Outline:

1. Writing short Python code using functions, loops, lists, numpy arrays, and dictionaries
2. Manipulating Python lists and numpy arrays and understanding the difference between them
3. Introducing the stats libraries scipy.stats and statsmodels

Getting Started with Python

Importing modules

All notebooks should begin with code that imports *modules*, collections of built-in, commonly-used Python functions. Below we import the Numpy module, a fast numerical programming library for scientific computing. Future labs will require additional modules, which we'll import with the same syntax.

`import MODULE_NAME as MODULE_NICKNAME`

In [2]:
```
import numpy as np #imports a fast numerical programming library
```

Now that Numpy has been imported, we can access some useful functions. For example, we can use `mean` to calculate the mean of a set of numbers.

In [3]:
```
my_list = [1.2, 2, 3.3]
np.mean(my_list)
```

Out[3]:
```
2.1666666666666665
```

Calculations and variables

In [4]:
```
# // is integer division
1/2, 1//2, 1.0/2, 3*3.2
```

Out[4]:
```
(0.5, 0, 0.5, 9.600000000000001)
```

The last line in a cell is returned as the output value, as above. For cells with multiple lines of results, we can display results using `print`, as can be seen below.

In [5]:
```
print(1 + 3.0, "\n", 9, 7)
5/3
```

```
4.0
 9 7
```

Out[5]:
```
1.6666666666666667
```

We can store integer or floating point values as variables. The other basic Python data types -- booleans, strings, lists -- can also be stored as variables.

In [6]:
```
a = 1
b = 2.0
```

Here is the storing of a list

In [7]:
```
a = [1, 2, 3]
```

Think of a variable as a label for a value, not a box in which you put the value

In [8]:
```
b = a
b
```

Out[8]:
```
[1, 2, 3]
```

This DOES NOT create a new copy of `a`. It merely puts a new label on the memory at a, as can be seen by the following code:

In [9]:
```
print("a", a)
print("b", b)
a[1] = 7
print("a after change", a)
print("b after change", b)
```

```
a [1, 2, 3]
b [1, 2, 3]
a after change [1, 7, 3]
b after change [1, 7, 3]
```

what if we use b=a.copy()?

ANSWER:

Then when we change `a`, `b` would not change also because `b` does not point to `a` because `b` is a copy when we do `b=a.copy()`.

Tuples

Multiple items on one line in the interface are returned as a *tuple*, an immutable sequence of Python objects. See the end of this notebook for an interesting use of `tuples`.

In [10]:
```
a = 1
b = 2.0
a + a, a - b, b * b, 10*a
```

```
(2, -1.0, 4.0, 10)
```

Out[10]:

`type()`

We can obtain the type of a variable, and use boolean comparisons to test these types. VERY USEFUL when things go wrong and you cannot understand why this method does not work on a specific variable!

In [11]:
```
type(a) == float
```

Out[11]:
```
False
```

In [12]:
```
type(a) == int
```

Out[12]:
```
True
```

In [13]:
```
type(a)
```

Out[13]:
```
int
```

EXERCISE 1: Create a tuple called tup with the following five objects:

- **The first element is a float of your choice**
- **The second element is an integer of your choice**
- **The third element is the difference of the first two elements**
- **The fourth element is the sum of the first two elements**
- **The fifth element is the first element divided by the second element**

- **Display the output of `tup`.**

Answer the following questions below:

- **What is the type of the variable `tup`?**
  - **`tup` is a tuple**
- **What happens if you try and chage an item in the tuple?**
  - **you cannot change an item in the tuple**
  - **"TypeError: 'tuple' object does not support item assignment"**

In [14]:
```
# your code here
ex1_float = 1.5
ex1_int = 1
assert(type(ex1_float) == float)
assert(type(ex1_int) == int)
tup = (ex1_float, ex1_int, ex1_float - ex1_int, ex1_float + ex1_int, ex1_float / ex1_int)
print(tup)
```

```
(1.5, 1, 0.5, 2.5, 1.5)
```

Lists

Much of Python is based on the notion of a list. In Python, a list is a sequence of items separated by commas, all within square brackets. The items can be integers, floating points, or another type. Unlike in C arrays, items in a Python list can be different types, so Python lists are more versatile than traditional arrays in C or other languages.

Let's start out by creating a few lists.

