Methods

We've already seen a few example of methods when learning about Object and Data Structure Types in Python. Methods are essentially functions built into objects. Later on in the course we will learn about how to create our own objects and methods using Object Oriented Programming (OOP) and classes.

Methods perform specific actions on an object and can also take arguments, just like a function. This lecture will serve as just a brief introduction to methods and get you thinking about overall design methods that we will touch back upon when we reach OOP in the course.

Methods are in the form:

```
object.method(arg1,arg2,etc...)
```

You'll later see that we can think of methods as having an argument 'self' referring to the object itself. You can't see this argument but we will be using it later on in the course during the OOP lectures.

Let's take a quick look at what an example of the various methods a list has:

In [1]:
```
# Create a simple list
lst = [1,2,3,4,5]
```

Fortunately, with iPython and the Jupyter Notebook we can quickly see all the possible methods using the tab key. The methods for a list are:

- append
- count
- extend
- insert
- pop
- remove
- reverse
- sort

Let's try out a few of them:

append() allows us to add elements to the end of a list:

In [2]:
```
lst.append(6)
```

In [3]:
```
lst
```

Out[3]:
```
[1, 2, 3, 4, 5, 6]
```

Great! Now how about count()? The count() method will count the number of occurrences of an element in a list.

In [4]:
```
# Check how many times 2 shows up in the list
lst.count(2)
```

Out[4]:
```
1
```

You can always use Shift+Tab in the Jupyter Notebook to get more help about the method. In general Python you can use the help() function:

In [5]:
```
help(lst.count)
```

```
Help on built-in function count:

count(...) method of builtins.list instance
    L.count(value) -> integer -- return number of occurrences of value
```

Feel free to play around with the rest of the methods for a list. Later on in this section your quiz will involve using help and Google searching for methods of different types of objects!

Great! By this lecture you should feel comfortable calling methods of objects in Python!

Functions

Introduction to Functions

This lecture will consist of explaining what a function is in Python and how to create one. Functions will be one of our main building blocks when we construct larger and larger amounts of code to solve problems.

So what is a function?

Formally, a function is a useful device that groups together a set of statements so they can be run more than once. They can also let us specify parameters that can serve as inputs to the functions.

On a more fundamental level, functions allow us to not have to repeatedly write the same code again and again. If you remember back to the lessons on strings and lists, remember that we used a function len() to get the length of a string. Since checking the length of a sequence is a common task you would want to write a function that can do this repeatedly at command.

Functions will be one of most basic levels of reusing code in Python, and it will also allow us to start thinking of program design (we will dive much deeper into the ideas of design when we learn about Object Oriented Programming).

def Statements

Let's see how to build out a function's syntax in Python. It has the following form:

In [1]:
```
def name_of_function(arg1,arg2):
    '''
    This is where the function's Document String (docstring) goes
    '''
    # Do stuff here
    # Return desired result
```

We begin with `def` then a space followed by the name of the function. Try to keep names relevant, for example len() is a good name for a length() function. Also be careful with names, you wouldn't want to call a function the same name as a built-in function in Python (https://docs.python.org/2/library/functions.html) (such as len).

Next come a pair of parentheses with a number of arguments separated by a comma. These arguments are the inputs for your function. You'll be able to use these inputs in your function and reference them. After this you put a colon.

Now here is the important step, you must indent to begin the code inside your function correctly. Python makes use of *whitespace* to organize code. Lots of other programing languages do not do this, so keep that in mind.

Next you'll see the docstring, this is where you write a basic description of the function. Using iPython and iPython Notebooks, you'll be able to read these docstrings by pressing Shift+Tab after a function name. Docstrings are not necessary for simple functions, but it's good practice to put them in so you or other people can easily understand the code you write.

After all this you begin writing the code you wish to execute.

The best way to learn functions is by going through examples. So let's try to go through examples that relate back to the various objects and data structures we learned about before.

Example 1: A simple print 'hello' function

In [2]:
```
def say_hello():
    print('hello')
```

Call the function:

In [3]:
```
say_hello()
```

```
hello
```

Example 2: A simple greeting function

Let's write a function that greets people with their name.

In [4]:
```
def greeting(name):
    print('Hello %s' %(name))
```

In [5]:
```
greeting('Jose')
```

```
Hello Jose
```

Using return

Let's see some example that use a `return` statement. `return` allows a function to *return* a result that can then be stored as a variable, or used in whatever manner a user wants.

Example 3: Addition function

In [6]:
```
def add_num(num1,num2):
    return num1+num2
```

In [7]:
```
add_num(4,5)
```

Out[7]:
```
9
```

In [8]:
```
# Can also save as variable due to return
result = add_num(4,5)
```

In [9]:
```
print(result)
```

```
9
```

What happens if we input two strings?

In [10]:
```
add_num('one','two')
```

Out[10]:
```
'onetwo'
```

Note that because we don't declare variable types in Python, this function could be used to add numbers or sequences together! We'll later learn about adding in checks to make sure a user puts in the correct arguments into a function.

Let's also start using `break`, `continue`, and `pass` statements in our code. We introduced these during the `while` lecture.

Finally let's go over a full example of creating a function to check if a number is prime (a common interview exercise).

We know a number is prime if that number is only evenly divisible by 1 and itself. Let's write our first version of the function to check all the numbers from 1 to N and perform modulo checks.

