  # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)  # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

but all the following calls would be invalid:

```
parrot()    # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
parrot(110, voltage=220)  # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

In a function call, keyword arguments must follow positional arguments.

Function Parameter Options

Undetermined Numbers of Arguments

- Functions can have variable numbers of arguments if the final parameter is preceded by a "*".
- Any excess arguments will be assigned to the last parameter as a tuple:

```
def average(*numbers):
        if len(numbers)==0:
                return None
        else:
                total = sum(numbers)
        return total / len(numbers)
```

- In this way the average function can be called with an undetermined number of arguments:

```
>>> average(2,3,4,3,2)
>>> average(2,3,4,3,2,1,8,10)
```

Modules

- A module is a file with function definitions, constants, or any type of object that you can use from other modules or from your main program.
- Modules also provide namespaces, so two functions may be given the same name provided that they are defined in different modules.
- To create a module, you have to create a file and save it with the ".py" extension.
- The name of the module is taken from the name of the file.
- If the module filename is `my_module.py`, the module name is `my_module`.

Modules

Creating Modules

- Create a file and save it with the ".py" extension.
- It should be saved in a directory where the Python interpreter searches for it, like those in the PYTHONPATH variable
- For example, store the function `save_list` in a module and call it **utils**. For this, create the file **utils.py** with the following contents:

```
# utils.py file
def save_list(input_list, file_name='temp.txt'):
"""A list (input_list) is saved to a file (file_name)"""
with open(file_name, 'w') as fh:
print(*input_list, sep='\n', file=fh)
return None
```

Modules

Creating Modules

- This way, this function (`save_list`) can be used from any program, provided that this file is saved in a location accessible from Python:

```
>>> import utils
>>> utils.save_list([1,2,3])
```

Modules

Using Modules

- To access the contents of a module, use import.
- It is customary to place the import statement at the beginning of the program.
- There are many ways to use import.
- The most used form is by calling a module by its name.
- To call the built-in module **os**, use,

```
import os
```

- When a module is imported for the first time, its contents are executed.
- If the module is imported more than once, the successive imports will not have any effect.

Modules

Using Modules

- Once a module is imported, to access a function or a variable, use the name of the module as a prefix:

```
>>> os.getcwd()
'/mnt/hda2'
>>> os.sep
'/'
```

- We can also import from a module only a required function. This way we can call it without having to use the name of the module as a prefix.

```
>>> from os import getcwd
>>> getcwd()
'/mnt/hda2'
```

Modules

Using Modules

- To import all the contents of a module, use the "*" operator (**asterisk**):

```
>>> from os import *
>>> getcwd()
'/mnt/hda2'
>>> sep
'/'
```

- import a module using a different name:

```
>>> from fibo import fib as fibonacci
fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Packages

- A package is a group of modules with some characteristics in common.
- They are directories with the modules or other directories inside.
- Also contains a special file named `__init__.py`.
  - This file indicates that the directory it contains is a Python package and can be imported as a module.
- The`__init__.py` files are required to make Python treat the directories as containing packages. In most cases, `__init__.py` is an empty file, but it can also execute initialization code for the package.

Packages

```
Bio/                            #Top-level package
        __init__.py            #Initialize the sound package
        Align/                    #Subpackage for Alignment related so
                __init__.py
                AlignInfo.py
        Alphabet/             #Subpackage for amino-acid alphabets
                __init__.py
                IUPAC.py
                Reduced.py
                Blast/             #Subpackage for Blast parsers
                        __init__.py
                        Applications.py
                        NCBIStandalone.py
                        NCBIWWW.py
                        NCBIXML.py
                        ParseBlastTable.py
                        Record.py
```

Packages

Users of the package can import individual modules from the package, for example:

```
import Bio.Blast.Applications
```

Even if there are differences between modules and packages, both terms are used interchangeably.

Installing Third-Party Modules

Python comes with several modules (built-in modules).

- These modules are bundled with Python so they are ready to use as soon as you have a working Python interpreter

There are also third-party modules that extend Python functionality. Install using the native way and the prefered method known as **Pip** Command
python modules can be installed

```
$ pip install MODULE_NAME
```

Additional Resources

- Defining Functions
  https://docs.python.org/3/tutorial/controlflow.html#defining-functions
- Modules, the Python tutorial.
  http://docs.python.org/tutorial/modules.html
- Installing Python modules.
  http://docs.python.org/install/index.html