In [15]:
```
empty_list = []
float_list = [1., 3., 5., 4., 2.]
int_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
mixed_list = [1, 2., 3, 4., 5]
print(empty_list)
print(int_list)
print(mixed_list, float_list)
```

```
[]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 2.0, 3, 4.0, 5] [1.0, 3.0, 5.0, 4.0, 2.0]
```

Lists in Python are zero-indexed, as in C. The first entry of the list has index 0, the second has index 1, and so on.

In [16]:
```
print(int_list[0])
print(float_list[1])
```

```
1
3.0
```

What happens if we try to use an index that doesn't exist for that list? Python will complain!

In [17]:
```
print(float_list[10])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/var/folders/ry/zrhyljyj7m310x6q60g91fqr0000gn/T/ipykernel_25030/347628444.py in <module>
----> 1 print(float_list[10])

IndexError: list index out of range
```

You can find the length of a list using the built-in function `len`:

In [18]:
```
print(float_list)
len(float_list)
```

```
[1.0, 3.0, 5.0, 4.0, 2.0]
```
Out[18]:
```
5
```

Indexing on lists plus Slicing

And since Python is zero-indexed, the last element of `float_list` is

In [19]:
```
float_list[len(float_list)-1]
```
Out[19]:
```
2.0
```

It is more idiomatic in Python to use -1 for the last element, -2 for the second last, and so on

In [20]:
```
float_list[-1]
```
Out[20]:
```
2.0
```

We can use the `:` operator to access a subset of the list. This is called slicing.

In [21]:
```
print(float_list[1:5])
print(float_list[0:2])
```

```
[3.0, 5.0, 4.0, 2.0]
[1.0, 3.0]
```

In [22]:
```
lst = ['hi', 7, 'c', 'cat', 'hello', 8]
lst[:2]
```
Out[22]:
```
['hi', 7]
```

You can slice "backwards" as well:

In [23]:
```
float_list[:-2] # up to second last
```
Out[23]:
```
[1.0, 3.0, 5.0]
```

In [24]:
```
float_list[:4] # up to but not including 5th element
```
Out[24]:
```
[1.0, 3.0, 5.0, 4.0]
```

You can also slice with a stride:

In [25]:
```
float_list[:4:2] # above but skipping every second element
```
Out[25]:
```
[1.0, 5.0]
```

We can iterate through a list using a loop. Here's a for loop.

In [26]:
```
for ele in float_list:
    print(ele)
```

```
1.0
3.0
5.0
4.0
2.0
```

What if you wanted the index as well?

Use the built-in python method `enumerate`, which can be used to create a list of tuples with each tuple of the form `(index, value)`.

In [27]:
```
for i, ele in enumerate(float_list):
    print(i, ele)
```

```
0 1.0
1 3.0
2 5.0
3 4.0
4 2.0
```

Appending and deleting

We can also append items to the end of the list using the `+` operator or with `append`.

In [28]:
```
float_list + [.333]
```
Out[28]:
```
[1.0, 3.0, 5.0, 4.0, 2.0, 0.333]
```

In [29]:
```
float_list.append(.444)
```

In [30]:
```
print(float_list)
len(float_list)
```

```
[1.0, 3.0, 5.0, 4.0, 2.0, 0.444]
```

```
Out[30]: 6
```

Now, run the cell with `float_list.append()` a second time. Then run the subsequent cell. What happens?

To remove an item from the list, use `del.`

```
In [31]:
```

```python
del(float_list[2])
print(float_list)
```

```
[1.0, 3.0, 4.0, 2.0, 0.444]
```

You may also add an element (elem) in a specific position (index) in the list

```
In [32]:
```

```python
elem = '3.14'
index = 1
float_list.insert(index, elem)
float_list
```

```
Out[32]: [1.0, '3.14', 3.0, 4.0, 2.0, 0.444]
```

List Comprehensions

Lists can be constructed in a compact way using a *list comprehension*. Here's a simple example.

```
In [33]:
```

```python
squaredlist = [i*i for i in int_list]
squaredlist
```

```
Out[33]: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

And here's a more complicated one, requiring a conditional.

