Functions

Contents

- Function Definition
- Scope of Variables
- More Examples
- Library Modules
- List Comprehension
- Default Argument Values
- Keyword Arguments
- Lambda Expression
- Top-Down Design

Function Definition

- Functions are commonly defined by statements of the form

```
def functionName(par1, par2,...):
    indented block of statements
    return expression
```

  where ***par1***, ***par2*** , . . . are the parameters and the ***expression*** evaluates to a literal of any type

- When defining a function (or a class), it is useful to document it using a string literal called the **doc** string at the start of the definition
  - The **doc** string is enclosed in three sets of double quotes (`"""`)
  - It will be placed in the **__doc__** attribute of the definition
  - It can be accessed using **print(functionName.__doc__)**
- We can also use **help(object)** to get the description of **object**

Function Definition

```
def print_max(a, b): # this function has no return statement
    if a > b:
        print(a, 'is maximum')
    elif a == b:
        print(a, 'is equal to', b)
    else:
        print(b, 'is maximum')

# directly pass literal values
print_max(3, 4)

x = 5
y = 7

# pass variables as arguments
print_max(x, y)
```

[Run]

```
4 is maximum
7 is maximum
```

Function Definition

- The return statement is used to return from a function, i.e., break out of the function
  - We can optionally return a value from the function as well

```
def maximum(x, y):
    if x > y:
        return x
    elif x == y:
        return 'The numbers are equal'
    else:
        return y

print(maximum(2, 3))
print(maximum(5, 5))
```

[Run]

```
3
The numbers are equal
```

* There is a built-in function called **`max`** that implements the 'find maximum' functionality

Function Definition

- Every function implicitly contains a **`return None`** statement at the end unless you have written your own return statement
  - **`None`** is a special type in Python that represents nothingness
  - A **`return`** statement without a value is equivalent to **`return None`**

```
def f():
    return

print(f())
```

[Run]

```
None
```

Scope of Variables

- Local variable:
  - A variable created inside a function
  - Can only be accessed by statements inside that function
  - Local variables are recreated each time the function is called (They cease to exist when the function is exited)
  - Variables created in two different functions with the same name are treated as completely different variables (i.e., variable names are local to the function)
- Global variable:
  - A variable recognized everywhere in a program

Scope of Variables

```
x = 50  # This x is a global variable

def func(x):
    print('x is', x)  # This x is a local variable
    x = 2
    print('Changed local x to', x)

func(x)
print('x is still', x)
```

[Run]

```
x is 50
Changed local x to 2
x is still 50
```

Scope of Variables

```
def main():
    ## Demonstrate the scope of variables.
    x = 2
    print(str(x) + ": function main")
    trivial()
    print(str(x) + ": function main")

def trivial():
    x = 3
    print(str(x) + ": function trivial")

main()
```

[Run]

```
2: function main
3: function trivial
2: function main
```

Scope of Variables

- One way to make a variable global is to place the assignment statement that creates it at the top of the program

- Any function can read the value of a global variable—however, the value cannot be altered inside a function unless the altering statement is preceded by a statement of the form

```
global globalVariableName
```

Scope of Variables

```
x = 50

def main():
    func()
    print('Value of x is', x)

def func():
    global x  # Can declare more than one global variable

    print('x is', x)
    x = 2
    print('Changed global x to', x)

main()
```

[Run]

```
x is 50
Changed global x to 2
Value of x is 2
```

Scope of Variables

- Named constant:
  - A special constant that will be used several times in the program
  - Created as a global variable whose name is written in uppercase letters with words separated by underscore characters

```
INTEREST_RATE = 0.04
MINIMUM_VOTING_AGE = 18

 . . . . . .
interestEarned = INTEREST_RATE * amountDeposited
 . . . . . .
if (age >= MINIMUM_VOTING_AGE):
    print("You are eligible to vote.")
 . . . . . .
```

  - To change the value of a named constant at a later time, you need to alter just one line of code at the top of the program

More Examples

```
def main():
    ## Extract the first name from a full name.
    fullName = input("Enter a person's full name: ")
    print("First name:", firstName(fullName))

def firstName(fullName):
    firstSpace = fullName.index(" ")
    givenName = fullName[:firstSpace]
    return givenName

main()
```

[Run]

```
Enter a person's full name: Franklin Delano Roosevelt
First name: Franklin
```

- The **`index`** method returns the index of the given character in the sequence

More Examples

```
def main():
    ## Calculate a person's weekly pay.
    hourlyWage = float(input("Enter the hourly wage: "))
    hoursWorked = int(input("Enter # hours worked: "))
    earnings = pay(hourlyWage, hoursWorked)
    print("Earnings: ${0:,.2f}".format(earnings))

def pay(wage, hours):
    if hours <= 40:
        amount = wage * hours
    else:
        amount = (wage * 40) + ((1.5) * wage * (hours - 40))
    return amount

main()
```

