

Modules in Python

Decision Control: Conditionals

CS 8: Introduction to Computer Science, Spring 2019
Lecture #5

Ziad Matni, Ph.D.
Dept. of Computer Science, UCSB

Administrative

- **Hw02 – due today**
- Hw03 – due next week
- Lab01 – due on Sunday by midnight (11:59 pm) on
Gradescope!
- You can check old homework on GradeScope

Lecture Outline

- `print()` vs. `return`
- Modules in Python
- Boolean Operations
- Conditionals *i.e.* Decision Control

Print vs. Return

What's the difference between these 2 functions?

```
def return_dbl( x ):  
    return x*2
```

```
def print_dbl( x ):  
    print(x*2)
```

What happens if I do this *in IDLE*?

```
>>> a = 13  
>>> return_dbl(a)  
>>> print_dbl(a)
```

What happens if I do this *in a program*?

```
a = 13  
return_dbl(a)  
print_dbl(a)
```

Print vs. Return

```
def return dbl( x ):  
    return x*2
```

```
def print dbl( x ):  
    print(x*2)
```

What happens if I do this in IDLE?

```
>>> a = 13  
>>> print(return dbl(a))  
>>> print(print dbl(a))
```

Would it be different in a program?

Printing vs. returning the output can lead to very different behaviors!!!!

Modules in Python

- We can collect a bunch of Python functions and save them together in a file
 - Often called a *module* or *library*
 - It's a good way to organize functions that “go together” for a specific purpose
 - The module can also have other code related to these functions
- We can then “*import*” that file over and use the functions for ourselves in our own program files

Modules in Python

- Recall that Python is Open-Source Software
 - What does that mean?
- A lot of modules have been created by various people and then put up for anyone to use
 - Example: modules to do advanced statistical analysis, to help solve linear algebra probs, help solve ODEs, etc...
 - Google “**popular python libraries**”!

Reminder: How Can We Use Functions?

- Once we *define* them, we can *call* them:
 - In a program (in same file where they're defined)
 - In the IDLE Python shell
 - Example: MyFunction(5, 6, 7)
- We can also call them:
 - From another file altogether
 - (to test them from) using Pytest

Creating a Module

Consider a couple of functions that we wrote and that we'd like to use in other Python programs:

```
# Doubling function
def dbl(x):
    return 2*x

# Halving function
def half(x):
    return x/2
```

Save them in a file.
Let's call it **PinkFloyd.py**
*(just happened to be what I was listening to,
but I'm sure you can come up with a better name)*

Using Modules in Other Files

Inside *another* file, we can still use the functions we defined and saved in **PinkFloyd.py**, like this:

Inside SomeOtherFile.py

```
# We want to use those functions in PinkFloyd.py
import PinkFloyd
# from PinkFloyd import *      # another way to do this

print("Inside PinkFloyd.py")

print(PinkFloyd dbl(5))
print(PinkFloyd dbl("UCSB"))
print(PinkFloyd dbl([1, 2, 3]))

print(PinkFloyd half(42))
```

Inside PinkFloyd.py

```
# Doubling function
def dbl(x):
    return 2*x

# Halving function
def half(x):
    return x/2
```

Conditional Execution

What if my module (e.g. PinkFloyd.py) has instructions in it *other* than the function defs?

```
# Doubling function
def dbl(x):
    return 2*x

# Halving function
def half(x):
    return x/2

print("I'm inside PinkFloyd.py")
print(dbl(82.12))
```

When you import the module, you may
NOT want this stuff to execute

Conditional Execution

What if my module (e.g. PinkFloyd.py) has instructions in it *other* than the function defs?

```
# Doubling function
def dbl(x):
    return 2*x

# Halving function
def half(x):
    return x/2

if __name__ == "__main__":
    print("I'm inside PinkFloyd.py")
    print(dbl(82.12))
```

Add this conditional statement.