In [11]:
```
def is_prime(num):
    '''
    Naive method of checking for primes.
    '''
    for n in range(2,num):
        if num % n == 0:
            print(num,'is not prime')
            break
    else: # If never mod zero, then prime
        print(num,'is prime!')
```

In [12]:
```
is_prime(16)
```

```
16 is not prime
```

In [13]:
```
is_prime(17)
```

```
17 is prime!
```

Note how the `else` lines up under `for` and not `if`. This is because we want the `for` loop to exhaust all possibilities in the range before printing our number is prime.

Also note how we break the code after the first print statement. As soon as we determine that a number is not prime we break out of the `for` loop.

We can actually improve this function by only checking to the square root of the target number, and by disregarding all even numbers after checking for 2. We'll also switch to returning a boolean value to get an example of using return statements:

In [14]:
```
import math

def is_prime2(num):
    '''
    Better method of checking for primes.
    '''
    if num % 2 == 0 and num > 2:
        return False
    for i in range(3, int(math.sqrt(num)) + 1, 2):
        if num % i == 0:
            return False
    return True
```

In [15]:
```
is_prime2(18)
```

Out[15]:
```
False
```

Why don't we have any `break` statements? It should be noted that as soon as a function *returns* something, it shuts down. A function can deliver multiple print statements, but it will only obey one `return`.

Great! You should now have a basic understanding of creating your own functions to save yourself from repeatedly writing code!

Function Practice Exercises

Problems are arranged in increasing difficulty:

- Warmup - these can be solved using basic comparisons and methods
- Level 1 - these may involve if/then conditional statements and simple methods
- Level 2 - these may require iterating over sequences, usually with some kind of loop
- Challenging - these will take some creativity to solve

WARMUP SECTION:

LESSER OF TWO EVENS: Write a function that returns the lesser of two given numbers *if* both numbers are even, but returns the greater if one or both numbers are odd

```
lesser_of_two_evens(2,4) --> 2
lesser_of_two_evens(2,5) --> 5
```

In [ ]:
```
def lesser_of_two_evens(a,b):
    pass
```

In [ ]:
```
# Check
lesser_of_two_evens(2,4)
```

In [ ]:
```
# Check
lesser_of_two_evens(2,5)
```

ANIMAL CRACKERS: Write a function takes a two-word string and returns True if both words begin with same letter

```
animal_crackers('Levelheaded Llama') --> True
animal_crackers('Crazy Kangaroo') --> False
```

In [ ]:
```
def animal_crackers(text):
    pass
```

In [ ]:
```
# Check
animal_crackers('Levelheaded Llama')
```

In [ ]:
```
# Check
animal_crackers('Crazy Kangaroo')
```

MAKES TWENTY: Given two integers, return True if the sum of the integers is 20 *or* if one of the integers is 20. If not, return False

```
makes_twenty(20,10) --> True
makes_twenty(12,8) --> True
makes_twenty(2,3) --> False
```

In [ ]:
```
def makes_twenty(n1,n2):
    pass
```

In [ ]:
```
# Check
makes_twenty(20,10)
```

In [ ]:
```
# Check
makes_twenty(2,3)
```

LEVEL 1 PROBLEMS

OLD MACDONALD: Write a function that capitalizes the first and fourth letters of a name

```
old_macdonald('macdonald') --> MacDonald
```

Note: `'macdonald'.capitalize()` returns `'Macdonald'`

In [ ]:
```
def old_macdonald(name):
    pass
```

In [ ]:
```
# Check
old_macdonald('macdonald')
```

MASTER YODA: Given a sentence, return a sentence with the words reversed

```
master_yoda('I am home') --> 'home am I'
master_yoda('We are ready') --> 'ready are We'
```

Note: The .join() method may be useful here. The .join() method allows you to join together strings in a list with some connector string. For example, some uses of the .join() method:

```
>>> "--".join(['a','b','c'])
>>> 'a--b--c'
```

This means if you had a list of words you wanted to turn back into a sentence, you could just join them with a single space string:

```
>>> " ".join(['Hello','world'])
>>> "Hello world"
```

In [ ]:
```
def master_yoda(text):
    pass
```

In [ ]:
```
# Check
master_yoda('I am home')
```

In [ ]:
```
# Check
master_yoda('We are ready')
```

ALMOST THERE: Given an integer n, return True if n is within 10 of either 100 or 200

```
almost_there(90) --> True
almost_there(104) --> True
almost_there(150) --> False
almost_there(209) --> True
```

NOTE: `abs(num)` returns the absolute value of a number

In [ ]:
```
def almost_there(n):
    pass
```

In [ ]:
```
# Check
almost_there(104)
```

In [ ]:
```
# Check
almost_there(150)
```

In [ ]:
```
# Check
almost_there(209)
```

LEVEL 2 PROBLEMS

FIND 33:

Given a list of ints, return True if the array contains a 3 next to a 3 somewhere.

```
has_33([1, 3, 3]) → True
has_33([1, 3, 1, 3]) → False
has_33([3, 1, 3]) → False
```

In [ ]:
```
def has_33(nums):
    pass
```

In [ ]:
```
# Check
has_33([1, 3, 3])
```

In [ ]:
```
# Check
has_33([1, 3, 1, 3])
```

In [ ]:
```
# Check
has_33([3, 1, 3])
```

PAPER DOLL: Given a string, return a string where for every character in the original there are three characters

```
paper_doll('Hello') --> 'HHHeeellllllooo'
paper_doll('Mississippi') --> 'MMMiiissssssiiipppppppiii'
```

