Computer Programming I

Binnur Kurt, PhD

BAU Bahçeşehir University

binnur.kurt@rc.bau.edu.tr

MODULE 7

FUNCTIONS

Introduction

- The best way to develop and maintain a large program is to construct it from smaller, more manageable pieces.
  - This technique is called divide and conquer.
- Using existing functions as building blocks for creating new programs is a key aspect of software reusability
  - Also a major benefit of object-oriented programming.
- Packaging code as a function allows you to execute it from various locations in your program just by calling the function, rather than duplicating the possibly lengthy code.
  - This also makes programs easier to modify.
  - When you change a function's code, all calls to the function execute the updated version.

Defining Functions

```
In [5]: def isOdd(num):
            """Calculate whether num is odd or not."""
            return num%2 == 1

In [6]: isOdd(13)

Out[6]: True

In [7]: isOdd(42)

Out[7]: False
```

Some popular Python Standard Library Modules

> **`collections`**—Data structures beyond lists, tuples, dictionaries and sets.
> Cryptography modules—Encrypting data for secure transmission.
> `csv`—Processing comma-separated value files (like those in Excel).
> **`datetime`**—Date and time manipulations. Also modules time and calendar.
> `decimal`—Fixed-point and floating-point arithmetic, including monetary calculations.
> **`doctest`**—Embed validation tests and expected results in docstrings for simple unit testing.
> `gettext` and `locale`—Internationalization and localization modules.
> **`json`**—JavaScript Object Notation (JSON) processing used with web services and NoSQL document databases.
> `math`—Common math constants and operations.

Some popular Python Standard Library Modules

> **`os`**—Interacting with the operating system.
> `profile`, `pstats`, `timeit`—Performance analysis.
> **`random`**—Pseudorandom numbers.
> `re`—Regular expressions for pattern matching.
> **`sqlite3`**—SQLite relational database access.
> `statistics`—Statistics functions such as mean, median, mode and variance.
> **`string`**—String processing.
> `sys`—Command-line argument processing; standard input, standard output and standard error streams.
> **`tkinter`**—Graphical user interfaces (GUIs) and canvas-based graphics.
> `turtle`—Turtle graphics.
> **`webbrowser`**—For conveniently displaying web pages in Python apps.

math Module Functions

Function	Description	Example
`ceil(x)`	Rounds *x* to the smallest integer not less than *x*	`ceil(9.2)` is `10.0` `ceil(-9.8)` is `-9.0`
`floor(x)`	Rounds *x* to the largest integer not greater than *x*	`floor(9.2)` is `9.0` `floor(-9.8)` is `-10.0`
`sin(x)`	Trigonometric sine of *x* (*x* in radians)	`sin(0.0)` is `0.0`
`cos(x)`	Trigonometric cosine of *x* (*x* in radians)	`cos(0.0)` is `1.0`
`tan(x)`	Trigonometric tangent of *x* (*x* in radians)	`tan(0.0)` is `0.0`
`exp(x)`	Exponential function e^x	`exp(1.0)` is `2.718282` `exp(2.0)` is `7.389056`
`log(x)`	Natural logarithm of *x* (base *e*)	`log(2.718282)` is `1.0` `log(7.389056)` is `2.0`
`log10(x)`	Logarithm of *x* (base 10)	`log10(10.0)` is `1.0` `log10(100.0)` is `2.0`
`pow(x, y)`	*x* raised to power *y* (x^y)	`pow(2.0, 7.0)` is `128.0` `pow(9.0, .5)` is `3.0`
`sqrt(x)`	square root of *x*	`sqrt(900.0)` is `30.0` `sqrt(9.0)` is `3.0`
`fabs(x)`	Absolute value of *x*—always returns a float. Python also has the built-in function `abs`, which returns an `int` or a `float`, based on its argument.	`fabs(5.1)` is `5.1` `fabs(-5.1)` is `5.1`
`fmod(x, y)`	Remainder of *x*/*y* as a floating-point number	`fmod(9.8, 4.0)` is `1.8`

Default Parameter Values

- When defining a function, you can specify that a parameter has a default parameter value.
- When calling the function, if you omit the argument for a parameter with a default parameter value, the default value for that parameter is automatically passed.

Default Parameter Values

```
In [10]: import math

In [11]: def circle_area(radius = 1) :
             return math.pi * radius * radius ;

In [12]: circle_area()
Out[12]: 3.141592653589793

In [13]: circle_area(1)
Out[13]: 3.141592653589793

In [14]: circle_area(1000)
Out[14]: 3141592.653589793
```

Keyword Arguments

> When calling functions, you can use keyword arguments to pass arguments in any order.

```
In [15]: def volume(price , quantity = 1):
             return price * quantity
```

```
In [16]: volume(100.0)
Out[16]: 100.0

In [17]: volume(price = 100.0)
Out[17]: 100.0

In [18]: volume(100,2)
Out[18]: 200

In [21]: volume(price=100, quantity=2)
Out[21]: 200

In [19]: volume(quantity = 2, price = 100)
Out[19]: 200
```