```
In [34]:
```

```python
comp_list1 = [2*i for i in squaredlist if i % 2 == 0]
print(comp_list1)
```

```
[8, 32, 72, 128, 200]
```

This is entirely equivalent to creating `comp_list1` using a loop with a conditional, as below:

```
In [35]:
```

```python
comp_list2 = []
for i in squaredlist:
    if i % 2 == 0:
        comp_list2.append(2*i)
print(comp_list2)
```

```
[8, 32, 72, 128, 200]
```

The list comprehension syntax

```
[expression for item in list if conditional]
```

is equivalent to the syntax

```
for item in list:
    if conditional:
        expression
```

Exercise 2: Build a list that contains every prime number between 1 and 100, in two different ways:

- 2.1 Using for loops and conditional if statements.
- 2.2 Using a list comprehension. You should be able to do this in one line of code. Hint: it might help to look up the function `all()` in the documentation.

```
In [36]:
```

```python
### Your code here
ex2_list_loops = []
for i in range(2, 100):
    for j in range (2, i):
        if i % j == 0:
            break # break and no append
    else: # no break
        ex2_list_loops.append(i)
print(f'{ex2_list_loops=}')

ex2_list_comprehension = [i for i in range(2,100) if all(i % j != 0 for j in range(2, i))]
print(f'{ex2_list_comprehension=}')
```

```
ex2_list_loops=[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
ex2_list_comprehension=[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Simple Functions

A *function* object is a reusable block of code that does a specific task. Functions are commonplace in Python, either on their own or as they belong to other objects. To invoke a function `func`, you call it as `func(arguments)`.

We've seen built-in Python functions and methods (details below). For example, `len()` and `print()` are built-in Python functions. And at the beginning, you called `np.mean()` to calculate the mean of three numbers, where `mean()` is a function in the numpy module and numpy was abbreviated as `np`. This syntax allows us to have multiple "mean" functions in different modules; calling this one as `np.mean()` guarantees that we will execute numpy's mean function, as opposed to a mean function from a different module.

User-defined functions

We'll now learn to write our own user-defined functions. Below is the syntax for defining a basic function with one input argument and one output. You can also define functions with no input or output arguments, or multiple input or output arguments.

```
def name_of_function(arg):
    ...
    return(output)
```

We can write functions with one input and one output argument. Here are two such functions.

In [37]:
```python
def square(x):
    x_sqr = x*x
    return(x_sqr)

def cube(x):
    x_cub = x*x*x
    return(x_cub)

square(5),cube(5)
```

Out[37]:
```
(25, 125)
```

What if you want to return two variables at a time? The usual way is to return a tuple:

In [38]:
```python
def square_and_cube(x):
    x_cub = x*x*x
    x_sqr = x*x
    return(x_sqr, x_cub)

square_and_cube(5)
```

Out[38]:
```
(25, 125)
```

Lambda functions

Often we quickly define mathematical functions with a one-line function called a *lambda* function. Lambda functions are great because they enable us to write functions without having to name them, ie, they're *anonymous*.
No return statement is needed.

In [39]:
```python
# create an anonymous function and assign it to the variable square
square = lambda x: x*x
print(square(3))

hypotenuse = lambda x, y: x*x + y*y

## Same as
# def hypotenuse(x, y):
#     return(x*x + y*y)

hypotenuse(3,4)
```

```
9
```

Out[39]:
```
25
```

Methods

A function that belongs to an object is called a *method*. By "object," we mean an "instance" of a class (e.g., list, integer, or floating point variable).

For example, when we invoke `append()` on an existing list, `append()` is a method.

In other words, a *method* is a function on a specific *instance* of a class (i.e., *object*). In this example, our class is a list. `float_list` is an instance of a list (thus, an object), and the `append()` function is technically a *method* since it pertains to the specific instance `float_list`.

In [40]:
```python
float_list = [1.0, 2.09, 4.0, 2.0, 0.444]
print(float_list)
float_list.append(56.7)
float_list
```

```
[1.0, 2.09, 4.0, 2.0, 0.444]
```

Out[40]:
```
[1.0, 2.09, 4.0, 2.0, 0.444, 56.7]
```

Exercise 3: generated a list of the prime numbers between 1 and 100
In Exercise 2, above, you wrote code that generated a list of the prime numbers between 1 and 100. Now, write a function called `isprime()` that takes in a positive integer N, and determines whether or not it is prime. Return `True` if it's prime and return `False` if it isn't. Then, using a list comprehension and `isprime()`, create a list `myprimes` that contains all the prime numbers less than 100.

