Python
Iteration

Learning Objectives

- Discuss iteration
- Connect iteration to square root calculation

Review

- So far, we have discussed
  - Values - numeric and string
  - Operators – a manner to calculate some result from a set of values
  - Variables – the ability to store a value for future use
    - **Recall:** we can reassign a variable to a different value throughout our program (this will come in useful later, today)
  - Functions – segments of code with defined interfaces we can use to partition the process of programming
  - Recursive functions – functions which call themselves, creating repetition
  - Decision making – choosing which segment of code to run, based on some boundary criterion

Iteration (Repetition)

- **Iteration:** repetition of a computational process
- We already have recursion, why do we want other ways of repeating code segments? What's the limitation of recursion?

Iteration (Repetition)

- **Iteration:** repetition of a computational process
- We already have recursion, why do we want other ways of repeating code segments? What's the limitation of recursion?
  - Recursion is expensive – requires many extra compute cycles and requires the computer to track all the waiting function calls
  - Remember the duck() function. After 1000 recursion, the stack overflowed and the program stopped

```
def recur(n):
    if n > 0:
        recur(n-1)

for x in range(100000):
    print(x)
    recur(x)
```

```
2960
2961
2962
2963
2964
2965

---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
<ipython-input-6-f97e124231e0> in <module>()
      5 for x in range(100000):
      6     print(x)
----> 7     recur(x)

<ipython-input-6-f97e124231e0> in recur(n)
      1 def recur(n):
      2     if n > 0:
----> 3         recur(n-1)
      4
      5 for x in range(100000):
```

Simple Repetition

- We can repeat a code segment using a simple repetition recipe based on the **for** keyword
- for **iterates** over the members of a collection of items
  - i.e. a for loop repeats a segment of code one time for each value in a set of values, changing the value of some variable on each loop
  - The variable which changes values is called an **iterator**
- We will revisit **for** many times

```
n = 10
sum = 0
for i in range(n):
    if ((i % 2) == 0):
        print(i)
        sum += i
print("The sum of even values from 0 to " + str(n-1) + " is " + str(sum))
```

```
0
2
4
6
8
The sum of even values from 0 to 9 is 20
```

Simple Repetition

- We can repeat a code segment using a simple repetition recipe based on the for keyword
- for iterates over the members of a collection of items
  - i.e. a for loop repeats a segment of code one time for each value in a set of values, changing the value of some variable on each loop
  - The variable which changes values is called an iterator
- We will revisit for many times

i is the iterator

range creates a list of integers from 0 to (n-1)

```
n = 10
sum = 0
for i in range(n):
    if ((i % 2) == 0):
        print(i)
        sum += i
print("The sum of even values from 0 to " + str(n-1) + " is " + str(sum))
```

```
0
2
4
6
8
The sum of even values from 0 to 9 is 20
```

range() function

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5,10)
range(5, 10)
>>> list(range(5,10))
[5, 6, 7, 8, 9]
>>> list(range(5,10,2))
[5, 7, 9]
>>> list(range(10,1,-1))
[10, 9, 8, 7, 6, 5, 4, 3, 2]
>>>
```

Iteration

- for specifically iterates over a collection of items, what if we want to repeat on some other condition?
- **while** keyword allows for repetition based on an arbitrary condition
- Looks like if, but repeats instead of only running once
- What would happen if the i += 1 line didn't exist?

