Lecture 11

Functions

Functions

A function is a groups of statements made to execute them more than once in a program. A function has a name.

Functions can compute a result value and can have parameters that serve as function inputs which may differ each time when function is executed.

Functions are used to:

- Reduce the size of code as it increases the code reusability
- Split a complex problem in to multiple modules (functions) to improve manageability

The Functions

Sequential codes are easy for small scale programs. It becomes harder to keep track of details when code size exceeds.

Advantages:

- Modularity
- Abstraction
- Code reusability

Disadvantages:

- Switching to a function takes time and memory

Module: independent functionalities in a project that can be implemented and tested independently and that these functionalities can be implicated in single project to perform its task.

A function is similar to sequential code, but in functions we assign name to sequential to call for multiple time.

Abstraction means that implementation level details will be hidden from and only superficial level is know, e.g. print(len). But the code that is written inside the function is hidden.

Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Components of a function:

- Function code - is something that is written inside function or function definition.
- Function call is mandatory for the execution of function.

Functions

Function Components:

- Function Signature
  - * Function Name
  - *Function Arguments (optional)
- Doc string
- Function Body
- Function Return Statement

Syntax:

```
def function_name (arguments):
    doc string
    body
    return statement
```

Function can be used without argument. E.g. Function of length, it accept argument and then return length of the argument.
Function body is the code.
Return statement if needed.

Arguments

Information can be passed into functions as arguments.
Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma. The terms parameter and argument can be used for the same thing: i.e. information that are passed into a function.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
def my_function(fname):
  print(fname + " Apple")

my_function("Red")
my_function("Yellow")
my_function("Green")
```

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

```
def my_function(fname, lname):
  print(fname + '', lname)

my_function("Red", 'apple')
my_function("Yellow", 'banana')
my_function("Green", 'chilli')
```

Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

```
def my_function(country = "Pakistan"):
  print("I am from " + country)

my_function("Karachi")
my_function("Lahore")
my_function()
my_function("DI khan")
```

```
= RESTART: C:\Users\He
8\def.py
I am from Karachi
I am from Lahore
I am from Pakistan
I am from DI khan
>>>
```

Coding Functions

- def is executable code
- def creates an object and assigns it to a name
- return sends a result object back to the caller
- yield sends a result object back to the caller, but remembers where it left off.
- global declares module-level variables that are to be assigned
- nonlocal declares enclosing function variables that are to be assigned
- Arguments are passed by assignment (object reference).
- Arguments are passed by position, unless you say otherwise.
- Arguments, return values, and variables are not declared

Functions

Example – 1:

Write a function that takes radius of a circle from user and returns its circumference.

```
def circle (r):
    return 2*3.14*r
```

```
=== RESTART: C:/Users/H
>>> print(circle(5))
15.700000000000001
>>> print(circle(3))
9.42
>>> x=7
>>> print(circle(x))
21.98
>>> r=3
>>> print(circle(r))
9.42
```

Functions

Example – 1: Understanding doc function.

```
def circle (r):
    '''returns the circumference of circle'''
    return 2*3.14*r
print(circle.__doc__)
```

Docstrings:

- Python docstrings are the string literals that appear right after the definition of a function, method, class, or module.

Docstring

- Docstring is the documentation for a function.
- Docstrings are associated with the object as their __doc__ attribute. We can later use this attribute to retrieve this docstring
- Doc Strings – enclosed in triple single or double quotes. Defines something that the function is doing.
- Built-in variable __doc__ we can print the value of docstring.
- Diff between doc string and hash sign? Comments will not be executed or accessed by any variable, while doc string is accessible.

Docstring

- Functions should be documented for the function users.
- Docstrings are the documentation for a function.
- Example: Implement a function called circle, which is passed a value for radius and it returns the perimeter of the circle.

```
def circle(r):
""" Arguments: radius of a circle (int)
Returns: perimeter of the circle (float)"""
return 2*3.14*r
```

Functions

Example – 2: Modifying example-1 to return area of circle too.

```
File  Edit  Format  Run  Options  Window  Help
def circle (r):
    '''returns the circumference of circle'''
    return r*3.14, 3.14*r**2
print(circle.__doc__)
```

```
=== RESTART: C:/Users/Hera Noor/AppDa
returns the circumference of circle
>>> print(circle(5))
(15.700000000000001, 78.5)
>>> x=7
>>> print(circle(x))
(21.98, 153.86)
>>> r=3
>>> print(circle(r))
(9.42, 28.26)
>>>
```

Note: The return statement is tuple as it is separated by comma.

Functions

Example – 3: Modifying example-2 to return area of circle via variables.