In [ ]:
```
def paper_doll(text):
    pass
```

In [ ]:
```
# Check
paper_doll('Hello')
```

In [ ]:
```
# Check
paper_doll('Mississippi')
```

BLACKJACK: Given three integers between 1 and 11, if their sum is less than or equal to 21, return their sum. If their sum exceeds 21 *and* there's an eleven, reduce the total sum by 10. Finally, if the sum (even after adjustment) exceeds 21, return 'BUST'

```
blackjack(5,6,7) --> 18
blackjack(9,9,9) --> 'BUST'
blackjack(9,9,11) --> 19
```

In [ ]:
```
def blackjack(a,b,c):
    pass
```

In [ ]:
```
# Check
blackjack(5,6,7)
```

In [ ]:
```
# Check
blackjack(9,9,9)
```

In [ ]:
```
# Check
blackjack(9,9,11)
```

SUMMER OF '69: Return the sum of the numbers in the array, except ignore sections of numbers starting with a 6 and extending to the next 9 (every 6 will be followed by at least one 9). Return 0 for no numbers.

```
summer_69([1, 3, 5]) --> 9
summer_69([4, 5, 6, 7, 8, 9]) --> 9
summer_69([2, 1, 6, 9, 11]) --> 14
```

In [ ]:
```
def summer_69(arr):
    pass
```

In [ ]:
```
# Check
summer_69([1, 3, 5])
```

In [ ]:
```
# Check
summer_69([4, 5, 6, 7, 8, 9])
```

In [ ]:
```
# Check
summer_69([2, 1, 6, 9, 11])
```

CHALLENGING PROBLEMS

SPY GAME: Write a function that takes in a list of integers and returns True if it contains 007 in order

```
spy_game([1,2,4,0,0,7,5]) --> True
spy_game([1,0,2,4,0,5,7]) --> True
spy_game([1,7,2,0,4,5,0]) --> False
```

In [ ]:
```
def spy_game(nums):
    pass
```

In [ ]:
```
# Check
spy_game([1,2,4,0,0,7,5])
```

In [ ]:
```
# Check
spy_game([1,0,2,4,0,5,7])
```

In [ ]:
```
# Check
spy_game([1,7,2,0,4,5,0])
```

COUNT PRIMES: Write a function that returns the *number* of prime numbers that exist up to and including a given number

```
count_primes(100) --> 25
```

By convention, 0 and 1 are not prime.

In [ ]:
```
def count_primes(num):
    pass
```

In [ ]:
```
# Check
count_primes(100)
```

Just for fun:

PRINT BIG: Write a function that takes in a single letter, and returns a 5x5 representation of that letter

```
print_big('a')

out:   *
      * *
     *****
     *   *
     *   *
```

HINT: Consider making a dictionary of possible patterns, and mapping the alphabet to specific 5-line combinations of patterns.
For purposes of this exercise, it's ok if your dictionary stops at "E".

In [ ]:
```
def print_big(letter):
    pass
```

In [ ]:
```
print_big('a')
```

Great Job!

Function Practice Exercises - Solutions

Problems are arranged in increasing difficulty:

- Warmup - these can be solved using basic comparisons and methods
- Level 1 - these may involve if/then conditional statements and simple methods
- Level 2 - these may require iterating over sequences, usually with some kind of loop
- Challenging - these will take some creativity to solve

WARMUP SECTION:

LESSER OF TWO EVENS: Write a function that returns the lesser of two given numbers *if* both numbers are even, but returns the greater if one or both numbers are odd

```
lesser_of_two_evens(2,4) --> 2
lesser_of_two_evens(2,5) --> 5
```

In [1]:
```
def lesser_of_two_evens(a,b):
    if a%2 == 0 and b%2 == 0:
        return min(a,b)
    else:
        return max(a,b)
```

In [2]:
```
# Check
lesser_of_two_evens(2,4)
```

Out[2]: 2

In [3]:
```
# Check
lesser_of_two_evens(2,5)
```

Out[3]: 5

ANIMAL CRACKERS: Write a function takes a two-word string and returns True if both words begin with same letter

```
animal_crackers('Levelheaded Llama') --> True
animal_crackers('Crazy Kangaroo') --> False
```

In [4]:
```
def animal_crackers(text):
    wordlist = text.split()
    return wordlist[0][0] == wordlist[1][0]
```

In [5]:
```
# Check
animal_crackers('Levelheaded Llama')
```

Out[5]: True

In [6]:
```
# Check
animal_crackers('Crazy Kangaroo')
```

Out[6]: False

MAKES TWENTY: Given two integers, return True if the sum of the integers is 20 *or* if one of the integers is 20. If not, return False

```
makes_twenty(20,10) --> True
makes_twenty(12,8) --> True
makes_twenty(2,3) --> False
```

In [7]:
```
def makes_twenty(n1,n2):
    return (n1+n2)==20 or n1==20 or n2==20
```

In [8]:
```
# Check
makes_twenty(20,10)
```

Out[8]: True

In [9]:
```
# Check
makes_twenty(12,8)
```

Out[9]: True

In [10]:
```
#Check
makes_twenty(2,3)
```

Out[10]: False

LEVEL 1 PROBLEMS

OLD MACDONALD: Write a function that capitalizes the first and fourth letters of a name

```
old_macdonald('macdonald') --> MacDonald
```

Note: `'macdonald'.capitalize()` returns `'Macdonald'`

In [11]:
```
def old_macdonald(name):
    if len(name) > 3:
        return name[:3].capitalize() + name[3:].capitalize()
    else:
        return 'Name is too short!'
```

