Function Scope

Guttag Chapter 4

Prof. Graber

Goals

- Review Function Definition and Calling
- Define **Formal** vs **Actual** Parameters
- Define Keyword Arguments, Positional Arguments, Default Values
- Define Scope
- Unravel Point of Execution
- Define Higher-Order Functions

Review

What is a Function

- Functions are like cooking recipes
- Set of instructions that have a name
  - Cake, Cookies
- Functions contain steps for how to **do** something with certain ingredients, called **parameters**

Function Definitions

```
def prime_test_exhaustive(x: int) -> bool:
  """Determine if positive input is prime, returning true for prime and false for not prime.
  if x <= 1:
    return False
  if x == 2:
```

```
6       return True
7     for guess in range(2, x, 1):
8       if x % guess == 0:
9         return False
10    return True
```

- line 1 is the function signature
  - `def` python keyword that starts off the function
  - `prime_test_exhaustive` name of the function
  - `(x: int)` formal parameters for the function inside ()
  - `x` name of the parameter, to be used inside!
  - `: int` input type annotation for x
  - `-> bool` output type annotation
  - `:` end of the function signature

Function Definitions

```
1   def prime_test_exhaustive(x: int) -> bool:
2     """Determine if positive input is prime, returning true for prime and false for not prime.
3     if x <= 1:
4       return False
5     if x == 2:
6       return True
7     for guess in range(2, x, 1):
8       if x % guess == 0:
9         return False
10    return True
```

- Everything after the function signature that is IN the function must be indented
- line 2 `"""Determine if positive input is prime, returning true for prime and false for not prime."""` docstring
- lines 3-10 contain code that makes use of the formal parameter x
- `return` keyword that ends the function and makes any following values available outside of the function
  - `return True` would make True available outside of the function

Function Calling with Literal

```
result = prime_test_exhaustive(97)
```

- read this expression from right to left
- input parameter is `97` inside ( )
  - `97` gets assigned to the function's formal parameter
- `prime_test_exhaustive` is the name of the function
- `=` assignment operator ready to assign anything that is returned out of the function
- `result` is the variable whose value will become whatever is returned out of the function
- n.b. this code does not print anything

Function Calling with Variable

```
number = 97
result = prime_test_exhaustive(number)
print(result)
```

- line 1 assigns 97 to the variable named `number`
- line 2 input parameter is `number` inside ( )
  - `number` gets assigned to the function's formal parameter
- `prime_test_exhaustive` is the name of the function
- `=` assignment operator ready to assign anything that is returned out of the function
- `result` is the variable whose value will become whatever is returned out of the function
- `result` is printed

Function Calling Wrong

```
result = prime_test_exhaustive
```

- first one refers only the memory location of the function!

. . .

```
prime_test_exhaustive()
```

- second one calls the function using `()` but no parameter is supplied!

. . .

```
prime_test_exhaustive(int)
```

- third one supplies a type instead of a value

Review Summary

- **Functions** are a set of instructions
- **Functions** contains return statements
- The function is executed when it is **called**
- The function **call** supplies actual parameters with real values
- The programmer must add logic to store the return value after the function is **called**.
- Printing should occur after a function **call**

Formal vs Actual Parameters

Formal Parameters

. . .

```
def prime_test_exhaustive(x: int) -> bool:
  """Determine if positive input is prime, returning true for prime and false for not prime.
  if x <= 1:
    return False
  if x == 2:
    return True
  for guess in range(2, x, 1):
    if x % guess == 0:
      return False
  return True

number = 97
result = prime_test_exhaustive(number)
print(result)
```

```
True
```

- the formal parameter is what is supplied in the **function signature** inside ()
  - `x`
- the formal parameter is the variable used within the function instructions, representing an abstract value
- it helps... **form** ....the function...

Actual Parameters

. . .

```
def prime_test_exhaustive(x: int) -> bool:
  """Determine if positive input is prime, returning true for prime and false for not prime.
  if x <= 1:
    return False
  if x == 2:
    return True
  for guess in range(2, x, 1):
    if x % guess == 0:
      return False
  return True

number = 97
result = prime_test_exhaustive(number)
print(result)
```

```
True
```

- the actual parameter is supplied in the **function call**
  - `number`
- the actual parameter has an **actual** value and is not abstract

Formal and Actual Parameters are different

. . .

```
def prime_test_exhaustive(x: int) -> bool:
  """Determine if positive input is prime, returning true for prime and false for not prime.
  if x <= 1:
    return False
  if x == 2:
    return True
  for guess in range(2, x, 1):
    if x % guess == 0:
      return False
  return True

number = 97
result = prime_test_exhaustive(number)
print(result)
```

```
True
```

- the name of the actual parameter does NOT have to match the name of the formal parameter!
- the value of the actual parameter gets assigned to the formal parameter when the function is called
- try it, assigning 101 to number!