In [41]:
```python
def isprime(n):
    if n > 1:
        for i in range(2, n):
            if n % i == 0:
                return False
        return True
    return False

myprimes = [i for i in range(100) if isprime(i)]
print(f'{myprimes=}')
```

```
myprimes=[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Introduction to Numpy

Scientific Python code uses a fast array structure, called the numpy array. Those who have programmed in Matlab will find this very natural. For reference, the numpy documention can be found here.

Let's make a numpy array:

In [42]:
```
my_array = np.array([1, 2, 3, 4])
my_array
```

Out[42]:
```
array([1, 2, 3, 4])
```

In [43]:
```
# works as it would with a standard list
len(my_array)
```

Out[43]:
```
4
```

The shape array of an array is very useful (we'll see more of it later when we talk about 2D arrays -- matrices -- and higher-dimensional arrays).

In [44]:
```
my_array.shape
```

Out[44]:
```
(4,)
```

Numpy arrays are typed. This means that by default, all the elements will be assumed to be of the same type (e.g., integer, float, String).

In [45]:
```
my_array.dtype
```

Out[45]:
```
dtype('int64')
```

Numpy arrays have similar functionality as lists! Below, we compute the length, slice the array, and iterate through it (one could identically perform the same with a list).

In [46]:
```
print(len(my_array))
print(my_array[2:4])
for ele in my_array:
    print(ele)
```

```
4
[3 4]
1
2
3
4
```

There are two ways to manipulate numpy arrays a) by using the numpy module's methods (e.g., `np.mean()`) or b) by applying the function np.mean() with the numpy array as an argument.

In [47]:
```
print(my_array.mean())
print(np.mean(my_array))
```

```
2.5
2.5
```

A `constructor` is a general programming term that refers to the mechanism for creating a new object (e.g., list, array, String).

There are many other efficient ways to construct numpy arrays. Here are some commonly used numpy array constructors. Read more details in the numpy documentation.

In [48]:
```
np.ones(10) # generates 10 floating point ones
```

Out[48]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Numpy gains a lot of its efficiency from being typed. That is, all elements in the array have the same type, such as integer or floating point. The default type, as can be seen above, is a float. (Each float uses either 32 or 64 bits of memory, depending on if the code is running a 32-bit or 64-bit machine, respectively).

In [49]:
```
np.dtype(float).itemsize # in bytes (remember, 1 byte = 8 bits)
```

Out[49]:
```
8
```

In [50]:
```
np.ones(10, dtype='int') # generates 10 integer ones
```

Out[50]:
```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

In [51]:
```
np.zeros(10)
```

Out[51]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

Often, you will want random numbers. Use the `random` constructor!

In [52]:
```
np.random.random(10) # uniform from [0,1]
```

Out[52]:
```
array([0.10436713, 0.51580973, 0.62142133, 0.12798517, 0.83866816,
       0.78720074, 0.3154462 , 0.09434505, 0.48669788, 0.82008574])
```

You can generate random numbers from a normal distribution with mean 0 and variance 1:

```
In [53]: normal_array = np.random.randn(1000)
         print("The sample mean and standard devation are %f and %f, respectively." %(np.mean(normal_array), np.std(normal_array)))
```

```
The sample mean and standard devation are -0.037671 and 0.973565, respectively.
```

```
In [54]: len(normal_array)
```

```
Out[54]: 1000
```

You can sample with and without replacement from an array. Let's first construct a list with evenly-spaced values:

```
In [55]: grid = np.arange(0., 1.01, 0.1)
         grid
```

```
Out[55]: array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

Without replacement

```
In [56]: np.random.choice(grid, 5, replace=False)
```

```
Out[56]: array([0.1, 0.8, 0.9, 1. , 0.3])
```

```
In [57]: np.random.choice(grid, 5, replace=False)
```

```
Out[57]: array([0.2, 0.3, 0.9, 0.4, 0.8])
```

With replacement:

```
In [58]: np.random.choice(grid, 20, replace=True)
```

```
Out[58]: array([0.2, 0.1, 0.8, 0.7, 0. , 0.3, 0.5, 0.6, 0.1, 0.6, 0.4, 0.4, 0. ,
                0.7, 0.6, 1. , 0.4, 0.8, 0.3, 0.6])
```

Numpy supports vector operations

What does this mean? It means that instead of adding two arrays, element by element, you can just say: add the two arrays.

