FUNCTIONS

CS 3080: Python Programming

University of Colorado
Colorado Springs

Functions

- A major purpose of functions is to group code that gets executed multiple times.

```
print('Howdy!’)
print('Howdy!!!’)
print('Hello there.’)
print('Howdy!’)
print('Howdy!!!’)
print('Hello there.’)
print('Howdy!’)
print('Howdy!!!’)
print('Hello there.')
```

Functions

- A **function** is like a mini-program inside a program.
- We create a function by using the **def** statement.
- This code inside the function is executed when the function is called, not when the function is first defined.

```
def hello():
    print('Howdy!')
    print('Howdy!!!')
    print('Hello there.')

hello()
hello()
hello()
```

Functions with arguments

- The value stored in a parameter is forgotten when the function returns.

```
def hello(name):
    print('Hello ' + name)

hello('Alice')
hello('Bob’)
```

Return statement

- When creating a function using the **def** statement, you can specify what the return value should be with a **return** statement

```
def getAnswer(answerNumber):
    if answerNumber == 1:
        return 'It is certain'
    elif answerNumber == 2:
        return 'Ask again later'
    elif answerNumber == 3:
        return 'My reply is no'
    elif answerNumber == 4:
        return 'Yes'
    elif answerNumber == 5:
        return 'Very doubtful'
    else:
        return 'Number not accepted'


r = random.randint(1, 5)
fortune = getAnswer(r)
print(fortune)
```

Return statement

- When creating a function using the **def** statement, you can specify what the return value should be with a **return** statement

```
def getAnswer(answerNumber):
    if answerNumber == 1:
        return 'It is certain'
    elif answerNumber == 2:
        return 'Ask again later'
    elif answerNumber == 3:
        return 'My reply is no'
    elif answerNumber == 4:
        return 'Yes'
    elif answerNumber == 5:
        return 'Very doubtful'
    else:
        return 'Number not accepted'


r = random.randint(1, 5)
fortune = getAnswer(r)
print(fortune)
```

We can pass return values as an argument to another function call. How could you shorten these three lines?

Return statement

- When creating a function using the **def** statement, you can specify what the return value should be with a **return** statement

```
def getAnswer(answerNumber):
    if answerNumber == 1:
        return 'It is certain'
    elif answerNumber == 2:
        return 'Ask again later'
    elif answerNumber == 3:
        return 'My reply is no'
    elif answerNumber == 4:
        return 'Yes'
    elif answerNumber == 5:
        return 'Very doubtful'
    else:
        return 'Number not accepted'


print(getAnswer(random.randint(1, 9)))
```

None value

- Represents the absence of a value.
- Other programming languages might call this value null, nil, or undefined.
- Behind the scenes, Python adds return **None** to the end of any function definition with no return statement

```
def hello(name):
    print(name)

spam = hello('Bob’)
print(spam)
# None
```

Keyword arguments

- Keyword arguments are identified by the keyword put before them in the function call.
- Keyword arguments are often used for optional parameters.

```
print('Hello’)
print('World’)
```

```
print('Hello', end=”)
print('World')
```

```
print('cats', 'dogs', 'mice’)
```

```
print('cats', 'dogs', 'mice', sep=',’)
```

Positional arguments vs keyword arguments

$$\texttt{print('cats',\ }\underbrace{\texttt{'dogs',\ 'mice',}}_{\text{Positional arguments}}\ \underbrace{\texttt{sep=',\ ',\ end\ =\ '.\textbackslash n'}}_{\text{Keyword arguments}}\texttt{)}$$

Keyword arguments

```
def fun sum(a, b):
      return a + b


sum(5, 10)                # As positional arguments
sum(a = 5, b = 10)        # As keyword arguments
sum(b = 10, a = 5)
```

Default arguments

- Default values indicate that the function argument will take that value if no argument value is passed during function call.

```
def student(firstname, lastname ='Mark', standard ='Fifth'):
     print(firstname, lastname, ‘is in', standard, 'grade')



student('John’)
```

Default arguments

- Default values indicate that the function argument will take that value if no argument value is passed during function call.

```
def student(firstname, lastname ='Mark', standard ='Fifth'):
     print(firstname, lastname, ’is in', standard, ’grade')



student('John’)                             # John Mark is in Fifth grade
student('John', 'Gates', 'Seventh’)
```

Default arguments

- Default values indicate that the function argument will take that value if no argument value is passed during function call.

```
def student(firstname, lastname ='Mark', standard ='Fifth'):
     print(firstname, lastname, 'is in', standard, 'grade')



student('John’)                               # John Mark is in Fifth grade
student('John', 'Gates', 'Seventh’)    # John Gates is in Seventh grade
student('John', 'Seventh’)
```

Default arguments

- Default values indicate that the function argument will take that value if no argument value is passed during function call.

```
def student(firstname, lastname ='Mark', standard ='Fifth'):
     print(firstname, lastname, 'is in', standard, 'grade')



student('John’)                                 # John Mark is in Fifth grade
student('John', 'Gates', 'Seventh’)    # John Gates is in Seventh grade
student('John', 'Seventh’)                 # John Seventh is in Fifth grade
student('John’, standard = ‘Seventh’)
```