Formal and Actual Parameters are different

. . .

This code will crash

```
def prime_test_exhaustive(x: int) -> bool:
  """Determine if positive input is prime, returning true for prime and false for not prime.
  if x <= 1:
    return False
  if x == 2:
    return True
  for guess in range(2, x, 1):
    if x % guess == 0:
      return False
  return True

result = prime_test_exhaustive(x)
print(result)
```

- the formal parameter only exists inside the function
- trying to use the formal parameter outside of the function will not work because it does not exist outside of the function

Formal and Actual Parameters are different

. . .

```
def prime_test_exhaustive(x: int) -> bool:
  """Determine if positive input is prime, returning true for prime and false for not prime.
  if x <= 1:
    return False
  if x == 2:
    return True
  for guess in range(2, x, 1):
    if x % guess == 0:
      return False
  return True

x = 97
result = prime_test_exhaustive(x)
print(result)
```

```
True
```

- the name of the actual parameter does NOT have to match the name of the formal parameter!
- but it could…they are completely different variables
- the value of the actual parameter gets assigned to the formal parameter when the function is called
- try it, assigning 101 to the global x!

Formal vs Actual Parameters Summary

- the formal parameter helps form the function, and is abstract
- the actual parameter has an actual value
- even if they have the same name, they are different variables!
- the actual parameter gets assigned to the formal parameter when the function is called

Keyword Arguments, Positional Arguments, Default Values

Function Calling with Positional Arguments

. . .

```
from typing import Union

def divide(denominator: int, numerator: int, decimal_flag: bool) -> Union[float, str]:
    if decimal_flag:
        return numerator/denominator
    return f"{numerator//denominator} R {numerator % denominator}"

result_decimal = divide(15, 31, True)
print(result_decimal)

result_non_decimal = divide(15, 31, False)
print(result_non_decimal)

result_misleading = divide(True, 15, 31)
print(result_misleading)
```

```
2.066666666666667
2 R 1
15.0
```

- Order of parameters matters!
- The python interpreter assumes that what you supply first should be assigned to the first formal parameter, etc.

Function Calling with Keyword

- Order of parameters does not matter when **keywords** are used
- keywords help label what is what
  - "...which one was the denominator again....?"

Function Calling with Keyword

```
from typing import Union

def divide(denominator: int, numerator: int, decimal_flag: bool) -> Union[float, str]:
    if decimal_flag:
        return numerator/denominator
    return f"{numerator//denominator} R {numerator % denominator}"

result_decimal = divide(decimal_flag=True, denominator=15, numerator=31)
print(result_decimal)

result_misleading = divide(True, 15, 31)
print(result_misleading)
```

```
2.066666666666667
15.0
```

- line 8 passes Keyword arguments `decimal_flag=True, numerator=16, denominator=30` inside ( )
  - `decimal_flag` is a formal parameter that was used in the function definition
  - `numerator` is a formal parameter that was used in the function definition
  - `denominator` is a formal parameter that was used in the function definition
  - `True` gets assigned to `decimal_flag`
  - `16` gets assigned to `numerator`
  - `30` gest assigned to `denominator`

Functions with Default Values

```
from typing import Union

def divide(denominator: int, numerator: int, decimal_flag: bool = False) -> Union[float, str]
    if decimal_flag:
        return numerator/denominator
    return f"{numerator//denominator} R {numerator % denominator}"

result_keyword = divide(numerator=31, denominator=15)
print(result_keyword)

result_positional = divide(15, 31)
print(result_positional)
```

```
2 R 1
2 R 1
```

- line 8 and 11 do not specify the `decimal_flag`
  - `decimal_flag` has a default value of `False` in the function signature
  - Unless otherwise specified, the default will be used

Keyword Arguments, Positional Arguments, Default Values Summary

- Positional Arguments must be supplied matching the order of the formal parameters in the function signature
- Keyword Arguments can be out of order
- Parameters with Default Values do not need to be supplied

Scope

Global Scope

- variables that are not inside of functions are **global**
- this means they can be accessed anywhere in the code, even inside of functions

. . .

```
this_is_global = "hello"
```

Function Scope

- variables that are created inside of functions are **local** to the function, NOT global
- this means they cannot be accessed outside of the function

. . .

```
this_is_global = "hello"

def my_function() -> str:
    this_is_local_to_function = "hello"
    if this_is_local_to_function == this_is_global:
        return "the global variable was accessible in the function!"

function_result = my_function()
print(f"function_result is {function_result}")
```

```
function_result is the global variable was accessible in the function!
```

Function Scope

. . .