In [12]:
```
# Check
old_macdonald('macdonald')
```

Out[12]:
```
'MacDonald'
```

MASTER YODA: Given a sentence, return a sentence with the words reversed

```
master_yoda('I am home') --> 'home am I'
master_yoda('We are ready') --> 'ready are We'
```

In [13]:
```
def master_yoda(text):
    return ' '.join(text.split()[::-1])
```

In [14]:
```
# Check
master_yoda('I am home')
```

Out[14]:
```
'home am I'
```

In [15]:
```
# Check
master_yoda('We are ready')
```

Out[15]:
```
'ready are We'
```

ALMOST THERE: Given an integer n, return True if n is within 10 of either 100 or 200

```
almost_there(90) --> True
almost_there(104) --> True
almost_there(150) --> False
almost_there(209) --> True
```

NOTE: `abs(num)` returns the absolute value of a number

In [16]:
```
def almost_there(n):
    return ((abs(100 - n) <= 10) or (abs(200 - n) <= 10))
```

In [17]:
```
# Check
almost_there(90)
```

Out[17]:
```
True
```

In [18]:
```
# Check
almost_there(104)
```

Out[18]:
```
True
```

In [19]:
```
# Check
almost_there(150)
```

Out[19]:
```
False
```

In [20]:
```
# Check
almost_there(209)
```

Out[20]:
```
True
```

LEVEL 2 PROBLEMS

FIND 33:

Given a list of ints, return True if the array contains a 3 next to a 3 somewhere.

```
has_33([1, 3, 3]) → True
has_33([1, 3, 1, 3]) → False
has_33([3, 1, 3]) → False
```

In [21]:
```
def has_33(nums):
    for i in range(0, len(nums)-1):

        # nicer looking alternative in commented code
        #if nums[i] == 3 and nums[i+1] == 3:

        if nums[i:i+2] == [3,3]:
            return True

    return False
```

In [22]:
```
# Check
has_33([1, 3, 3])
```

Out[22]:
```
True
```

In [23]:
```
# Check
has_33([1, 3, 1, 3])
```

Out[23]:
```
False
```

In [24]:
```
# Check
has_33([3, 1, 3])
```

Out[24]:
```
False
```

PAPER DOLL: Given a string, return a string where for every character in the original there are three characters

```
paper_doll('Hello') --> 'HHHeeellllllooo'
paper_doll('Mississippi') --> 'MMMiiissssssiiipppppppiii'
```

In [25]:
```
def paper_doll(text):
    result = ''
    for char in text:
        result += char * 3
    return result
```

In [26]:
```
# Check
paper_doll('Hello')
```

Out[26]: 'HHHeeellllllooo'

In [27]:
```
# Check
paper_doll('Mississippi')
```

Out[27]: 'MMMiiissssssiiissssssiiipppppppiii'

BLACKJACK: Given three integers between 1 and 11, if their sum is less than or equal to 21, return their sum. If their sum exceeds 21 *and* there's an eleven, reduce the total sum by 10. Finally, if the sum (even after adjustment) exceeds 21, return 'BUST'

```
blackjack(5,6,7) --> 18
blackjack(9,9,9) --> 'BUST'
blackjack(9,9,11) --> 19
```

In [28]:
```
def blackjack(a,b,c):

    if sum((a,b,c)) <= 21:
        return sum((a,b,c))
    elif sum((a,b,c)) <=31 and 11 in (a,b,c):
        return sum((a,b,c)) - 10
    else:
        return 'BUST'
```

In [29]:
```
# Check
blackjack(5,6,7)
```

Out[29]: 18

In [30]:
```
# Check
blackjack(9,9,9)
```

Out[30]: 'BUST'

In [31]:
```
# Check
blackjack(9,9,11)
```

Out[31]: 19

SUMMER OF '69: Return the sum of the numbers in the array, except ignore sections of numbers starting with a 6 and extending to the next 9 (every 6 will be followed by at least one 9). Return 0 for no numbers.

```
summer_69([1, 3, 5]) --> 9
summer_69([4, 5, 6, 7, 8, 9]) --> 9
summer_69([2, 1, 6, 9, 11]) --> 14
```

In [32]:
```
def summer_69(arr):
    total = 0
    add = True
    for num in arr:
        while add:
            if num != 6:
                total += num
                break
            else:
                add = False
        while not add:
            if num != 9:
                break
            else:
                add = True
                break
    return total
```

In [33]:
```
# Check
summer_69([1, 3, 5])
```

Out[33]: 9

In [34]:
```
# Check
summer_69([4, 5, 6, 7, 8, 9])
```

Out[34]: 9

In [35]:
```
# Check
summer_69([2, 1, 6, 9, 11])
```

Out[35]: 14

CHALLENGING PROBLEMS

SPY GAME: Write a function that takes in a list of integers and returns True if it contains 007 in order

```
spy_game([1,2,4,0,0,7,5]) --> True
spy_game([1,0,2,4,0,5,7]) --> True
spy_game([1,7,2,0,4,5,0]) --> False
```

In [36]:
```
def spy_game(nums):

    code = [0,0,7,'x']

    for num in nums:
        if num == code[0]:
            code.pop(0)   # code.remove(num) also works

    return len(code) == 1
```

In [37]:
```
# Check
spy_game([1,2,4,0,0,7,5])
```

Out[37]: True

In [38]:
```
# Check
spy_game([1,0,2,4,0,5,7])
```

Out[38]: True

In [39]:
```
# Check
spy_game([1,7,2,0,4,5,0])
```

Out[39]: False

COUNT PRIMES: Write a function that returns the *number* of prime numbers that exist up to and including a given number

```
count_primes(100) --> 25
```

By convention, 0 and 1 are not prime.

