LECTURE 3

DOHYUNG KIM

WHAT IS DISCUSSED IN THE LAST CLASS

- Object and Expression
  - Built-in types, constants, functions
  - Built-in operators
    - Operator order
    - Floating point errors
    - Short-cut evaluation
    - Type-checking

TODAY, WE WILL LEARN ABOUT

- Functions in python
  - Function definition/call, return
  - Local variables
  - Global variables
  - Parameters and arguments
  - Function as an object

TODAY, WE WILL LEARN ABOUT

- Functions in python
  - Function definition/call, return
  - Local variables
  - Global variables
  - Parameters and arguments
  - Function as an object

FUNCTION

- A **function definition** specifies the name of a new function and the sequence of statements that are executed when the function is called

```
def print_message():
  print("Python is fantastic")
  print("Programming is fun")
  print("--------------------")
```

indentation

```
def repeat_message():
  print_message()
  print_message()

repeat_message()
```

VOCABULARY

- **def** is used to define a function
- **func** is the function name
- **y, z** are called parameters

- Call function
- 1,2 are called arguments

- **Global** variable

main.py saved

```python
x = 10
def func(y, z):
    val = 1
    res = x + y + z + val
    return res
print(func(1, 2))
print(val)
```

```
14
Traceback (most recent call last):
  File "main.py", line 7, in <module>
    print(val)
NameError: name 'val' is not defined
>
```

- **return** is used to pass the result to the point where the function is called

- **Local** variable

- Expression (a part of statement) vs. Statement

RETURN STATEMENT

```
def isPositive(x):
  return (x > 0)

print(isPositive(5))
print(isPositive(0))

def isNegative(x):
  print("Hello!")
  return (x < 0)
  print("Goodbye!")

print(isNegative(5))

def f(x):
  result = x + 42

print(f(5))
```

PRACTICE

- What would be the result?

```
def cubed(x):
  print(x**3)

cubed(2)
print(cubed(3))
print(2*cubed(5))
```

PARAMETER AND RETURN TYPE

```python
def hypotenuse(a, b):
  return ((a**2) + (b**2))**0.5

print(hypotenuse(3, 4))
print("--------------------")

def xor(b1, b2):
  return ((b1 and (not b2)) or (b2 and (not b1)))

print(xor(True,  True))
print(xor(True,  False))
print(xor(False, True))
print(xor(False, False))
print("--------------------")

def isPositive(n):
  return (n > 0)

print(isPositive(10))
print(isPositive(-1.234))
```

```
5.0
--------------------
False
True
True
False
--------------------
True
False
>
```

FUNCTION COMPOSITION

```
def f(w):
  return 10*w

def g(x, y):
  return f(3*x) + y

def h(z):
  return f(g(z, f(z+1)))

print(h(1))
```

```
500
```

—> h(1)
—> f(z+1) returns 20
—> g(1, 20)
  —> f(3) returns 30
  —> return 50
—> f(50) returns 500
print(500)

TODAY, WE WILL LEARN ABOUT

- Functions in python
  - Function definition/call, return
  - Local variables
  - Global variables
  - Parameters and arguments
  - Function as an object

LOCAL VARIABLES

- A function to evaluate the quadratic function $ax^2 + bx + c$:

```
def quadratic(a,b,c,x):
  quad_term = a*x**2
  lin_term  = b*x
  return quad_term + lin_term + c

print(quadratic(1,2,3,4))
```

- The variables **quad_term** and **lin_term** exist only during the execution of the function quadratic. They are called **local variable**

- A **function's parameters** are also local variable. The only difference is that when the function is called, they are initialized from the function argument

LOCAL VARIABLES

```python
a = "Letter a"

def f(a):
  print("A =", a)

def g():
  a = 7
  f(a + 1)
  print("A =", a)

print("A =", a)
f(3.14)
print("A =", a)
g()
print("A =", a)
```

```
A = Letter a
A = 3.14
A = Letter a
A = 8
A = 7
A = Letter a
```

* Notes

- The values of a inside function f() and g() are different from "Letter a"
- They are valid until the function is terminated

WHY LOCAL VARIABLES?