Arbitrary Argument Lists

> Functions with arbitrary argument lists, such as built-in functions min and max, can receive any number of arguments.

```
In [25]: def avg(*values):
             return sum(values)/len(values)

In [28]: avg(4,15,23)
Out[28]: 14.0

In [29]: avg(4,8,15,16,23,42)
Out[29]: 18.0

In [30]: avg(*range(1,100))
Out[30]: 50.0
```

Scope Rules

- Each identifier has a scope that determines where you can use it in your program.
- **`Local Scope`**
  - A local variable's identifier has local scope.
  - It's "in scope" only from its definition to the end of the function's block.
  - It "goes out of scope" when the function returns to its caller.
  - So, a local variable can be used only inside the function that defines it.

Scope Rules

- Each identifier has a scope that determines where you can use it in your program.
- **`Global Scope`**
  - Identifiers defined outside any function (or class) have global scope—these may include functions, variables and classes.
  - Variables with global scope are known as global variables.
  - Identifiers with global scope can be used in a .py file or interactive session anywhere after they're defined.

Accessing a Global Variable from a Function

> The local x shadows the global x

```
In [31]: x = 42

In [41]: def global_access():
             x = 108
             print('x=',x)

In [42]: global_access()
         x= 108

In [43]: x
Out[43]: 42
```

Accessing a Global Variable from a Function

```
In [44]: x = 42

In [45]: def global_access():
             global x
             x = 108
             print('x=',x)

In [46]: global_access()
         x= 108

In [47]: x
Out[47]: 108
```

Shadowing Functions

```
In [48]: sum = 1 + 2 + 3 + 4 + 5

In [49]: sum
Out[49]: 15

In [50]: sum([1, 2, 3, 4, 5])
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-50-5d2c2c89f40b> in <module>
----> 1 sum([1, 2, 3, 4, 5])

TypeError: 'int' object is not callable
```

`import`: A Deeper Look

- You've imported modules (such as math and random) with a statement like:

  `import` *module_name*

- then accessed their features via each module's name and a dot (.).
- Also, you've imported a specific identifier from a module (such as the decimal module's Decimal type) with a statement like:

  `from` *module_name* **`import`** *identifier*

- then used that identifier without having to precede it with the module name and a dot (.).

Importing Multiple Identifiers from a Module

> Using the `from… import` statement you can import a comma-separated list of identifiers from a module then use them in your code without having to precede them with the module name and a dot (.)

```
In [51]: from math import ceil, floor

In [52]: ceil(3.1415162342)
Out[52]: 4

In [53]: floor(3.1415162342)
Out[53]: 3
```

Avoid Wildcard Imports

> You can import all identifiers defined in a module with a wildcard import of the form

`from` *module_name* **`import`** *

> This makes all of the module's identifiers available for use in your code

> Importing a module's identifiers with a wildcard import can lead to subtle errors

```
In [54]: e = 'Jack Bauer'

In [55]: from math import *

In [56]: e
Out[56]: 2.718281828459045
```

Binding Names for Modules and Module Identifiers

> The import statement's as clause allows you to specify the name used to reference the module's identifiers.

```
In [57]: import statistics as stats

In [58]: lottery = [4, 8, 15, 16, 23, 42]

In [60]: stats.mean(lottery)
Out[60]: 18
```

Passing Arguments to Functions: A Deeper Look

> In many programming languages, there are two ways to pass arguments—pass-by-value and pass-by-reference (sometimes called call-by-value and call-by-reference, respectively):
  - With pass-by-value, the called function receives a copy of the argument's value and works exclusively with that copy.
    - Changes to the function's copy do not affect the original variable's value in the caller.
  - With pass-by-reference, the called function can access the argument's value in the caller directly and modify the value if it's mutable.
> Python arguments are always passed by reference.
> When a function call provides an argument, Python copies the argument object's reference—not the object itself—into the corresponding parameter
  - This is important for performance.

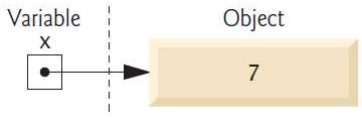
Memory Addresses, References and “Pointers”

> You interact with an object via a reference, which behind the scenes is that object’s address (or location) in the computer’s memory
  – sometimes called a “pointer” in other languages.

```
In [65]: x = 7

In [63]: id(x)

Out[63]: 140725106195520
```



Immutable Objects as Arguments

> When a function receives as an argument a reference to an immutable (unmodifiable) object—such as an int, float, string or tuple—even though you have direct access to the original object in the caller, you cannot modify the original immutable object’s value.

Immutable Objects as Arguments

```
In [68]: x = 42

In [71]: id(x)
Out[71]: 140725106195520

In [69]: cube(x)
         id(number) before modifying number: 140725106195520
         id(number) before modifying number: 2227039275248
Out[69]: 74088

In [70]: id(x)
Out[70]: 140725106195520
```