In [40]:
```
def count_primes(num):
    primes = [2]
    x = 3
    if num < 2:  # for the case of num = 0 or 1
        return 0
    while x <= num:
        for y in range(3,x,2):  # test all odd factors up to x-1
            if x%y == 0:
                x += 2
                break
        else:
            primes.append(x)
            x += 2
    print(primes)
    return len(primes)
```

In [41]:
```
# Check
count_primes(100)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 7
3, 79, 83, 89, 97]
```

Out[41]: 25

BONUS: Here's a faster version that makes use of the prime numbers we're collecting as we go!

In [42]:
```
def count_primes2(num):
    primes = [2]
    x = 3
    if num < 2:
        return 0
    while x <= num:
        for y in primes:  # use the primes list!
            if x%y == 0:
                x += 2
                break
        else:
            primes.append(x)
            x += 2
    print(primes)
    return len(primes)
```

In [43]:
```
count_primes2(100)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 7
3, 79, 83, 89, 97]
```

Out[43]: 25

Just for fun, not a real problem :)

PRINT BIG: Write a function that takes in a single letter, and returns a 5x5 representation of that letter

```
print_big('a')

out:   *
      * *
     *****
     *   *
     *   *
```

HINT: Consider making a dictionary of possible patterns, and mapping the alphabet to specific 5-line combinations of patterns.
For purposes of this exercise, it's ok if your dictionary stops at "E".

In [44]:
```
def print_big(letter):
    patterns = {1:'  *  ',2:' * * ',3:'*   *',4:'*****',5:'**** ',6:'   * ',7:' 
    alphabet = {'A':[1,2,4,3,3],'B':[5,3,5,3,5],'C':[4,9,9,9,4],'D':[5,3,3,3,5],
    for pattern in alphabet[letter.upper()]:
        print(patterns[pattern])
```

In [45]:
```
print_big('a')
```

```
  *
 * *
*****
*   *
*   *
```

Great Job!

Lambda Expressions, Map, and Filter

Now its time to quickly learn about two built in functions, filter and map. Once we learn about how these operate, we can learn about the lambda expression, which will come in handy when you begin to develop your skills further!

map function

The **map** function allows you to "map" a function to an iterable object. That is to say you can quickly call the same function to every item in an iterable, such as a list. For example:

In [1]:
```
def square(num):
    return num**2
```

In [2]:
```
my_nums = [1,2,3,4,5]
```

In [5]:
```
map(square,my_nums)
```

Out[5]:
```
<map at 0x205baec21d0>
```

In [7]:
```
# To get the results, either iterate through map()
# or just cast to a list
list(map(square,my_nums))
```

Out[7]:
```
[1, 4, 9, 16, 25]
```

The functions can also be more complex

In [8]:
```
def splicer(mystring):
    if len(mystring) % 2 == 0:
        return 'even'
    else:
        return mystring[0]
```

In [9]:
```
mynames = ['John','Cindy','Sarah','Kelly','Mike']
```

In [10]:
```
list(map(splicer,mynames))
```

Out[10]:
```
['even', 'C', 'S', 'K', 'even']
```

filter function

The filter function returns an iterator yielding those items of iterable for which function(item) is true. Meaning you need to filter by a function that returns either True or False. Then passing that into filter (along with your iterable) and you will get back only the results that would return True when passed to the function.

In [12]:
```
def check_even(num):
    return num % 2 == 0
```

In [13]:
```
nums = [0,1,2,3,4,5,6,7,8,9,10]
```

In [15]:
```
filter(check_even,nums)
```

Out[15]:
```
<filter at 0x205baed4710>
```

In [16]:
```
list(filter(check_even,nums))
```

Out[16]:
```
[0, 2, 4, 6, 8, 10]
```

lambda expression

One of Pythons most useful (and for beginners, confusing) tools is the lambda expression. lambda expressions allow us to create "anonymous" functions. This basically means we can quickly make ad-hoc functions without needing to properly define a function using def.

Function objects returned by running lambda expressions work exactly the same as those created and assigned by defs. There is key difference that makes lambda useful in specialized roles:

lambda's body is a single expression, not a block of statements.

- The lambda's body is similar to what we would put in a def body's return statement. We simply type the result as an expression instead of explicitly returning it. Because it is limited to an expression, a lambda is less general that a def. We can only squeeze design, to limit program nesting. lambda is designed for coding simple functions, and def handles the larger tasks.

Lets slowly break down a lambda expression by deconstructing a function:

In [17]:
```
def square(num):
    result = num**2
    return result
```

In [18]:
```
square(2)
```

Out[18]:
```
4
```

We could simplify it:

In [19]:
```
def square(num):
    return num**2
```

In [20]:
```
square(2)
```

Out[20]:
```
4
```

We could actually even write this all on one line.

In [21]:
```
def square(num): return num**2
```

In [22]:
```
square(2)
```

Out[22]:
```
4
```

This is the form a function that a lambda expression intends to replicate. A lambda expression can then be written as:

In [23]:
```
lambda num: num ** 2
```

Out[23]:
```
<function __main__.<lambda>>
```

In [25]:
```
# You wouldn't usually assign a name to a lambda expression, this is just for der
square = lambda num: num **2
```

In [26]:
```
square(2)
```

Out[26]:
```
4
```

So why would use this? Many function calls need a function passed in, such as map and filter. Often you only need to use the function you are passing in once, so instead of formally defining it, you just use the lambda expression. Let's repeat some of the examples from above with a lambda expression

In [29]:
```
list(map(lambda num: num ** 2, my_nums))
```

Out[29]:
```
[1, 4, 9, 16, 25]
```

In [30]:
```
list(filter(lambda n: n % 2 == 0,nums))
```

Out[30]:
```
[0, 2, 4, 6, 8, 10]
```

Here are a few more examples, keep in mind the more comples a function is, the harder it is to translate into a lambda expression, meaning sometimes its just easier (and often the only way) to create the def keyword function.