```
In [59]: first = np.ones(5)
         second = np.ones(5)
         first + second # adds in-place
```

```
Out[59]: array([2., 2., 2., 2., 2.])
```

Note that this behavior is very different from python lists where concatenation happens.

```
In [60]: first_list = [1., 1., 1., 1., 1.]
         second_list = [1., 1., 1., 1., 1.]
         first_list + second_list # concatenation
```

```
Out[60]: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

On some computer chips, this numpy addition actually happens in parallel and can yield significant increases in speed. But even on regular chips, the advantage of greater readability is important.

Broadcasting

Numpy supports a concept known as *broadcasting*, which dictates how arrays of different sizes are combined together. There are too many rules to list here, but importantly, multiplying an array by a number multiplies each element by the number. Adding a number adds the number to each element.

```
In [61]: first + 1
```

```
Out[61]: array([2., 2., 2., 2., 2.])
```

```
In [62]: first*5
```

```
Out[62]: array([5., 5., 5., 5., 5.])
```

This means that if you wanted the distribution $N(5, 7)$ you could do:

```
In [63]: normal_5_7 = 5 + 7*normal_array
         np.mean(normal_5_7), np.std(normal_5_7)
```

```
Out[63]: (4.7363057134322535, 6.814955686469278)
```

Multiplying two arrays multiplies them element-by-element

```
In [64]: (first +1) * (first*5)
```

```
Out[64]: array([10., 10., 10., 10., 10.])
```

You might have wanted to compute the dot product instead:

```
In [65]: np.dot((first +1) , (first*5))
```

```
Out[65]: 50.0
```

Exercise 4: Matrix multiplication

Using numpy, create a random 5X5 matrix and multiply is by the 5X5 unit matrix

In [69]:
```
### Your code here
np.random.random((5,5)) @ np.ones((5,5))
```

Out[69]:
```
array([[2.78933138, 2.78933138, 2.78933138, 2.78933138, 2.78933138],
       [3.40900768, 3.40900768, 3.40900768, 3.40900768, 3.40900768],
       [2.33559543, 2.33559543, 2.33559543, 2.33559543, 2.33559543],
       [2.97417337, 2.97417337, 2.97417337, 2.97417337, 2.97417337],
       [2.03232847, 2.03232847, 2.03232847, 2.03232847, 2.03232847]])
```

Probabilitiy Distributions from `scipy.stats` and `statsmodels`

Two useful statistics libraries in python are `scipy` and `statsmodels`.

For example to load the z_test:

In [70]:
```
import statsmodels
from statsmodels.stats.proportion import proportions_ztest
```

In [71]:
```
x = np.array([74,100])
n = np.array([152,266])

zstat, pvalue = statsmodels.stats.proportion.proportions_ztest(x, n)
print("Two-sided z-test for proportions: \n","z =",zstat,", pvalue =",pvalue)
```

```
Two-sided z-test for proportions:
 z = 2.212695207500177 , pvalue = 0.026918666032452288
```

In [72]:
```
#The `%matplotlib inline` ensures that plots are rendered inline in the browser.
%matplotlib inline
import matplotlib.pyplot as plt
```

Let's get the normal distribution namespace from `scipy.stats`. See here for Documentation.

In [73]:
```
from scipy.stats import norm
```

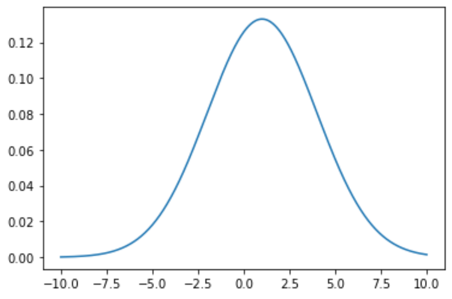
Let's create 1,000 points between -10 and 10

In [74]:
```
x = np.linspace(-10, 10, 1000) # linspace() returns evenly-spaced numbers over a specified interval
x[0:10], x[-10:]
```

Out[74]:
```
(array([-10.        ,  -9.97997998,  -9.95995996,  -9.93993994,
         -9.91991992,  -9.8998999 ,  -9.87987988,  -9.85985986,
         -9.83983984,  -9.81981982]),
 array([ 9.81981982,  9.83983984,  9.85985986,  9.87987988,  9.8998999 ,
         9.91991992,  9.93993994,  9.95995996,  9.97997998, 10.        ]))
```

Let's get the pdf of a normal distribution with a mean of 1 and standard deviation 3, and plot it using the grid points computed before:

In [75]:
```
pdf_x = norm.pdf(x, 1, 3)
plt.plot(x, pdf_x);
```



And you can get random variables using the `rvs` function.