This code crashes

. . .

```
this_is_global = "hello"

def my_function() -> str:
    this_is_local_to_function = "hello"
    if this_is_local_to_function == this_is_global:
        return "the global variable was accessible in the function!"

function_result = my_function()
print(f"function_result is {function_result}")

print(this_is_local_to_function)
```

. . .

- try it!

Function Scope

- When a function is called, it is executed in a temporary isolated environment called a **stack frame**
- Variables that are defined inside a function are only known within the stack frame!
- After a function is done running, the stack frame is **popped**, i.e. it disappears
  - it's variables will not be known to the outside world!
- The best way to make a function variable available to the outside world is to `return` it

Function Scope

Question

- if the stack frame of an executing function forms a barrier with the outside world, how does it know what the actual parameter is?

Answer

- the values of the actual parameters are assigned to the formal parameters by the python interpreter!
- you can think of it like this: `formal_param = actual_param`
  - the binding is done by assignment with the `=`

Scope Summary

- Variables not inside a function are in the global scope
- Variables defined inside a function are in the function scope
- The python interpreter knows what scope it is in as it executes code!
- Functions are executed in their own stack frame

Unravelling Point of Execution

Tracing Execution

- Tracing the path that the interpreter takes though code is informative for understanding why scope matters.
- The interpreter goes line by line to things that it can execute!
- It will jump around when it encounters loops and conditional branching
- Functions are NOT executed until the function is **called**

Tracing Execution

. . .

```
def prime_test_exhaustive(x: int) -> bool:
  """Determine if positive input is prime, returning true for prime and false for not prime.
  if x <= 1:
    return False
  if x == 2:
    return True
  for guess in range(2, x, 1):
    if x % guess == 0:
      return False
  return True

x = 4
result = prime_test_exhaustive(x)
print(result)
```

```
False
```

. . .

- TODO: Trace the code below

step	line #	scope	var names	values
1	?	?	?	?
2	?	?	?	?
etc	?	?	?	?

Tracing Execution

```
def prime_test_exhaustive(x: int) -> bool:
  """Determine if positive input is prime, returning true for prime and false for not prime.
  if x <= 1:
    return False
  if x == 2:
    return True
  for guess in range(2, x, 1):
    if x % guess == 0:
      return False
  return True
```

```
x = 4
result = prime_test_exhaustive(x)
print(result)
```

```
False
```

step	line #	scope	var names	values
1	12	G	x	4
2	13	G	x	4
3	3	L	local x	4
4	5	L	local x	4
5	7	L	local x, local guess	4, 2
6	8	L	local x, local guess	4, 2
7	9	L	local x, local guess	4, 2
8	13	G	x, result	4, False
9	14	G	x, result	4, False

Tracing Execution 2

- TODO: Trace the code below

. . .

```
def something_else(x: int) -> bool:
  """Create a situation for scope analysis."""
  x += 1
  return x + 1


x = 4
result = something_else(x + 1)
print(f"result is {result}")
print(f"x is {x}")
```

```
result is 7
x is 4
```

step	line #	scope	var names	values
1	?	?	?	?
2	?	?	?	?
etc	?	?	?	?

Guttag Example

```
# how many lines does this code print, and what is the last line printed?

def f(x):
    def g():
        x = 'abc'
        print(f"x = {x}")
        return
    def h():
         z = x
         print(f"z = {z}")
         return
    x = x + 1
    print(f"x = {x}")
    h()
    g()
    print(f"x = {x}")
    return g

x = 3
z = f(x)
print(f"x = {x}")
print(f"z = {z}")
z()
```

Execution Summary

- The interpreter goes line by line to things that it can execute!
- Functions are NOT executed until the function is **called**
- variables with the same name in different scopes are **different** variables!
- Refer to Guttag Chapter 4

Higher Order Functions

Regular Function

. . .

```
print(abs(-10)) # works
print(abs([15, -1,-3, 19, -25, -3.1])) # crashes
```

- **abs** is a built-in python function!
- it can only work on one number at a time
- try it!

Higher Order Function

. . .

```
from typing import List, Callable

def apply_to_each(input_list: List, function: Callable) -> List:
    output_list = []
    for i in input_list:
        new_i = function(i)
        output_list.append(new_i)
    return output_list

my_list = [15, -1,-3, 19, -25, -3.1]
result = apply_to_each(my_list, abs)
print(result)
```

```
[15, 1, 3, 19, 25, 3.1]
```

- line 11, abs is passed in as a parameter to **apply_to_each**
- note that ONLY THE FUNCTION NAME was the actual parameter
- apply_to_each is a **higher-order function** because it takes a function as a parameter

Overall Summary

- Reviewed Function Definition and Calling
- Defined **Formal** vs **Actual** Parameters
- Defined Keyword Arguments, Positional Arguments, Default Values
- Defined Scope
- Unravelled Point of Execution
- Defined Higher-Order Functions

End