** Lambda expression for grabbing the first character of a string: **

In [31]:
```
lambda s: s[0]
```

Out[31]:
```
<function __main__.<lambda>>
```

** Lambda expression for reversing a string: **

In [32]:
```
lambda s: s[::-1]
```

Out[32]:
```
<function __main__.<lambda>>
```

You can even pass in multiple arguments into a lambda expression. Again, keep in mind that not every function can be translated into a lambda expression.

In [34]:
```
lambda x,y : x + y
```

Out[34]:
```
<function __main__.<lambda>>
```

You will find yourself using lambda expressions often with certain non-built-in libraries, for example the pandas library for data analysis works very well with lambda expressions.

Nested Statements and Scope

Now that we have gone over writing our own functions, it's important to understand how Python deals with the variable names you assign. When you create a variable name in Python the name is stored in a *name-space*. Variable names also have a *scope*, the scope determines the visibility of that variable name to other parts of your code.

Let's start with a quick thought experiment; imagine the following code:

In [1]:
```
x = 25

def printer():
    x = 50
    return x

# print(x)
# print(printer())
```

What do you imagine the output of printer() is? 25 or 50? What is the output of print x? 25 or 50?

In [2]:
```
print(x)
```

```
25
```

In [3]:
```
print(printer())
```

```
50
```

Interesting! But how does Python know which **x** you're referring to in your code? This is where the idea of scope comes in. Python has a set of rules it follows to decide what variables (such as **x** in this case) you are referencing in your code. Lets break down the rules:

This idea of scope in your code is very important to understand in order to properly assign and call variable names.

In simple terms, the idea of scope can be described by 3 general rules:

1. Name assignments will create or change local names by default.
2. Name references search (at most) four scopes, these are:
    - local
    - enclosing functions
    - global
    - built-in
3. Names declared in global and nonlocal statements map assigned names to enclosing module and function scopes.

The statement in #2 above can be defined by the LEGB rule.

LEGB Rule:

L: Local — Names assigned in any way within a function (def or lambda), and not declared global in that function.

E: Enclosing function locals — Names in the local scope of any and all enclosing functions (def or lambda), from inner to outer.

G: Global (module) — Names assigned at the top-level of a module file, or declared global in a def within the file.

B: Built-in (Python) — Names preassigned in the built-in names module : open, range, SyntaxError,...

Quick examples of LEGB

Local

In [4]:
```
# x is local here:
f = lambda x:x**2
```

Enclosing function locals

This occurs when we have a function inside a function (nested functions)

In [5]:
```
name = 'This is a global name'

def greet():
    # Enclosing function
    name = 'Sammy'

    def hello():
        print('Hello '+name)

    hello()

greet()
```

```
Hello Sammy
```

Note how Sammy was used, because the hello() function was enclosed inside of the greet function!

Global

Luckily in Jupyter a quick way to test for global variables is to see if another cell recognizes the variable!

In [6]:
```
print(name)
```

```
This is a global name
```

Built-in

These are the built-in function names in Python (don't overwrite these!)

In [7]:
```
len
```

Out[7]:
```
<function len>
```

Local Variables

When you declare variables inside a function definition, they are not related in any way to other variables with the same names used outside the function - i.e. variable names are local to the function. This is called the scope of the variable. All variables have the scope of the block they are declared in starting from the point of definition of the name.

Example:

In [8]:
```
x = 50

def func(x):
    print('x is', x)
    x = 2
    print('Changed local x to', x)

func(x)
print('x is still', x)
```

```
x is 50
Changed local x to 2
x is still 50
```

The first time that we print the value of the name **x** with the first line in the function's body, Python uses the value of the parameter declared in the main block, above the function definition.

Next, we assign the value 2 to **x*. *The name* *x** is local to our function. So, when we change the value of **x** in the function, the **x** defined in the main block remains unaffected.

With the last print statement, we display the value of **x** as defined in the main block, thereby confirming that it is actually unaffected by the local assignment within the previously called function.

The `global` statement

If you want to assign a value to a name defined at the top level of the program (i.e. not inside any kind of scope such as functions or classes), then you have to tell Python that the name is not local, but it is global. We do this using the `global` statement. It is impossible to assign a value to a

variable defined outside a function without the global statement.

You can use the values of such variables defined outside the function (assuming there is no variable with the same name within the function). However, this is not encouraged and should be avoided since it becomes unclear to the reader of the program as to where that variable's definition is. Using the `global` statement makes it amply clear that the variable is defined in an outermost block.

Example:

In [9]:
```
x = 50

def func():
    global x
    print('This function is now using the global x!')
    print('Because of global x is: ', x)
    x = 2
    print('Ran func(), changed global x to', x)

print('Before calling func(), x is: ', x)
func()
print('Value of x (outside of func()) is: ', x)
```

```
Before calling func(), x is:  50
This function is now using the global x!
Because of global x is:  50
Ran func(), changed global x to 2
Value of x (outside of func()) is:  2
```

The `global` statement is used to declare that **x** is a global variable - hence, when we assign a value to **x** inside the function, that change is reflected when we use the value of **x** in the main block.