```
def circle (r):
    area=3.14*r**2
    circumference=r*3.14
    return area, circumference
```

```
=== RESTART: C:/Users/Hera Noor/AppData
>>> print(circle(5))
(78.5, 15.700000000000001)
>>> x=7
>>> print(circle(x))
(153.86, 21.98)
>>> r=3
>>> print(circle(r))
(28.26, 9.42)
>>>
```

We are utilizing variables and variables
have the values which are calculated against the formula provided by the programmer.

Functions

Example – 4: Accessing values of tuples.

```
>>> result=circle(5)
>>> print('area=', result[0],'and',
'circumference=',result[1])
area= 78.5 and circumference= 15.700000000000001
>>>
```

Return function is returning two variables stored in result.

Result is returning two values [0] value which is at zeroth index of return tuple as same for [1].

Here, result is in tuple format, so we are mapping the result value by [0] and [1].

Functions

Example – 5: Modifying example-4 to print area and circumference of circle too.

```
def circle (r):
    print('circumference=',r*3.14)
    print('area=',3.14*r**2)
```

```
>>> r=5
>>> circle(r)
circumference= 15.700000000000001
area= 78.5
>>> circle(7)
circumference= 21.98
area= 153.86
>>>
```

Help function

The help function is used to view documentation

```
>>> help(circle)
Help on function circle in module __main__:

circle(r)

>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.

>>> help(int)
```

Squeezed text (266 lines).

Functions

Example – 6: Write a function to count ODD values in a list.

In line "odd_count=odd(mylist)", we call our function named odd to apply on mylist, which means that we are passing list to the function and the function will be called. 'mylist' will be mapped to 'list' in the function odd.

```
def odd (list):
    count=0
    for i in list:
        if i%2!=0:
            count+=1
    return count
mylist=[3,2,1,5,7,8,2]
odd_count=odd(mylist)
print(odd_count)
```

Functions

Example – 7: Write a function to count negative values in a list.

```
def neg (list):
    count=0
    for i in list:
        if i<0:
            count+=1
    return count
mylist=[3,2,-33,4,-60,0,1,-92]
negative_count=neg(mylist)
print(negative_count)
```

The LEGB Rule

The Python interpreter searches a name (variable of function) in the following order:

1. Search the local namespace – L
2. Search the enclosing function's namespace (for nested functions) – E
3. Search the global namespace – G
4. Search the namespace of module builtins – B

The builtin module is automatically loaded every time Python interpreter starts.

- It is the top level execution environment.
- It provides direct access to all built-in identifiers (types/classes and functions) of Python.

The LEGB Rule

Example:

```
def f(b):
    a=b
    return a*b

x=3
print(f(x))
print(x)
```

- f has global namespace as it is defined in the main module.
- a and b, defined in f() have local scopes.
- x defines in main module has global namespace.
- print() function is defined in builtins modules.

ns_main (global)
f
x

ns_f (local)
a
b

ns_print (module builtins)
print

Output
9
3

The Global keyword

The keyword called global used inside a function allows the function to update a global variable.

Practice Problem: Modifying the last code on the previous slide.

```
def h():
    global x
    x+=1
    print(x)

x=5
h()
print(x)
```

Now this is the same global x accessed from this local namespace

Output

6

6

Nested Function

```
def outer():
    x=5
    def inner():
        x=3
        print('Inner x:',x)
    print('Outer x:',x)
    inner()
x=10
outer()
print('Global x:',x)
```

```
= RESTART: C:/Users
 namespace.py
Outer x: 5
Inner x: 3
Global x: 10
>>>
```

Functions (Scope & Argument)

Argument Types

- Arguments or parameters are the information passed to a function.
- Arguments are specified after the function name, inside the parentheses.
- A function can have as more than one arguments, separated with commas.
- There are two type of arguments in Python:
  - Positional Arguments
  - Keyword Arguments

Argument Types

Positional Arguments

- Must be put in correct order when calling the function.
- Do not have any default value called args.

Keyword Arguments

- Include a keyword and equal sign.
- Used to set defaults; have some default value.
- Placed after the positional arguments if any.
- Can be placed in any order among other keyword arguments when calling the function. called kwargs.

Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a tuple of arguments, and can access the items accordingly:

```
def my_function(*kids):
  print("The youngest child is " + kids[0])

my_function("Hadi", "Rahim", "Tina")
```

Keyword Arguments

You can also send arguments with the key = value syntax.

This way the order of the arguments does not matter.

The phrase Keyword Arguments are often shortened to kwargs in Python documentations.

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child1)

my_function(child1 = "Hadi", child2 = "Wara", child3 = "Rahim")
```

Arbitrary Keyword Arguments, **kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.

