Administrative Items

• Today (2/22/2017)
  • Functions
  • Loops Review

• Monday (2/27/2017)
  • Comprehensive Midterm #1 Review
Working with Functions in Python
Introduction to Programming - Python
Functions
Functions

• A function is a group of statements that exist within a program for the purpose of performing a specific task

• Since the beginning of the semester we have been using a number of Python’s built-in functions, including:
  • print()
  • range()
  • len()
  • random.randint()
  • … etc
Functions

• Most programs perform tasks that are large enough to be broken down into subtasks

• Because of this, programmers often organize their programs into smaller, more manageable chunks by writing their own functions

• Instead of writing one large set of statements we can break down a program into several small functions, allowing us to “divide and conquer” a programming problem
Defining Functions

• Functions, like variables must be named and created before you can use them

• The same naming rules apply for both variables and functions
  • You can’t use any of Python’s keywords
  • No spaces
  • The first character must be A-Z or a-z or the “_” character
  • After the first character you can use A-Z, a-z, “_” or 0-9
  • Uppercase and lowercase characters are distinct
Defining functions

def myfunction():
    print ("Printed from inside a function")

# call the function
myfunction()
Some notes on functions

• When you run a function you say that you “call” it

• Once a function has completed, Python will return back to the line directly after the initial function call

• When a function is called programmers commonly say that the “control” of the program has been transferred to the function. The function is responsible for the program’s execution.

• Functions must be defined before they can be used. In Python we generally place all of our functions at the beginning of our programs.
Flow of Execution
Flow of Execution

Code

```python
print("Welcome!")
for x in range(3):
    print(x)
print("Goodbye!")
```

Output
Flow of Execution

Code

```python
print("Welcome!")
for x in range(3):
    print(x)
print("Goodbye!")
```

Output

Welcome!

```
Flow of Execution

```
Code
print("Welcome!")
for x in range(3):
    print(x)
print("Goodbye!")
```

Output
```
Welcome!
0
1
2
Goodbye!
```
Flow of Execution

Code

```
print("Welcome!")
for x in range(3):
    print(x)
print("Goodbye!")
```

Output

```
Welcome!
0
```
Flow of Execution

**Code**

```python
print ("Welcome!")

for x in range(3):
    print (x)

print ("Goodbye!")
```

**Output**

Welcome!
0

Welcome!
0

Goodbye!
Flow of Execution

```
Code

print("Welcome!")
for x in range(3):
    print(x)
print("Goodbye!")

Output

Welcome!
0
1

```
Flow of Execution

**Code**

```python
print("Welcome!")
for x in range(3):
    print(x)
print("Goodbye!")
```

**Output**

```
Welcome!
0
1
```

# Flow of Execution

<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>print (&quot;Welcome!&quot;)</code></td>
<td>Welcome!</td>
</tr>
<tr>
<td><code>for x in range(3):</code></td>
<td>0</td>
</tr>
<tr>
<td><code>print (x)</code></td>
<td>1</td>
</tr>
<tr>
<td><code>print (x)</code></td>
<td>2</td>
</tr>
<tr>
<td><code>print (&quot;Goodbye!&quot;)</code></td>
<td></td>
</tr>
</tbody>
</table>
Flow of Execution

<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
</table>
| ```python
  print("Welcome!")
  for x in range(3):
    print(x)
  print("Goodbye!")
``` | ```
  Welcome!
  0
  1
  2
  Goodbye!
``` |
Flow of Execution – With Functions
<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>def hello():</td>
<td></td>
</tr>
<tr>
<td>print (&quot;Hi there!&quot;)</td>
<td></td>
</tr>
<tr>
<td>print (&quot;I'm a function!&quot;)</td>
<td></td>
</tr>
<tr>
<td>print (&quot;Good morning&quot;)</td>
<td></td>
</tr>
<tr>
<td>print (&quot;Welcome to class&quot;)</td>
<td></td>
</tr>
<tr>
<td>hello()</td>
<td></td>
</tr>
<tr>
<td>print (&quot;And now we're done.&quot;))</td>
<td></td>
</tr>
</tbody>
</table>
Flow of Execution with Functions

<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
</table>
|```python
def hello():
    print("Hi there!")
    print("I'm a function!")

print("Good morning")
print("Welcome to class")

hello()

print("And now we're done.")``` |        |
Flow of Execution with Functions

Code

```python
def hello():
    print("Hi there!")
    print("I'm a function!")
    print("Good morning")
    print("Welcome to class")

hello()

print("And now we're done.")
```

Output

Good morning
Flow of Execution with Functions

```
Code

def hello():
    print ("Hi there!")
    print ("I'm a function!")

print ("Good morning")
print ("Welcome to class")

hello()

print ("And now we're done.")