Default arguments

- Default values indicate that the function argument will take that value if no argument value is passed during function call.

```
def student(firstname, lastname ='Mark', standard ='Fifth'):
     print(firstname, lastname, 'is in', standard, 'grade')



student('John’)                                # John Mark is in Fifth grade
student('John', 'Gates', 'Seventh’)    # John Gates is in Seventh grade
student('John', 'Seventh’)                 # John Seventh is in Fifth grade
student('John’, standard = ‘Seventh’)  # John Mark is in Seventh grade
```

Default arguments

- Default values indicate that the function argument will take that value if no argument value is passed during function call.

```
def student(firstname, lastname ='Mark', standard ='Fifth'):
     print(firstname, lastname, 'is in', standard, 'grade')


student()
```

Default arguments

- Default values indicate that the function argument will take that value if no argument value is passed during function call.

```
def student(firstname, lastname ='Mark', standard ='Fifth'):
     print(firstname, lastname, 'is in', standard, 'grade')

student()                                        # Error: required argument missing
student(firstname ='John', 'Seventh’)
```

Default arguments

- Default values indicate that the function argument will take that value if no argument value is passed during function call.

```
def student(firstname, lastname ='Mark', standard ='Fifth'):
     print(firstname, lastname, 'is in', standard, 'grade')


student()                                   # Error: required argument missing
student(firstname ='John', 'Seventh’)   # Error: non keyword argument
                                            # after a keyword argument
student(subject ='Maths’)
```

Default arguments

- Default values indicate that the function argument will take that value if no argument value is passed during function call.

```
def student(firstname, lastname ='Mark', standard ='Fifth'):
     print(firstname, lastname, 'is in', standard, 'grade')


student()                                    # Error: required argument missing
student(firstname ='John', 'Seventh’)   # Error: non keyword argument
                                             # after a keyword argument
student(subject ='Maths’)                    # Error: unknown keyword argument
```

Local and Global scope

- Parameters and variables that are assigned in a called function are said to exist in that function’s **local scope** >>> **Local variable**
- Variables that are assigned outside all functions are said to exist in the **global scope** >>> **Global variable**
- When a scope is destroyed, all the values stored in the scope’s variables are forgotten.
- A local scope can access global variables.
- Code in a function’s local scope cannot use variables in any other local scope.
- You can use the same name for different variables if they are in different scopes.
- While using global variables in small programs is fine, it is a bad habit to rely on global variables as your programs get larger and larger.

Local and Global scope

```
def spam():
    eggs = 'spam local’
    print(eggs)  # prints 'spam local’

def bacon():
    eggs = 'bacon local’
    print(eggs)  # prints 'bacon local’
    spam()
    print(eggs)  # prints 'bacon local’

eggs = 'global’
bacon()
print(eggs)  # prints 'global'
```

Global statement

- If you need to **modify** a global variable from within a function, use the **global** statement.

```
def spam():
    global eggs # In this function, eggs  refers to the global
                # variable, so don’t create a local variable with
                # this name.
    eggs = 'spam’

eggs = 'global’
spam()
print(eggs)
```

Global statement

- If you need to **modify** a global variable from within a function, use the **global** statement.

```
def spam():
    print(eggs)
    eggs = 'spam local’

eggs = 'global’
spam()
```

Global statement

- If you need to **modify** a global variable from within a function, use the **global** statement.

```
def spam():
    print(eggs)  # UnboundLocalError: local variable 'eggs’
                 # referenced before assignment
    eggs = 'spam local’

eggs = 'global’
spam()
```

Function attributes

```
def say_whee():
    say_whee.count += 1
    print("Whee!")

say_whee.count = 0
say_whee()
say_whee()

print(say_whee.count)
```

Function attributes

```
def add_exclamation(s):
    add_exclamation.some_attribute = 'Function attribute’
    print(s + '!’)

add_exclamation('burma’)
add_exclamation.another_attribute = "Another function attribute”
print(add_exclamation.some_attribute)
print(add_exclamation.another_attribute)
# burma!
# Function attribute
# Another function attribute
```

Exception handling

```
def spam(divideBy):
    return 42 / divideBy

print(spam(2))
print(spam(12))
print(spam(0))    # ZeroDivisionError: division by zero
print(spam(1))
```

Exception handling

- You want the program to detect errors, handle them, and then continue to run.
- Errors can be handled with **try** and **except** statements.
- The code that could potentially have an error is put in a try clause.
- The program execution moves to the start of a following except clause if an error happens.
- After running that code, the execution continues as normal.

Exception handling

```
def spam(divideBy):
    return 42 / divideBy

print(spam(2))
print(spam("Whee"))
print(spam(12))
print(spam(0))
print(spam(1))
```

```
def spam(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
        print('Error: Argument cannot be a 0.’)

print(spam(2))
print(spam("Whee"))
print(spam(12))
print(spam(0))
print(spam(1))
```

Exception handling

```
def spam(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
        print('Error: Argument cannot be a 0.’)
    except TypeError:
        print('Error: Argument should be an int or a float’)

print(spam(2))
print(spam("Whee"))
print(spam(12))
print(spam(0))
print(spam(1))
```