You can specify more than one global variable using the same global statement e.g. `global x, y, z`.

Conclusion

You should now have a good understanding of Scope (you may have already intuitively felt right about Scope which is great!) One last mention is that you can use the **globals()** and **locals()** functions to check what are your current local and global variables.

Another thing to keep in mind is that everything in Python is an object! I can assign variables to functions just like I can with numbers! We will go over this again in the decorator section of the course!

*args and **kwargs

Work with Python long enough, and eventually you will encounter `*args` and `**kwargs`. These strange terms show up as parameters in function definitions. What do they do? Let's review a simple function:

In [1]:
```
def myfunc(a,b):
    return sum((a,b))*.05

myfunc(40,60)
```

Out[1]:
```
5.0
```

This function returns 5% of the sum of **a** and **b**. In this example, **a** and **b** are *positional* arguments; that is, 40 is assigned to **a because it is the first argument, and 60 to **b**. Notice also that to work with multiple positional arguments in the `sum()` function we had to pass them in as a tuple.

What if we want to work with more than two numbers? One way would be to assign a *lot* of parameters, and give each one a default value.

In [2]:
```
def myfunc(a=0,b=0,c=0,d=0,e=0):
    return sum((a,b,c,d,e))*.05

myfunc(40,60,20)
```

Out[2]:
```
6.0
```

Obviously this is not a very efficient solution, and that's where `*args` comes in.

*args

When a function parameter starts with an asterisk, it allows for an *arbitrary number* of arguments, and the function takes them in as a tuple of values. Rewriting the above function:

In [3]:
```
def myfunc(*args):
    return sum(args)*.05

myfunc(40,60,20)
```

Out[3]:
```
6.0
```

Notice how passing the keyword "args" into the `sum()` function did the same thing as a tuple of arguments.

It is worth noting that the word "args" is itself arbitrary - any word will do so long as it's preceded by an asterisk. To demonstrate this:

In [4]:
```
def myfunc(*spam):
    return sum(spam)*.05

myfunc(40,60,20)
```

Out[4]:
```
6.0
```

**kwargs

Similarly, Python offers a way to handle arbitrary numbers of *keyworded* arguments. Instead of creating a tuple of values, `**kwargs` builds a dictionary of key/value pairs. For example:

In [5]:
```
def myfunc(**kwargs):
    if 'fruit' in kwargs:
        print(f"My favorite fruit is {kwargs['fruit']}")  # review String Format
    else:
        print("I don't like fruit")

myfunc(fruit='pineapple')
```

```
My favorite fruit is pineapple
```

In [6]:
```
myfunc()
```

```
I don't like fruit
```

*args and **kwargs combined

You can pass `*args` and `**kwargs` into the same function, but `*args` have to appear before `**kwargs`

In [7]:
```
def myfunc(*args, **kwargs):
    if 'fruit' and 'juice' in kwargs:
        print(f"I like {' and '.join(args)} and my favorite fruit is {kwargs['fru
        print(f"May I have some {kwargs['juice']} juice?")
    else:
        pass

myfunc('eggs','spam',fruit='cherries',juice='orange')
```

```
I like eggs and spam and my favorite fruit is cherries
May I have some orange juice?
```

Placing keyworded arguments ahead of positional arguments raises an exception:

In [8]:
```
myfunc(fruit='cherries',juice='orange','eggs','spam')
```

```
  File "<ipython-input-8-fc6ff65addcc>", line 1
    myfunc(fruit='cherries',juice='orange','eggs','spam')
                                           ^
SyntaxError: positional argument follows keyword argument
```

As with "args", you can use any name you'd like for keyworded arguments - "kwargs" is just a popular convention.

That's it! Now you should understand how `*args` and `**kwargs` provide the flexibilty to work with arbitrary numbers of arguments!

Functions and Methods Homework

Complete the following questions:

____

****Write a function that computes the volume of a sphere given its radius.****

The volume of a sphere is given as $$\frac{4}{3} πr^3$$

In [1]:
```
def vol(rad):
    pass
```

In [2]:
```
# Check
vol(2)
```

Out[2]:
```
33.49333333333333
```

Write a function that checks whether a number is in a given range (inclusive of high and low)

In [3]:
```
def ran_check(num,low,high):
    pass
```

In [4]:
```
# Check
ran_check(5,2,7)
```

```
5 is in the range between 2 and 7
```

If you only wanted to return a boolean:

In [5]:
```
def ran_bool(num,low,high):
    pass
```

In [6]:
```
ran_bool(3,1,10)
```

Out[6]:
```
True
```

Write a Python function that accepts a string and calculates the number of upper case letters and lower case letters.

```
Sample String : 'Hello Mr. Rogers, how are you this fine Tuesday?'
Expected Output :
No. of Upper case characters : 4
No. of Lower case Characters : 33
```

HINT: Two string methods that might prove useful: **.isupper()** and **.islower()**

If you feel ambitious, explore the Collections module to solve this problem!