Output

Good morning
Welcome to class
```
Flow of Execution with Functions

Code

```python
def hello():
    print("Hi there!")
    print("I'm a function!")

print("Good morning")
print("Welcome to class")

hello()

print("And now we're done.")
```

Output

Good morning
Welcome to class
Flow of Execution with Functions

Code

```python
def hello():
    print("Hi there!")
    print("I'm a function!")

print("Good morning")
print("Welcome to class")

hello()

print("And now we're done.")
```

Output

Good morning
Welcome to class
### Code

def hello():
    print("Hi there!")
    print("I'm a function!")

print("Good morning")
print("Welcome to class")

hello()

print("And now we're done.")

### Output

Good morning
Welcome to class
Hi there!

Good morning
Welcome to class
Hi there!

And now we're done.
Flow of Execution with Functions

**Code**

```python
def hello():
    print("Hi there!")
    print("I'm a function!")
print("Good morning")
print("Welcome to class")
hello()
print("And now we're done.")
```

**Output**

Good morning
Welcome to class
Hi there!
I'm a function!

And now we're done.
Flow of Execution with Functions

<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
</table>
| def hello():
  print ("Hi there!")
  print ("I'm a function!")

  print("Good morning")
  print("Welcome to class")

  hello()

  print ("And now we're done.") | Good morning
Welcome to class
Hi there!
I'm a function!

Flow of Execution with Functions

```python
def hello():
    print("Hi there!")
    print("I'm a function!")
    print("Good morning")
    print("Welcome to class")

hello()

print("And now we're done.")
```

- Good morning
- Welcome to class
- Hi there!
- I'm a function!
- And now we're done.
Multiple Functions
Multiple functions

def hello():
    print ("Hello there!")

def goodbye():
    print ("See ya!")

hello()
goodbye()
def main():
    print ("I have a message for you.")
    message()
    print ("Goodbye!")
def message():
    print ("The password is ‘foo’")
main()
REVIEW
While Loop Basics
While Loop Basics

• Programmers commonly find that they need to write code that performs the same task over and over again

• One solution to this kind of problem is to use "while" loop, which involves the following:
  • Write the code for the task one time
  • Place the code into a while loop which causes Python to repeat it over and over
  • Writing a condition that can be used to stop the loop once the desired # of repetitions has been achieved
  • Then double-check your work by asking: which tasks happen before the loop? Which tasks happen during the loop? Which tasks happen after the loop?
While Loop Basics

- Evaluate a Boolean expression.
- If it is False, skip the block of statements associated with the while loop and condition the program as normal
- If it is True
  - Execute a series of statements.
  - At the end of the statement block re-evaluate the condition
    - If it is True, repeat the block of statements
    - If it is False, skip the block of statements associated with the while loop and continue the program as normal
While Loop Basics

While condition:

statement
statement
statement
statement

}{

standard Boolean condition that evaluates to True or False

the statements that will be repeated

indentation indicates that the statements under the while loop should be repeated
When to use a While loop

• We can use while loops when we need to repeat a task multiple times in order to solve a particular problem. For example:
  • Print the phrase "Hello, World" 100 times
  • Ask the user to enter 10 price values and add them to an accumulator variable

• A while loop works well for tasks that require an unknown number of iterations. For example:
  • Ask the user to enter in a positive number. If the user enters a negative number, re-prompt them until they supply a positive number.
Accumulator Variables

Many programming tasks require you to calculate the total of a series of numbers or the number of times you iterate through a loop (kind of like your homework!)

We can utilize an “accumulator” variable to do this.
Using Accumulator Variables

• Set up your accumulator variables outside of your loops. I generally initialize my accumulator variables right before I enter a repetition structure.

• Decide on a value you want to start your accumulator values at. 0 or 0.0 is generally a good starting point depending on whether you are counting whole numbers or numbers with fractional values.

• Use a self-referential assignment statement when incrementing an accumulator variable. Example:
  • counter = counter + 1
Self-referential assignment statements

