HSS 611 - Week 4: Functions, Modules, Exceptions

2023-09-18

Agenda

- Functions
  - Writing functions
  - Variable scope
  - Return statement
- Modules
  - Importing functions
- Handling exceptions

Good Programming

- We learned to write ad-hoc code
- If we repeatedly copy, paste, and modify similar code lines, it can be
  - Inefficient: (really) long lines of code
  - Problematic: prone to errors while editing

Good Programming

- Write more succinct code by writing **functions**
- Functions are reusable pieces of code
- Makes code more maintainable
- If you anticipate repeating the same or very similar code more than once, it may be worth writing a reusable function

Microwave

- Imagine you had to manufacture a new microwave for each use
    - That's what constantly copying lines of code is like
    - Very inefficient
- To use (not create) one, you don't need to always know how it internally works
    - Inside is a black box
    - We use functions written not jut by us but others

Functions

- Let's dissect:
  - Has a **name**
  - Has **arguments** / **parameters** (0 or more)
  - Has a **docstring** (optional but recommended)
  - Has a **body**
  - **Return**s something (not always though)

Functions

```python
def is_even(i):
    """
    Input: i, a positive integer
    Returns True if i is even, otherwise False
    """
    return i % 2 == 0
is_even(5)
```

```
False
```

Functions

- **`def`** is the keyword used to define the function
- Name of the function comes after `def`
    - In this case, **`is_even`** is the name
- Then, inside (), comes the **arguments / parameters**
    - In this case, `**i**` is the only argument / parameter

Functions

- The docstring, enclosed in `"""`, provides info on how to use the function to the end user
- The docstring can be called with **`help()`**

```
help(is_even)
```

```
Help on function is_even in module __main__:

is_even(i)
    Input: i, a positive integer
    Returns True if i is even, otherwise False
```

Functions

- The body contains the code to be executed when the function is invoked
- The function usually **returns** something
  - This is done with the **`return`** keyword
  - After **`return`** is invoked, the function is exited

```
def is_even(i):
    """
    Input: i, a positive integer
    Returns True if i is even, otherwise False
    """
    return i % 2 == 0
is_even(5)
```

```
False
```

Variable Scope

- Scope refers to the region of code where a variable can be accessed or modified
- Initially, we are in the global scope
- When a function is entered, a new (local or function) scope is created

Variable Scope

What will this print?

```
i = 3

def square(x):
    x =  x ** 2
    return x

z = square(i)
print(z)
```

Variable Scope

What will this print?

```
i = 3

def square(x):
    x =  x ** 2
    return x

z = square(i)
print(z)
```

```
9
```

Variable Scope

What about this?

```
x = 3

def square(x):
    x =  x ** 2
    return x

z = square(x)
print(x)
```

Variable Scope

Notice, the x became 9 in the function's scope; not globally

```
x = 3

def square(x):
    x =  x ** 2
    return x

z = square(x)
print(x)
```

```
3
```

Variable Scope

z **is** 9 though, because we assigned **`square(x)`**

```python
x = 3

def square(x):
    x =  x ** 2
    return x

z = square(x)
print(z)
```

```
9
```

Be careful with function scope

- Because x is not one of the arguments, looks for an x in the global environment

```
x = 5

def p(y):
  return x

p(777)
z = p(777)
print(z)
```

```
5
```

Again, be careful with function scope

```
x = 5

def p(x):
  x = x + 1
  return x

p(x)
```

```
6
```

- Note that the global x is still intact (the x in the function used only in that scope)

```
print(x)
```

```
5
```

Function with no return

- Functions without a return statement will return `None`

```
def say_hello(name):
    print('Hello, ' + name + '!')

var = say_hello('Linda')
```

```
Hello, Linda!
```

```
type(var)
```

```
NoneType
```

Function with no return

- See the difference

```
def say_hello2(name):
    greeting = 'Hello, ' + name + '!'
    return greeting

var2 = say_hello2('Linda')
print(var2)
type(var2)
```

```
Hello, Linda!
```

```
str
```