```
n = 10
sum = 0
i = 0
while (i < n):
    if ((i % 2) == 0):
        print(i)
        sum += i
    i += 1
print("The sum of even values from 0 to " + str(n-1) + " is " + str(sum))
```

```
0
2
4
6
8
The sum of even values from 0 to 9 is 20
```

Iteration

- for specifically iterates over a collection of items, what if we want to repeat on some other condition?
- **while** keyword allows for repetition based on an arbitrary condition
- Looks like if, but repeats instead of only running once
- What would happen if the i += 1 line didn't exist?
  - The boundary condition is never violated, creating an **infinite loop**

```
n = 10
sum = 0
i = 0
while (i < n):
    if ((i % 2) == 0):
        print(i)
        sum += i
print("The sum of even values from 0 to " + str(n-1) + " is " + str(sum))
```

```
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
```

Break

- It is possible to stop a loop partway through, using the **break** instruction
- Without the break, this would be an infinite loop

```
choice = 0
while (True):
    choice = int(input("Pick a number from 1 - 10:  "))
    if(choice == 6):
        print("Right!")
        break
    else:
        print("Wrong!")
```

```
Pick a number from 1 - 10:  3
Wrong!
Pick a number from 1 - 10:  5
Wrong!
Pick a number from 1 - 10:  6
Right!
```

Break

- It is possible to stop a loop partway through, using the **break** instruction
- Without the break, this would be an infinite loop

  - Break should be rare, not common
  - We already have ways to invert logic using negation (!, not)
  - By placing the boundary in the middle of the loop, we make debugging an error harder

```
choice = 0
while (choice != 6):
    choice = int(input("Pick a number from 1 - 10:  "))
    if(choice == 6):
        print("Right!")
    else:
        print("Wrong!")
```

```
Pick a number from 1 - 10:  3
Wrong!
Pick a number from 1 - 10:  5
Wrong!
Pick a number from 1 - 10:  6
Right!
```

Square Root Calculation

- Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.
- For example, one way of computing square roots is Newton's method.
- Suppose that you want to know the square root of a. If you start with almost any estimate, x, you can compute a better estimate with the formula

$$y = \frac{x + a/x}{2}$$

```
>>> a = 4
>>> x = 3
>>> y = (x + a/x)/2
>>> y
2.1666666666666665
```

Square Root Calculation

- Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.
- For example, one way of computing square roots is Newton's method.
- Suppose that you want to know the square root of a. If you start with almost any estimate, x, you can compute a better estimate with the formula

$$y = \frac{x + a/x}{2}$$

```
>>> x = y
>>> y = (x + a/x)/2
>>> y
2.0064102564102564
>>> x = y
>>> y = (x + a/x)/2
>>> y
2.0000102400262145
>>> x = y
>>> y = (x + a/x)/2
>>> y
2.0000000000262146
>>> x = y
>>> y = (x + a/x)/2
>>> y
2.0
```

Square Root Calculation

- Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.
- For example, one way of computing square roots is Newton's method.
- Suppose that you want to know the square root of a. If you start with almost any estimate, x, you can compute a better estimate with the formula

$$y = \frac{x + a/x}{2}$$

Algorithm:

a = Chosen number

x = Estimate of square root

Calculate y with formula

if x != y, then x = y repeat previous line

else y is the square root of a, stop

Square Root Calculation

- Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.
- For example, one way of computing square roots is Newton's method.
- Suppose that you want to know the square root of a. If you start with almost any estimate, x, you can compute a better estimate with the formula

$$y = \frac{x + a/x}{2}$$

```
while True:
    print(x)
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

Square Root Calculation

- For most values of a this works fine, but in general it is dangerous to test float equality.
- Floating-point values are only approximately right: most rational numbers, like 1/3, and irrational numbers can't be represented exactly with a float.
- Rather than checking whether x and y are exactly equal, it is safer to use the built-in function **abs** to compute the absolute value, or magnitude, of the difference between them
- Here epsilon is a very small number like 10^{-16}

$$y = \frac{x + a/x}{2}$$

```
while True:
    print(x)
    y = (x + a/x) / 2
    if abs(y-x) < epsilon:
        break
    x = y
```

Resources

- Bryan Burlingame's notes
- Downey, A. (2016) *Think Python, Second Edition* Sebastopol, CA: O'Reilly Media
- (n.d.). 3.7.0 Documentation. *6. Expressions — Python 3.7.0 documentation*. Retrieved September 11, 2018, from http://docs.python.org/3.7/reference/expressions.html