```plaintext
# default x to 5
x = 5

x = x + 1

5 + 1 = 6
```
# set up our graphical canvas
# width = 500, height = 500
turtle.setup(500, 500)
wn = turtle.Screen()
# hide the turtle
turtle.hideturtle()

# set the speed to the absolute fastest speed possible
turtle.speed(0)

# set up our background
wn.bgcolor("red")

# pick up the pen and move to a random position on the screen
turtle.penup()
turtle.goto(random.randint(-250, 250), random.randint(-250, 250))
turtle.pendown()

draw a square
x=1
while x<5:
    turtle.forward(100)
    turtle.right(90)
    x+=1

# pick up the pen and move to a random position on the screen
turtle.penup()
turtle.goto(random.randint(-250, 250), random.randint(-250, 250))
turtle.pendown()

draw a square
while x<5:
    turtle.forward(100)
    turtle.right(90)
    x+=1

# pick up the pen and move to a random position on the screen
Augmented Assignment Operators

• The self-referential assignment statement that we just used is extremely useful, and can be extended to use any of the other math operations we have covered in class so far.
  • $a = a + 1$
  • $b = b \times 2$
  • $c = c / 3$
  • $d = d - 4$
Augmented Assignment Operators

• However, Python (and most other programming languages) contains a series of “shortcuts” that can be used to cut down the amount of typing when working with self-referential assignment statements.

• We call these shortcuts the “augmented assignment operators”
## Augmented Assignment Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Usage</th>
<th>Equal to</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+=</code></td>
<td><code>c += 5</code></td>
<td><code>c = c + 5</code></td>
</tr>
<tr>
<td><code>-=</code></td>
<td><code>c -= 2</code></td>
<td><code>c = c - 2</code></td>
</tr>
<tr>
<td><code>*=</code></td>
<td><code>c *= 3</code></td>
<td><code>c = c * 2</code></td>
</tr>
<tr>
<td><code>/=</code></td>
<td><code>c /= 3</code></td>
<td><code>c = c / 3</code></td>
</tr>
<tr>
<td><code>%=</code></td>
<td><code>c %= 3</code></td>
<td><code>c = c % 3</code></td>
</tr>
</tbody>
</table>
Sentinels
(key words)
Sentinels

• Imagine that you want to ask your users to enter in a large number of items that need to be calculated in a certain way.

• You don’t know how many values the user will be entering.

• Given our current toolset we really only have ways to handle this kind of scenario:
  • Ask the user at the end of each iteration if they want to continue. This can be annoying and make your program cumbersome if you will be entering in hundreds or thousands of values.
  • Ask the user ahead of time how many items they will be entering. This can be difficult since the user may not know at the beginning of the loop how many items they will be working with.
**Sentinels**

- A sentinel value is a pre-defined value that the user can type in to indicate that they are finished entering data.

**Example:**
- `>> Enter a test score (type -1 to end): 100`
- `>> Enter a test score (type -1 to end): 80`
- `>> Enter a test score (type -1 to end): -1`
- `>> Your test average is: 90 %`

- In the example above the value -1 is considered a sentinel -- it indicates to the program that the user is finished entering data.

- Sentinels must be distinctive enough that they will not be mistaken for regular data (in the previous example the value -1 was used – there is no way that a “real” test value could be -1).
Count Controlled Loops
Count Controlled Loops

• A count controlled loop is a repetition structure that iterates a specific number of times.

• In contrast, a condition controlled loop iterates a variable number of times – we control the # of iterations through our Boolean condition.
Count Controlled Loops

• You can write a count controlled loop using a while() loop. For example:

  counter = 0
  while counter < 5:
      print (“This will print 5 times”)
      counter += 1
Count Controlled loops

- Python (and all other programming languages) have special structures which can be used to implement a count controlled loop without needing to use a condition controlled loop (though you could always use a condition controlled loop if you wanted to)
The “for” loop

- The “for” loop is Python’s native count controlled loop.

- Example:

```python
for num in [1,2,3,4,5]:
    print ("This will print 5 times")
```
The "for" loop