[Run]

```
Enter the hourly wage: 24.50
Enter # hours worked: 45
Earnings: $1,163.75
```

More Examples

```
def main():
    ## Display the vowels appearing in a word.
    word = input("Enter a word: ")
    listOfVowels = occurringVowels(word)
    print("The following vowels occur in the word:", end=' ')
    stringOfVowels = " ".join(listOfVowels)
    print(stringOfVowels)

def occurringVowels(word):
    word = word.upper()
    vowels = ('A', 'E', 'I', 'O', 'U')
    includedVowels = []
    for vowel in vowels:
        if vowel in word:
            includedVowels.append(vowel)
    return includedVowels

main()
```

– This is an example of a list-valued function

More Examples

[Run]

```
Enter a word: important
The following vowels occur in the word: A I O
```

More Examples

```
INTEREST_RATE = .04    # annual rate of interest

def main():
    ## Calculate the balance and interest earned
    (deposit, numberOfYears) = getInput()
    bal, intEarned = balAndInterest(deposit, numberOfYears)
    displayOutput(bal, intEarned)

def getInput():
    deposit = int(input("Enter the amount of deposit: "))
    numberOfYears = int(input("Enter # of years: "))
    return (deposit, numberOfYears)
def balAndInterest(principal, numYears):
    balance = principal * ((1 + INTEREST_RATE) ** numYears)
    interestEarned = balance - principal
    return (balance, interestEarned)
def displayOutput(bal, intEarned):
    print("Balance: ${0:,.2f}   Interest Earned: ${1:,.2f}"
          .format(bal, intEarned))

main()
```

More Examples

[Run]

```
Enter the amount of deposit: 10000
Enter # of years: 10
Balance: $14,802.44   Interest Earned: $4,802.44
```

- The function **`balAndInterest`** does not return two values but just one value that is a tuple containing two values
- The fourth line of the function **`main`** can be written

```
(bal, intEarned) = balAndInterest(deposit, numberOfYears)
```

More Examples

```
def main():
    ## Custom sort a list of words.
    list1 = ["democratic", "sequoia", "equals", "brrr",
             "break", "two"]
    list1.sort(key=len)
    print("Sorted by length in ascending order:")
    print(list1, '\n')
    list1.sort(key=numberOfVowels, reverse=True)
    print("Sorted by number of vowels in descending order:")
    print(list1)

def numberOfVowels(word):
    vowels = ('a', 'e', 'i', 'o', 'u')
    total = 0
    for vowel in vowels:
        total += word.count(vowel)
    return total

main()
```

More Examples

[Run]

```
Sorted by length in ascending order:
['two', 'brrr', 'break', 'equals', 'sequoia', 'democratic']

Sorted by number of vowels in descending order:
['sequoia', 'democratic', 'equals', 'break', 'two', 'brrr']
```

- To create a custom sort by any criteria we choose we add the optional argument **`key=keyValue`** to the sort method
  - ***`keyValue`*** is the name of a function
  - The function takes each item of the list as input and returns the value of the property we want to sort on
- The argument **`reverse=True`** can be added to sort in descending order

More Examples

- While the **`sort`** method alters the order of the items in a list, the **`sorted`** function returns a new ordered copy of a list

```
>>> list1 = ['white', 'blue', 'red']
>>> list2 = sorted(list1)
>>> list2
['blue', 'red', 'white']
>>> sorted(list1, reverse=True)
['white', 'red', 'blue']
>>> sorted(list1, key=len)
['red', 'blue', 'white']
>>> list1
['white', 'blue', 'red']
>>> sorted('spam')
['a', 'm', 'p', 's']
```

  - While the **`sort`** method only can be used with lists, the **`sorted`** function also can be used with lists, strings, and tuples

Library Modules

- A library module is a file with the extension **`.py`** containing functions and variables that can be used (we say imported) by any program
  - The library module can be created in IDLE or any text editor and looks like an ordinary Python program
- To gain access to the functions and variables of a library module, place a statement of the form **`import moduleName`** at the beginning of the program
  - Any function from the module can be used in the program by prepending the function name with the module name followed by a period

Library Modules

- Assuming that the function **`pay`** in the example program on p.14 is contained in a file named **`finance.py`** that is located in the same folder as the example, the example could be rewritten as

```
import finance

def main():
    ## Calculate a person's weekly pay.
    hourlyWage = float(input("Enter the hourly wage: "))
    hoursWorked = int(input("Enter # hours worked: "))
    earnings = finance.pay(hourlyWage, hoursWorked)
    print("Earnings: ${0:,.2f}".format(earnings))

main()
```