- To use the function quadratic, you only want to remember this

```
def quadratic(a,b,c,x):
    #implemented somehow
```

- Some variables are used only in a certain part of the program and we don't need to keep track of them

- **Modularization** means that software consists of parts that are developed and tested separately. To use a part, you don't need to understand how it is implemented

PRACTICE

- What would be the results of these example codes? Why?

```python
def swap(a, b):
  a, b = b, a

x, y = 123, 456
swap(x, y)
print(x, y)
```

```python
def f(x):
  print("In f, x =", x)
  x += 7
  return round(x / 3)

def g(x):
  x *= 10
  return 2 * f(x)

def h(x):
  x += 3
  return f(x+4) + g(x)

print(h(f(1)))
```

TODAY, WE WILL LEARN ABOUT

- Functions in python
  - Function definition/call, return
  - Local variables
  - Global variables
  - Parameters and arguments
  - Function as an object

GLOBAL VARIABLES

- Variables **defined outside of a function** are called global variables
- Generally, it is **not recommended to use global variables**. However, you need to understand how it works

```
g = 100

def f(x):
    return x + g

print(f(5))
print(f(6))
print(g)
```

LOCAL AND GLOBAL

- If a variable is **only used** inside a function, it is **global**
- If a variable is **assigned** in a function, it is **local**

```
val=3

def f1():
  return 3*val + 5

def f2():
  val = 5;
  return 3*val + 5
```

GLOBAL VARIABLES

- If To modify a global variable within a function, you must use the keyword **global** before the variable

```
g = 100

def f(x):
  global g
  g += 1
  return x + g

print(f(5))
print(f(6))
print(g)
```

PRACTICE

- What would be the result?

```
a = 17
def test():
  print(a)
  a = 13
  print(a)

test()
```

PRACTICE

- What would be the result?

```python
a = 17
def test():
  print(a)
  a = 13
  print(a)

test()
```

Error!!

```
Traceback (most recent call last):
  File "main.py", line 7, in <module>
    test()
  File "main.py", line 3, in test
    print(a)
UnboundLocalError: local variable 'a' referenced before assignment
>
```

* note : **a** is a **local variable** in the test function because of the assignment, but has no value inside when the first print statement is executed

TODAY, WE WILL LEARN ABOUT

- Functions in python
  - Function definition/call, return
  - Local variables
  - Global variables
  - Parameters and arguments
  - Function as an object

DEFAULT PARAMETER

- Sometimes, a function has a parameter that has a default value

```
def f(x, y=10):
  return x + y

print(f(5))
print(f(5,1))
```

- **Default parameters have to locate after normal parameters**

```
def f(x=10, y):
  return x + y
```

```
  File "main.py", line 1
    def f(x=10, y):
          ^
SyntaxError: non-default argument follows default argument
>
```

DEFAULT PARAMETER

- What would be printed out? 5? 6?

```
val = 5

def f(x = val):
  print(x)

val = 6
f()
```

DEFAULT PARAMETER

- What would be printed out? 5? 6?

```
val = 5

def f(x = val):
  print(x)

val = 6
f()
```

- **The default values are evaluated at the point of function definition**

NAMED PARAMETER

- We can include the name of the parameter in the function call to make the code clearer the order of argument does not matter

```
def avg(first, second=10, third=100):
  return (first+second+third)/3

print(avg(1, third=10))
print(avg(first=1, third=10))
print(avg(third=10, first=1))
```

```
7.0
7.0
7.0
>
```

- What if the following statement is executed?

```
print(avg(third=3, 1))
```

```
  File "main.py", line 7
    print(avg(third=3, 1))
                         ^
SyntaxError: positional argument follows keyword argument
>
```

VARIABLE-LENGTH ARGUMENTS

- Python allows to pass a variable number of arguments to a function
- An asterisk (*) is placed before the variable name that will hold the values of multiple arguments (non-keyword)

```
def myFun(*argv):
  res = ""
  for arg in argv:
    res += arg
  print(res)

myFun('Hello')
myFun('Hello', 'Welcome')
myFun('Hello', 'Welcome', 'to', 'Python', 'Programming', 'Class')
```

```
Hello
HelloWelcome
HelloWelcometoPythonProgrammingClass
>
```

- The details of the above code will be covered in the later class

TODAY, WE WILL LEARN ABOUT

- Functions in python
  - Function definition/call, return
  - Local variables
  - Global variables
  - Parameters and arguments
  - Function as an object

FUNCTIONS ARE OBJECTS

- A function is also an object

```
import math

def f(x):
  return math.sin(x/3 + math.pi/4)

print(f)
print(type(f))
```

```
<function f at 0x7fbcc239a1f0>
<class 'function'>
```

FUNCTIONS ARE OBJECTS

- You can use a function as an argument

```
def shout(text):
  return text.upper()

def whisper(text):
  return text.lower()

def greet(func):
    # storing the function in a variable
  greeting = func("Hello, I am created by a function passed by argument.")
  print(greeting)

greet(shout)
greet(whisper)
```

```
HI, I AM CREATED BY A FUNCTION PASSED AS AN ARGUMENT.
hi, i am created by a function passed as an argument.
```

FUNCTIONS RETURNING A FUNCTION OBJECT

- Let's define the function that returns a quadratic function f defined by

$f(x) = ax^2 + bx + c$

```
def quadratic(a,b,c):
  def f(x):
    quad_term = a*x**2
    lin_term  = b*x
    return quad_term + lin_term + c
  return f

print(quadratic(1,0,1)(2))
```

QUESTION?