```
for variable in [value1, value2, etc]:
    statement
    statement
    statement
```

target variable

"for" keyword starts the loop

"in" keyword

list of items to iterate over

indentation

statements to be executed
The “for” loop

- The “for” loop will iterate once for each item defined in the list passed to it when the loop begins.

- Lists in Python are defined by the square bracket characters “[“ and “]”. Items in a list are separated by a comma.

- The first time a “for” loop iterates the target variable will assume the value of the first item in the list.

- The second time a “for” loop iterates the target variable will assume the value of the second item in the list.

- This continues until you reach the end of the list.
The “for” loop

for c in [1,2,3,4]:
    print (c)
The “for” loop

• We will talk more about lists near the end of the semester. With that said, lists can contain collections of different kinds of data. For example:

    for name in ['Craig', 'John', 'Chris']:
        print("The current user is ": name)
The range() function
The range() function

• So far we have been iterating over lists using pre-defined values in our for() loops

• Example:

```python
for x in [1,2,3,4,5]:
    print ('hi')
```

• The range() function lets you dynamically generate lists based on criteria that you define
The range() function

for i in range(5):
    print ('iteration #', i)
The range() function

- The range() function takes at least one argument. In its simplest form it takes a single integer.

- The range() function returns an “iterable”, which is a Python data type similar to a list.

- When passed a single integer the range function will generate an iterable that will cause a for() loop from 0 to the number specified minus one.
The range() function

range() function call | iterable
---|---
range(5) | [0,1,2,3,4]
range(10) | [0,1,2,3,4,5,6,7,8,9]
The range() function

- You can pass additional parameters to the range() function to cause it to behave differently.

- Examples:
  
  ```python
  range(1,5)  # set a start and end value for the range  
              # [1,2,3,4]
  
  range(5,10) # [5,6,7,8,9]
  
  range(0,10,2)  # set a start, end and step (or increment) value  
                 # [0,2,4,6,8]
  
  range(1,10,2)  # [1,3,5,7,9]
  ```
Using loop targets
Loop targets

- In a for loop we generally use the target variable as a reference value for some kind of calculation.
- Remember that the value of the target variable changes with each iteration of the loop.
User controlled ranges
User controlled ranges

• In many cases a programmer knows how many iterations they need in order to accomplish a desired task

• However, sometimes we need to ask the user to control the # of iterations within a loop.

• You can easily do this by substituting a variable within the range() function to control the start, end and step values of the iterable that will be generated
Reverse ranges
Reverse ranges

• The step value passed to the range() function does not necessarily have to be positive.

• If you pass a negative step value to the range() function it will count backwards for you.
Programming Challenge: Blast Off

• Write a countdown program that prompts the user for a max value (i.e. 30)

• Print out a countdown from that number down to zero, then print “blast off!”
Nested Loops
Nested Loops

• A nested loop can be described as a “loop inside of a loop”

• It’s the same idea as nested selection statements (“if” statements inside other “if” statements)
Some notes on Nested Loops

Some notes on nested loops:

- The innermost loop will iterate through all its iterations for every single iteration of an outer loop
- Inner loops complete their iterations faster than outer loops
- To get the total number of iterations of a nested loop, multiply the number of iterations of all the loops
# make the turtle graphics module available
import turtle

# also make the random module available
import random

# set up our graphical canvas
# width = 500, height = 500
turtle.setup(500, 500)
wn = turtle.Screen()
# hide the turtle
turtle.hideturtle()

# set the speed to the absolute fastest speed possible
turtle.speed(0)

# set up our background
wn.bgcolor("red")
z=1
y=1
while y<=z:
    # pick up the pen and move to a random position on the screen
turtle.pendown()
turtle.goto(random.randint(-250, 250), random.randint(-250, 250))
turtle.pendown()

    # draw a square
x=1
while x<5:
    turtle.forward(100)
turtle.right(90)
x+=1
y+=1
z+=1
Nested Loops in the Wild!
Nested Loops in the Wild: Decorative Arts

• There are many examples of the Decorative Arts using nested loops to create wonderful and colorful patterns!

• The Grammar of Ornament (http://digicoll.library.wisc.edu/cgi-bin/DLDecArts/DLDecArts-idx?id=DLDecArts.GramOrnJones) is a Victorian Classic … a collection of patterns used in the Decorative Arts Around the World.
Nested Loops in the Wild: Decorative Arts

• How many repeating and nested patterns can you isolate in the following Ancient Greek designs? (Source)
Nested Loops in the Wild: TIME

- Time is a series of nested for loops and at the center of human experience

- clocks by cwandt.com