This way the function will receive a dictionary of arguments, and can access the items accordingly. If the number of keyword arguments is unknown, add a double ** before the parameter name.

```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Hadi", lname = "Rehman")
```

Variable Length Argument - **kwargs

Example:

```
def my_func(**kwargs):
    for item in kwargs.items():
        print (item)
my_func(first ='Computer', last='Programming')
```

```
= RESTART: C:\Users\Hera Noc
\define func namespace.py
('first', 'Computer')
('last', 'Programming')
>>>
```

VLA – Variable Length Argument

- In Python, variable number of arguments can be passed to a function.

Example: Consider the min() function.

min(3,5,66)
min(3,4,2,6,44,1,0,88,6)

} Both are correct.
- min() is a VLA function

- Special symbols are used for this purpose.
  - *<args_name> or *args: for positional / non-keyword arguments.
  - **<argument_name> or **kwargs: for keyword arguments.

VLA *args

Example: The following code used a function to print all objects passed to it, on separate lines.

CODE

```
def my_print(*args):
    for value in args:
        print(value)
my_print('Learning','Python','3')
print()
my_print('Learning','Python','3','and','Java','too')
```

OUTPUT

```
\define func namespace
Learning
Python
3

Learning
Python
3
and
Java
too
>>>
```

VLA - *args

Practice Problem 1: Write function which is passed variable number of integers and it returns the sum of all these numbers.

CODE

```
def adder(*args):
    sum=0
    for value in args:
        sum+=value
    return sum
print(adder(4,5,9,1))
```

OUTPUT

```
= RESTART: C:\
\define func n
19
>>>
```

VLA - *args

Practice Problem 2: Modify code of practice problem 1 to add another argument n. Now the function returns sum/n. n should be the first argument.

CODE

```
def adder(n, *args):
    sum=0
    for value in args:
        sum+=value
    return sum/n
print(adder(2,4,5,9,1))
```

OUTPUT

```
= RESTART:
\define fun
9.5
>>>
```

Module, Packages and Libraries

- A Python module is a py file containing functions and classes.
  - useful code that eliminates the need for writing codes from scratch.

- A package or library is a collection of modules.

- Python has a rich set of libraries.
  - There are over 137,000 python libraries present today.

- Python libraries play a vital role in developing machine learning, data science, data visualization, image and data manipulation applications and more.

Builtin Modules

- It contains all the core built-in types/classes and functions.
- Loaded automatically when the interpreter starts.
- It does not contain all the functions and classes supported by Python.
  - size is kept small for efficiency.
- Some functions of builtins module are:
  - min()
  - max()
  - len()
  - print()
- Other modules are kept in the library called The Standard Library.

The Standard Library

- It is the native library of Python that comes with Python installation.
- The modules of this library need to be imported if used.
- Some modules in The Standard Library include:
  - mathematical functions
  - pseudorandom generators
  - fraction handling
  - date-time functions
  - graphical user interface (GUI) development

See https://docs.python.org/ for details on builtins module and the Standard Library.

Third Party Library

- Python has a huge collection of third-party modules/packages available for usage.
- These modules/packages must be installed and then imported if used.
- Some commonly used Python third party packages:
  - Pillow: for handling different formats of images
  - Matplotlib: for two dimensional graphs and plots
  - Numpy: for array processing
  - Scipy: for scientific and technical computation
  - Pandas: to organize, explore, represent, and manipulate data
  - PyGame: for games
- Package installers:
  - pip installs Python packages
  - conda installs packages written in any language

The import Statement

- When the Python interpreter executes an import statement:
  - the corresponding file containing the module is searched.
  - the module code is run to create objects defined in that module.
  - a namespace is created where the objects live.

- Hence a separated namespace is created for every module.

- An application may consist of several modules:
  - top level module is the main program from which the execution starts. It is called main.
  - the remaining modules are library modules that are imported by the top level module.

How to import…

1. Using import statement

Creates a separate namespace

Example:

```
>>> import math
>>> math.sqrt(49)
```

Single statement can be used to import more than one module.

```
>>> import math,random
```

How to import…

2. Using from with import

- It copies selected functions into the namespace of the importing module.
- Remaining functions are not instantiated.
- No separate namespace is created.
- Functions can be called without the dot operator.

Example:

```
>>> from math import sqrt
>>> sqrt(49)
```

How to import…

3. Using from to import all functions of a module

Example:

```
>>> from math import *
>>> sqrt(49)
```

- Now all the functions are copied to the calling namespace and can be used without the dot operator.
- Not a good idea!!
  - Takes too much space
  - Variable names may clash

Task – 10

1. Develop a simple calculator (using functions) that defines addition, subtraction, multiplication and division operation. User selects the operation and provides the operands then output will be generated.

Thank you!