  - The only changes in the program are the replacements of
    - the definition of **`pay`** function with the import statement
    - **`pay`** in the assignment statement with **`finance.pay`**

Library Modules

- There are different ways to import modules, as e.g., to import some functions from the **`random`** module:
  (1) **`from random import randint, choice`**
  (2) **`from random import *`**
  (3) **`import random`**

(1) imports just two functions from the module

(2) imports every function from the module

  – You should usually avoid doing this, as the module may contain some names that will interfere with your own variable names (e.g., a variable **`total`** and a function **`total`**, in imported module)

(3) imports an entire module without interference

  – To use a function from the module, preface it with random followed by a dot, e.g., **`random.randint(1,10)`**

Library Modules

- The **`as`** keyword can be used to change the name that your program uses to refer to a module or things from a module:

```
import numpy as np

from tools import combinations_with_replacement as cwr

from math import log as ln
```

- Usually, these statements go at the beginning of the program, but they can go anywhere as long as they come before the code that uses the module

Library Modules

- Some modules from the Python standard library:

Module	Some Tasks Performed by Its Functions
`os`	Delete and rename files
`os.path`	Determine whether a file exists in a specified folder. This module is a submodule of os
`pickle`	Store objects (such as dictionaries, lists, and sets) in files and retrieve them from files
`random`	Randomly select numbers and subsets
`tkinter`	Enable programs to have a graphical user interface
`turtle`	Enable turtle graphics

List Comprehension

- If **`list1`** is a list, then the following statement creates a new list, **`list2`**, and places **`f(item)`** into the list for each **`item`** in **`list1`**

```
list2 = [f(x) for x in list1]
```

where **`f`** is either a Python built-in function or a user-defined function

```
>>> list1 = ['2', '5', '6', '7']
>>> [int(x) for x in list1]
[2, 5, 6, 7]
>>> def g(x):
        return(int(x) ** 2)

>>> [g(x) for x in list1]
[4, 25, 36, 49]
>>>
```

List Comprehension

- The `for` clause in a list comprehension can optionally be followed by an `if` clause

```
>>> [g(x) for x in list1 if int(x) % 2 == 1]
[25, 49]
>>>
```

- List comprehension can be applied to objects other than lists, such as, strings, tuples, and arithmetic progressions generated by range functions

```
>>> [ord(x) for x in "abc"]
[97, 98, 99]
>>> [x ** .5 for x in (4, -1, 9) if x >= 0]
[2.0, 3.0]
>>> [x ** 2 for x in range(3)]
[0, 1, 4]
>>>
```

List Comprehension

- If `str` is any single-character string, then `ord(str)` is the ASCII value of the character
- If `n` is a nonnegative number, then `chr(n)` is the single-character string consisting of the character with ASCII value `n`

```
>>> print(ord('A'))
65
>>> print(chr(65))
A
```

Default Argument Values

- Some (or all) of the parameters of a function can be made optional and have default values—values that are assigned to them when no values are passed to them
- A typical format for a function definition using default values is

```
def functionName(par1, par2, par3=value3, par4=value4):
```

- Caution: In a function definition, the parameters without default values must precede the parameters with default values

Default Argument Values

```
def main():
    say('Hello')
    say('World', 5)

def say(message, times=1):
    print(message * times)

main()
```

[Run]

```
Hello
WorldWorldWorldWorldWorld
```

Keyword Arguments

- Arguments can be passed to functions by using the names of the corresponding parameters instead of relying on position

```
def main():
    func(3, 7)
    func(25, c=24)
    func(c=50, a=100)
def func(a, b=5, c=10):
    print('a is', a, 'and b is', b, 'and c is', c)

main()
```

[Run]

```
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

- Caution: Arguments passed by position must precede arguments passed by keyword

Lambda Expression

- Lambda expressions are one-line mini-functions
  - Compute a single expression
  - Cannot be used as a replacement for complex functions

```
lambda par1, par2,...: expression
```

```
names = ["Dennis Ritchie", "Alan Kay", "John Backus",
                "James Gosling"]
names.sort(key=lambda name: name.split()[-1])
nameString = ", ".join(names)
print(nameString)
```

[Run]

```
John Backus, James Gosling, Alan Kay, Dennis Ritchie
```

Top-Down Design

- Functions allow programmers to focus on the main flow of a complex task and defer the details of implementation
  - As a rule, a function should perform only one task, or several closely related tasks, and should be kept relatively small
  - Functions are used to break complex problems into small problems, to eliminate repetitive code, and to make a program easier to read by separating it into logical units
- The first function of a program is named main and sometimes will be preceded by import statements and global variables
  - All programs will end with the statement `main()` to call the program's main function
  - The function main should be a supervisory function calling other functions according to the application's logic