In [7]:
```
def up_low(s):
    pass
```

In [8]:
```
s = 'Hello Mr. Rogers, how are you this fine Tuesday?'
up_low(s)
```

```
Original String :  Hello Mr. Rogers, how are you this fine Tuesday?
No. of Upper case characters :  4
No. of Lower case Characters :  33
```

Write a Python function that takes a list and returns a new list with unique elements of the first list.

```
Sample List : [1,1,1,1,2,2,3,3,3,3,4,5]
Unique List : [1, 2, 3, 4, 5]
```

In [9]:
```
def unique_list(lst):
    pass
```

In [10]:
```
unique_list([1,1,1,1,2,2,3,3,3,3,4,5])
```

Out[10]:
```
[1, 2, 3, 4, 5]
```

Write a Python function to multiply all the numbers in a list.

```
Sample List : [1, 2, 3, -4]
Expected Output : -24
```

In [11]:
```
def multiply(numbers):
    pass
```

In [12]:
```
multiply([1,2,3,-4])
```

Out[12]:
```
-24
```

Write a Python function that checks whether a passed in string is palindrome or not.

Note: A palindrome is word, phrase, or sequence that reads the same backward as forward, e.g., madam or nurses run.

In [13]:
```
def palindrome(s):
    pass
```

In [14]:
```
palindrome('helleh')
```

Out[14]:
```
True
```

Hard:

Write a Python function to check whether a string is pangram or not.

```
Note : Pangrams are words or sentences containing every letter of the al
phabet at least once.
For example : "The quick brown fox jumps over the lazy dog"
```

Hint: Look at the string module

In [15]:
```
import string

def ispangram(str1, alphabet=string.ascii_lowercase):
    pass
```

In [16]:
```
ispangram("The quick brown fox jumps over the lazy dog")
```

Out[16]:
```
True
```

In [17]:
```
string.ascii_lowercase
```

Out[17]:
```
'abcdefghijklmnopqrstuvwxyz'
```

Great Job!

Functions and Methods Homework Solutions

Write a function that computes the volume of a sphere given its radius.

In [1]:
```
def vol(rad):
    return (4/3)*(3.14)*(rad**3)
```

In [2]:
```
# Check
vol(2)
```

Out[2]:
```
33.49333333333333
```

Write a function that checks whether a number is in a given range (inclusive of high and low)

In [3]:
```
def ran_check(num,low,high):
    #Check if num is between low and high (including low and high)
    if num in range(low,high+1):
        print('{} is in the range between {} and {}'.format(num,low,high))
    else:
        print('The number is outside the range.')
```

In [4]:
```
# Check
ran_check(5,2,7)
```

```
5 is in the range between 2 and 7
```

If you only wanted to return a boolean:

In [5]:
```
def ran_bool(num,low,high):
    return num in range(low,high+1)
```

In [6]:
```
ran_bool(3,1,10)
```

Out[6]:
```
True
```

Write a Python function that accepts a string and calculates the number of upper case letters and lower case letters.

```
Sample String : 'Hello Mr. Rogers, how are you this fine Tuesday?'
Expected Output :
No. of Upper case characters : 4
No. of Lower case Characters : 33
```

If you feel ambitious, explore the Collections module to solve this problem!

In [7]:
```
def up_low(s):
    d={"upper":0, "lower":0}
    for c in s:
        if c.isupper():
            d["upper"]+=1
        elif c.islower():
            d["lower"]+=1
        else:
            pass
    print("Original String : ", s)
    print("No. of Upper case characters : ", d["upper"])
    print("No. of Lower case Characters : ", d["lower"])
```

In [8]:
```
s = 'Hello Mr. Rogers, how are you this fine Tuesday?'
up_low(s)
```

```
Original String :  Hello Mr. Rogers, how are you this fine Tuesday?
No. of Upper case characters :  4
No. of Lower case Characters :  33
```

Write a Python function that takes a list and returns a new list with unique elements of the first list.

```
Sample List : [1,1,1,1,2,2,3,3,3,3,4,5]
Unique List : [1, 2, 3, 4, 5]
```

In [9]:
```
def unique_list(lst):
    # Also possible to use list(set())
    x = []
    for a in lst:
        if a not in x:
            x.append(a)
    return x
```

In [10]:
```
unique_list([1,1,1,1,2,2,3,3,3,3,4,5])
```

Out[10]:
```
[1, 2, 3, 4, 5]
```

Write a Python function to multiply all the numbers in a list.

```
Sample List : [1, 2, 3, -4]
Expected Output : -24
```

In [11]:
```
def multiply(numbers):
    total = 1
    for x in numbers:
        total *= x
    return total
```

In [12]:
```
multiply([1,2,3,-4])
```

Out[12]:
```
-24
```

Write a Python function that checks whether a passed string is palindrome or not.

Note: A palindrome is word, phrase, or sequence that reads the same backward as forward, e.g., madam or nurses run.

In [13]:
```
def palindrome(s):

    s = s.replace(' ','') # This replaces all spaces ' ' with no space ''. (Fixes
    return s == s[::-1]   # Check through slicing
```

In [14]:
```
palindrome('nurses run')
```

Out[14]:
```
True
```

In [15]:
```
palindrome('abcba')
```

Out[15]:
```
True
```

Hard:

Write a Python function to check whether a string is pangram or not.

```
Note : Pangrams are words or sentences containing every letter of the al
phabet at least once.
For example : "The quick brown fox jumps over the lazy dog"
```

Hint: Look at the string module

In [16]:
```
import string

def ispangram(str1, alphabet=string.ascii_lowercase):
    alphaset = set(alphabet)
    return alphaset <= set(str1.lower())
```

In [17]:
```
ispangram("The quick brown fox jumps over the lazy dog")
```

Out[17]:
```
True
```

In [18]:
```
string.ascii_lowercase
```

Out[18]:
```
'abcdefghijklmnopqrstuvwxyz'
```