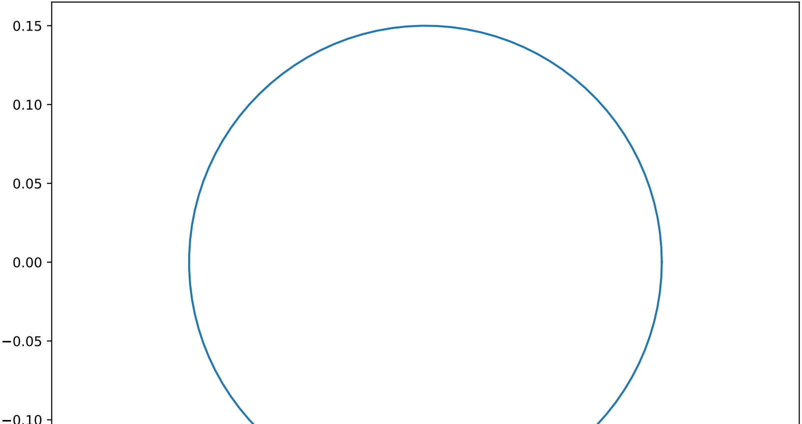
Function with no return

- Though it might sound pointless, it can be useful

```
import numpy as np

def plot_circle(diameter):

    # create an array of angles from 0 to 2*pi
    theta = np.linspace(0, 2 * np.pi, 100)

    # calculate the radius of the circle
    radius = diameter / 2.0

    # calculate the x and y coordinates of the circle
    x = radius * np.cos(theta)
    y = radius * np.sin(theta)

    # plot the circle
    plt.plot(x, y)

    # set aspect ratio to be equal, so the circle looks like a circle
    plt.axis('equal')

    # show the plot
    plt.show()
```

Fucntion with no return

```
import numpy as np
import matplotlib.pyplot as plt
circle_plot = plot_circle(0.3)
print(type(circle_plot))
```



More on return

- `return` can only be used inside of a function
- There can be multiple `returns` in a function
- Only one of them will be used each time function is invoked
- Once `return` is hit, function's scope is exited and nothing else in the function is run

Function with many return statements

- Will hit one of the three returns depending on `number`

```
def check_number(number):
    if number > 0:
        return "positive"
    elif number < 0:
        return "negative"
    else:
        return "zero"
```

```
check_number(7)
```

```
'positive'
```

Another example

- What type of an object will this function return?

```python
def calculate_rectangle_properties(length, width):
    if length <= 0 or width <= 0:
      return None, None, None
    perimeter = 2 * (length + width)
    area = length * width
    diagonal = (length ** 2 + width ** 2) ** 0.5

    return perimeter, area, diagonal
```

```python
calculate_rectangle_properties(1, 0)
```

```
(None, None, None)
```

Python Modules

- Python modules are files (.py) that (mainly) contain function definitions
- They allow us to organize, distribute code; to share and reuse others' code too
- Keep code coherent and self-contained
- One can `import` modules or some functions from modules

Example

- math_operations module saved as math_operations.py

```
# math_operations.py

def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

Example

- Import whole module:

```
import sys
sys.path.append('/Users/taegyoon/Desktop') # add directory
import math_operations
```

- To use a function from the module, need to refer to what we imported

```
math_operations.add(3,5)
```

```
8
```

Example

- We could also import the specific function

```
from math_operations import subtract
```

- Then use the imported function directly

```
subtract(4,10)
```

```
-6
```

Example

- Can rename function while importing

```
from math_operations import subtract as sub
```

- Then use with that name

```
sub(1,2)
```

```
-1
```

Example

- There are many modules in the Standard Library and external libraries that one can and should use!
- Standard library example

```
from datetime import date
today = date.today()
print("Today's date:", today)
```

```
Today's date: 2023-09-18
```

Exceptions

- When there is something wrong with **syntax**, Python will throw an error (**syntax** error)

```
print("error")) # SyntaxError: unmatched ')'
```

- But even without **syntax** error, there can be `Exceptions`, which are errors during execution

