"Carpe noctem." (Please fill out the survey!)

Week 2: Functions, Strings, and Loops

Notes

- Gone on 4/15 because of the Undergraduate Research Forum; Farzam will be holding a problem solving/coding practice session (let us know if you plan on coming)!
- Skipping object-oriented programming for now, but we will be talking about writing basic functions
- Questions about problems 1 or 2? (the whole set is due next week)
- Paul Points system (suggested by Rishabh Kothari)

Functions

Very important concept!

Functions generally take inputs and do something with them; they may also **return** an output.

- `print()` is a function.
- `type()` is a function that tells you the type of a value.

Examples (type conversion, integer to string):

```
a = 74
b = str(a)
type(a)
type(b)
```

Similar functions: `int()`, `float()`

Note: `int()` does not round; it simply removes the fractional part of a number!

```
c = int(5.55) # returns 5
```

Examples of built-in functions

`abs(x)` – returns the absolute value of a number

`len(x)` – returns the length of an object (the number of items)

This is useful for working with strings and lists!

`pow(x, y)` = returns x to the power y (equivalent to `x**y`, which we have seen before)

Defining custom functions

```
def myfunction():
      print("This is my function.")
      print("There are many like it.")
      print("But this one is mine.")


myfunction() # call our function
```

Notice the indentation on the `print()` lines. Python uses indentation to indicate lines of code in the same **block**, while other languages would surround blocks with braces instead. Most of our blocks will be **function bodies** (the code in functions). You can indent with either a tab character or a fixed number of spaces (typically 2 or 4).

This function does not take any arguments, but you could easily make it so that it does (stay tuned). This function also does not return anything.

Function with a return value

```
def addTwo(x):
    return x + 2
addTwo(4)
```

Here's a function that takes a number and **returns** the value of that number plus two. To indicate a return value, use the `return` keyword.

Many programs operate by primarily making use of this basic principle; functions return values that can be used as parameters to other functions. There is an even entire branch of programming called *functional programming*. See Haskell and R for examples of where this can be done.

Scope

Variables assigned a value in a block do not exist with that value outside of the block. Variables defined in a block are only accessible while you are inside the block (the variable has **local scope**). Variables defined outside of blocks are accessible to every block inside.

```
a = 3 # this a is in global scope
def myfunction():
    a = 4         # a and b are in local scope
    b = 5         # when working with this function
    print(a + b) # uses the a in local scope
                  # not the a in global scope
myfunction() # prints 9
print(a)     # prints 3, not 4
print(b)     # NameError
```

String review

A **string** is an immutable sequence of characters. Immutable means that you cannot change the characters that make up a particular string. You can reassign a new string to a variable, but this is not the same thing as actually changing the original string itself. (whiteboard)

Strings are zero-indexed, meaning that the first item in a list has index 0, the next item has index 1, and so on…

Let's look at one string example.

```
stream = "Aptamer"
```

Character	A	p	t	a	m	e	r
Index	0	1	2	3	4	5	6

Note that "A" is not equal to "a"; these are distinct characters.

Strings cont.

You can retrieve the first item from the string `stream` by typing `stream[0]`. You can retrieve the last item from stream by typing `stream[6]` (the string is seven characters long, but the last index is 6).

```
stream = "Aptamer"
```

Character	A	p	t	a	m	e	r
Index	0	1	2	3	4	5	6

```
print(stream[0]) # prints A
print(stream[3]) # prints a
print(stream[2+2]) # prints m
```

String of unknown length

What do you do if you want the last character of a string that the user has input into your program, but you don't know the length of the string? Use len()!

```
name = input("What is your name?")
print("The last letter is " + name[len(name) - 1])
```

Why do we need to subtract 1?

Notice that we are using the value that len(name) returns as part of the expression for the index for the last character.

Escape characters

Certain characters in strings have special meaning. For example, "\n" means to produce a new line. "\n" is an **escape character**. This looks like two characters, but really, the whole thing is treated as one special character.

```
string = "Hello\nworld!"
print(string)
```

String slicing!

What do you do if you just want the third through fifth characters of a string? Use slicing! Slice syntax looks similar to the syntax for accessing an element. To get a slice of string `string` from index a to index `b`, use `string[a:b]`. You can also add a step parameter! For example, a step size of two means to skip every other character. The syntax is `string[start:stop:step]`.

```
stream = "Aptamer"
slice = stream[2:5] # get 3rd, 4th, 5th characters
print(slice)
```

What happens if you use -1 for step?

Strings: method demonstrations

```
name = "paul nguyen"
name.capitalize() # only affects first letter
name.upper()
name.lower()
name.find("ng")
name.find("z")

test = "    Hello world!    "
test.strip()
test.lstrip()
test.rstrip()
```

for loop syntax

for loops sort of read like English statements. For every "thing" in a sequence, like a list of numbers or names, we want to do something with that "thing".

```
for name in names:
    print name
```

name is a variable that only exists in the for loop. names is a list of strings – in this case, the strings represent names of people.

for loops with range()

Looping over a sequence is a common concept in programming. Let's talk about looping over a sequence of integers using for loops and range().

The range() function takes three parameters: a **start** integer (optional), a **stop** integer (required), and the **step** size (optional). If you don't include a start and step, Python uses 0 for the start and 1 for the step.

Note that the stop integer is not included in the sequence, but the start integer is (e.g., range(1, 10) can be thought of as the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9; this is not quite correct in Python 3, but let's go with it for now).

What do you think this code will print?

```
for i in range(5, 12):
    print(i)
```

Note on range()

In Python 2, range() was called xrange() and range() meant a different function that is not used in Python 3, so if you are looking at old code, keep this in mind.

range() with step size

```
for i in range(0, 10, 2):
    print(i) # print the even numbers from
             # 0 to 10, not including 10
```

range() in reverse

```
for i in range(10, 0, -1):
    print(i) # print the numbers from 0 to
             # 10, not including 0, in
             # reverse order
```

for loops with sequences

for can be used to **iterate** over sequences such as strings, which are basically sequences of characters. This is extremely powerful; we will talk more about iteration in the future.

```
vowels = "AEIOU"
word = "facetiously"
vowel_count = 0
for letter in word.upper(): # make string uppercase
     if letter in vowels:
          vowel_count += 1
print("The number of vowels is: " + str(vowel_count))
```

`while` loops

`while` loops continually execute the statements within while a condition is met. Be careful about accidentally writing infinite loops! You should either update the condition that the `while` loop is checking or otherwise `break` out of the loop.

```
x = 0
while x <= 5:
    print(x, x*x)
    x += 1
```

If we have time…

Python data model:
https://docs.python.org/3/reference/datamodel.html