Now the 2 print statements are only executed when we run PinkFloyd.py, not when we import it

Testing

```
#test dbl.py
import pytest
from PinkFloyd import dbl

def test dbl_1():
    assert dbl(0) == 0
def test dbl_2():
    assert dbl(2) == 4
def test dbl_3():
    assert dbl("UCSB") == "UCSBUCSB"
```

Run these tests from the unix command line:

```
$python3 -m pytest test dbl.py
```

Relational Operators

- To check if two objects are equal, we use:
== operator
- We can check other types of relationships between two objects:

< *Less than*

<= *Less than or equal to*

> *Greater than*

>= *Greater than or equal to*

!= *Not equal to*

Q: All of these produce what kind of data type?

A: Boolean (i.e. True or False)

What is the Output of the **print()** statement?

```
a = 3  
b = (a != 3)  
print(b)
```

- A. True
- B. False 
- C. 3
- D. Syntax error

Logic Operators

- Logic AND
 - Use in Python: `x and y`
 - Presents a **True** output if both `x` and `y` are True
- Logic OR
 - Use in Python: `x or y`
 - Presents a **True** output if either `x` and `y` are True
- Logic NOT
 - Use in Python: `not x`
 - Presents a **True** output if `x` is **False** (and vice-versa)

Exercise 1

What is the output of these print() statements?

x = True

y = False

z = True

print(x and y) ← False

print(x and not y) ← True

print((x and y) or z) ← True

Exercise 2

What is the output of these print() statements?

a = 3

b = 4

c = 5

print(a == 3 and b == 4) ← True
print(not (b < 2) and not (a != 5)) ← False
print((a > 0 or b == 0) and (c <= 5)) ← True

Exercise 3

- Write a function, **CheckNegInt()** that takes in one argument **x** and returns **True** if 2 conditions are met:
 - That **x** is an integer
 - That **x** is a negative number

```
def CheckNegInt(x):  
    c = False  
    if (type(x) == int) and (x < 0):  
        c = True  
    return c
```

Controlling the Flow of a Program

- Programs will often need to make decisions on what to continue doing
 - Like coming to a fork in the road...
- We present the algorithm/program with a *conditional statement* (a.k.a *if-then-else*)

Conditional Statements: **if** and **else**

Typical Example (NOTE THE INDENTS!!!)

```
x = int(input("Enter any integer: "))

if x >= 0:
    print ("You entered a positive number!")
    print ("Or it could be zero!")

else:
    print ("You entered a negative number!")
```

Let's try it out!

Conditional Statements: if and else

The syntax in Python is:

```
if conditional_statement :  
    statement 1  
    statement 2  
  
    ...  
elif conditional_statement :  
    else statements  
elif conditional_statement :  
    more else statements  
else:  
    default else statements
```

Let's try it out!

```
a = int(input("Enter a number: "))  
# The above line makes the program  
# ask the user for a direct input  
# into an integer. More on this later.  
  
if (a < 5):  
    print("It's less than five!")  
  
elif (a > 5):  
    print("It's more than five!!!")  
  
else:  
    print("It's equal to five!!!!")
```

Conditional Statements ARE Boolean Values

```
a = int(input("Enter a number: "))
# The above line makes the program
# ask the user for a direct input
# into an integer. More on this later.

if (a < 5):
    print("It's less than five!")

elif (a > 5):
    print("It's more than five!!!")

else:
    print("It's equal to five!!!!")
```

*Boolean statements
(they're either TRUE or FALSE)*

Nested If-Else Statements

Think of If-Else as a way to describe “logical branching”

```
a = int(input("What is the cost of item X? "))  
b = int(input("Enter (0) for not available, (1) for available "))
```

```
if (b == 0):  
    print("It doesn't matter what it costs: it's not available!")  
else:  
    if (a >= 100):  
        print("That's expensive!")  
    else:  
        print("That's not too expensive!")
```

What does this do?

Exercise:

What happens if I enter:

1. 100 for a and 0 for b?
2. 200 for a and 1 for b?
3. 20 for a and 0 for b?
4. 99 for a and 1 for b?

Let's try it out!

YOUR TO-DOS

- Finish reading **Chapter 3**
- Start reading **Chapter 5**
- Start on **HW3** (due next **TUESDAY**)
- Do **Lab1** (turn it in by **Sunday**)

- Dance like you mean it

</LECTURE>