```
result = 10 / 0 # ZeroDivisionError: division by zero
```

Handling Exceptions

- If your code can encounter an exception, you can handle that using try / except
- Try this out

```
del x
try:
  print(x) # will not work
except:
  print("An exception occured")
```

Handling Exceptions

- Multiple exceptions are possible
- With the error type specified, the except block runs only when an error of the specified type occurs

```
try:
  print(k)
except NameError: # this is executed only for NameError
  print("Variable is not defined")
except:
  print("Something else went wrong")
```

Handling Exceptions

- Multiple exceptions are possible
- With the error type specified, the except block runs only when an error of the specified type occurs

```python
try:
  print(k)
except NameError: # this is executed only for NameError
  print("Variable is not defined")
except:
  print("Something else went wrong")
```

```
Variable is not defined
```

Handling Exceptions

- Similarly

```
l = [1,2]
try:
    4/0
    print(l[3])
except ZeroDivisionError as e: # e contains details
    print(e) # the default error message
except IndexError as e:
    print(e)
```

Handling Exceptions

- Note

```
l = [1,2]
try:
    4/0
    print(l[3])
except ZeroDivisionError as e:
    print(e)
except IndexError as e:
    print(e)
```

Handling Exceptions

- Even

```
l = [1,2]
try:
  4/0
  print(l[3])
except (ZeroDivisionError, IndexError) as e:
  print(e)

try:
  print(l[3])
  4/0
except (ZeroDivisionError, IndexError) as e:
  print(e)
```

```
division by zero
list index out of range
```

Handling Exceptions

- If you know the exact source of error or the scope of potential errors in advance, you can also use `if` to prevent (rather than handle) them
- See discussions here

```
try:
  print(x) # will not work
except:
  print("An exception occured")

if 'x' in globals():
    print(x)
else:
    print("An exception occurred")
```

```
5
5
```

Handling Exceptions

- We may want to raise (or throw) reasonable exceptions

```python
def calculate_rectangle_properties(length, width):

    if length <= 0 or width <= 0:
        raise Exception("Dimensions need to be positive.")

    perimeter = 2 * (length + width)
    area = length * width
    diagonal = (length ** 2 + width ** 2) ** 0.5

    return perimeter, area, diagonal
```

Handling Exceptions

- This can be used to help end users use our function

```
def calculate_rectangle_properties(length, width):

    if length <= 0 or width <= 0:
        raise Exception("Dimensions need to be positive.")

    perimeter = 2 * (length + width)
    area = length * width
    diagonal = (length ** 2 + width ** 2) ** 0.5

    return perimeter, area, diagonal
```

Handling Exceptions

- Try

```
calculate_rectangle_properties(0, 3)
calculate_rectangle_properties(3, 3)
```

Handling Exceptions

- We can do this as well

```
try:
  a, b, c = calculate_rectangle_properties(3, 3)
except:
  print("something went wrong")
```

Reading and Writing Text Files

- Write a regular text file

```
file = open('mytext.txt', 'w')
file.write('Hi, there!.\nThis is my text file')
file.close()
```

Reading and Writing Text Files

- Read it back in

```
file = open('mytext.txt', 'r')
content = file.read()
file.close()
```

Reading and Writing Text Files

- Check content

```
print(content)
```

```
Hi, there!.
This is my text file
```

Reading and Writing Text Files

- `open()`, `read()` / `write()`, `close()` is a bit cumbersome
- The more preferred syntax: **`with()`**

```python
with open('mytext.txt', 'w') as file:
    file.write('Comment 1\nComment 2\nComment 3')
```

Reading and Writing Text Files

- readlines()

```python
with open('mytext.txt', 'r') as file:
    content = file.readlines()
print(content)
type(content)
```

```
['Comment 1\n', 'Comment 2\n', 'Comment 3']
```

```
list
```

Reading and Writing Text Files

- read()

```
with open('mytext.txt', 'r') as file:
    content = file.read()
print(content)
type(content)
```

```
Comment 1
Comment 2
Comment 3
```

```
str
```