ENGR 101
Engineering Design Workshop

Lecture 5: More Loops, Functions
Edgardo Molina

City College of New York
Writing Python Code

• Source Code == text files (replace '.txt' with '.py')

  • No special format, even notepad will work (but why do that to yourself???)

• Run code in a shell (Terminal / Command Prompt)

  • call the python interpreter to run your program:

    laptop:~ molina$ python myprogram.py

    C:\Documents and Settings\molina> python myprogram.py
Control Structures:
more control
while - else

```python
while test1: # on continue we go straight here
    Do some stuff
    if test2: break
    if test3: continue
    Skipped if test2 true
else:
    Do this on fail of test1 but not on break

Break goes here without else
```
Nesting Control Structures

• We can nest control structures (if, while, for)

• We can nest many times

    while...
        while...
            if...
                for...

• There is a limit; If you reach it, something is **WRONG**

• Abuse makes code unreadable; Use functions instead... (more in a bit)
Counter-Controlled Loops

# decreasing count
count = 5;
while count > 0:
    print(count)
    # do stuff
    count -= 1

# decreasing count
for count in range(5, 0, -1):
    print(count)
    # do stuff

# increasing count
count = 0;
while count < 5:
    print(count)
    # do stuff
    count += 1

equiv

output

5
4
3
2
1

output

0
1
2
3
4
Sentinel-Controlled Loops

```python
value = someValue
while value != sentinelValue:
    # process value
    # get another value

secret = 42
num = int(input("Enter Guess: "))
while num != secret:
    num = int(input("Guess again: "))
else:
    print("You Guessed right!")
```
Accumulating

```
num1 = int(input("Sum of numbers from: "))
num2 = int(input("To number: "))
mysum = 0.0;

while num1 <= num2:
    mysum += num1
    num1 += 1

print("Total sum:", mysum)
```

\[ y = \sum_{i=a}^{n} i \]
Swapping

• $x = 2$
• $y = 3$

• Swap (WRONG)
  • $x = y$
  • $y = x$

• $x = 3$
• $y = 3$

• Swap (CORRECT)
  • $z = x$
  • $x = y$
  • $y = z$

• $x = 3$
• $y = 2$
Multiple Assignments

• \(a\text{Int}, b\text{Int}, c\text{Int} = 1, 2, 3\)

• Swapping with multiple assignment
  
  • \(a\text{Int}, b\text{Int} = b\text{Int}, a\text{Int}\)

• Why does this work? (entire right side is evaluated before assignments)
Debugging
Debugging

• **Syntax Errors**: Python gives us an alert; code crashes

• **Runtime Errors**: How do we fix incorrect results (logic) in our programs?
  
  • We need to trace the codes execution flow.

  • **Tracing**: Keep track of variable values as each line is executed

  • **Print Statements**: strategically add print to view results at each step; don't over do or it will be difficult to keep track

  • Can help us detect **Infinite Loops**
Functions
Why Use Functions?

• Functions provide **encapsulation**, making code *better*, readable

• **Divide and Conquer Problem Solving**
  
  • Break large complicated problems into smaller sub-problems
  
  • Solution to sub-problems written in functions
  
  • sub-problem solutions can be **reused**, **shared**

• **Simplification/Readability**
  
  • Removes duplicate code with function calls
Why Use Functions?

- Abstraction
  - Provides a high-level interface to program
  - You know WHAT it does, not HOW it does it

- Security
  - A small well defined piece of code is easier to prove correct
  - Code can be fixed in one place, and will be corrected everywhere function is called
Function Definition

def functionName(parameter1, parameter2):
    # do stuff
    # do more stuff
    return result
Function Calls

```python
>>> functionName()
>>> function_a(2, 3)
>>> y = function_a(2, 3)
```

Argument List

Store result returned
Functions that do things...

• no parameters

• no return statement

• affect the environment

```python
def turn_right():
    turn_left()
    turn_left()
    turn_left()

def print_long_msg():
    print("Long msg...")
```
Functions that have parameters...

- **definition** has parameters
- **call** takes arguments
- no return statement
- affect the environment

```python
# The definition; doesn't run code
def debug_print(vname, value):
    print("The variable", vname)
    print("has value", value)

var1 = 254343
# The call; runs the code
var_print('var1', var1)
```
Functions that return results...

- **return** keyword
- Function performs processing
- Returns value/object

```python
def celcius2fahrenheit(celcius):
    return celcius*1.8 + 32
```
Functions with default parameters...

• Parameters can have default values

• We can call this function with:

```python
def print_message(msg, times=1):
    for i in range(times):
        print(msg)
```

```python
>>> print_message("Hello class.")
```

```python
>>> print_message("Hello class.", 3)
```

• Can explicitly name arguments:

```python
>>> print_message(times=3, msg="Hello class.")
```