Referencies

A useful book by Jake Vanderplas: PythonDataScienceHandbook.

You may also benefit from using Chris Albon's web site as a reference. It contains lots of useful information.

Dictionaries

A dictionary is another data structure (aka storage container) -- arguably the most powerful. Like a list, a dictionary is a sequence of items. Unlike a list, a dictionary is unordered and its items are accessed with keys and not integer positions.

Dictionaries are the closest data structure we have to a database.

Let's make a dictionary with a course number and their corresponding enrollment numbers.

In [76]:
```
enroll2021_dict = {'CS1': 500, 'CS2': 400, 'Stat1': 300, 'Stat2': 300, 'EE1': 400}
enroll2021_dict
```
Out[76]:
```
{'CS1': 500, 'CS2': 400, 'Stat1': 300, 'Stat2': 300, 'EE1': 400}
```
One can obtain the value corresponding to a key via:

In [77]:
```
enroll2021_dict['CS1']
```
Out[77]:
```
500
```
If you try to access a key that isn't present, your code will yield an error:

In [78]:
```
enroll2021_dict['CS148']
```
```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/var/folders/ry/zrhyljyj7m310x6q60g91fqr0000gn/T/ipykernel_25030/390996035.py in <module>
----> 1 enroll2021_dict['CS148']

KeyError: 'CS148'
```
Alternatively, the `.get()` function allows one to gracefully handle these situations by providing a default value if the key isn't found:

In [79]:
```
enroll2021_dict.get('CS148', 5)
```
Out[79]:
```
5
```
Note, this does not *store* a new value for the key; it only provides a value to return if the key isn't found.

In [80]:
```
enroll2021_dict.get('CS148', None)
```
All sorts of iterations are supported:

In [81]:
```
enroll2021_dict.values()
```
Out[81]:
```
dict_values([500, 400, 300, 300, 400])
```
In [82]:
```
enroll2021_dict.items()
```
Out[82]:
```
dict_items([('CS1', 500), ('CS2', 400), ('Stat1', 300), ('Stat2', 300), ('EE1', 400)])
```
We can iterate over the tuples obtained above:

In [83]:
```
for key, value in enroll2021_dict.items():
    print("%s: %d" %(key, value))
```
```
CS1: 500
CS2: 400
Stat1: 300
Stat2: 300
EE1: 400
```
Simply iterating over a dictionary gives us the keys. This is useful when we want to do something with each item:

In [84]:
```
second_dict={}
for key in enroll2021_dict:
    second_dict[key] = enroll2021_dict[key]
second_dict
```
Out[84]:
```
{'CS1': 500, 'CS2': 400, 'Stat1': 300, 'Stat2': 300, 'EE1': 400}
```
The above is an actual copy of _enroll2021*dict's* allocated memory, unlike, `second_dict = enroll2021_dict` which would have made both variables label the same memory location.

In the previous dictionary example, the keys were strings corresponding to course names. Keys don't have to be strings, though; they can be other *immutable* data type such as numbers or tuples (not lists, as lists are mutable).

Dictionary comprehension:

You can construct dictionaries using a *dictionary comprehension*, which is similar to a list comprehension. Notice the brackets {} and the use of `zip` (see next cell for more on `zip`)

In [85]:
```
float_list = [1., 3., 5., 4., 2.]
int_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

my_dict = {k:v for (k, v) in zip(int_list, float_list)}
my_dict
```
Out[85]:
```
{1: 1.0, 2: 3.0, 3: 5.0, 4: 4.0, 5: 2.0}
```
Creating tuples with `zip`

`zip` is a Python built-in function that returns an iterator that aggregates elements from each of the iterables. This is an iterator of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted. The `set()` built-in function returns a `set` object, optionally with elements taken from another iterable. By using `set()` you can make `zip` printable. In the example below, the iterables are the two lists, `float_list` and `int_list`. We can have more than two iterables.

In [86]:
```
float_list = [1., 3., 5., 4., 2.]
int_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

viz_zip = set(zip(int_list, float_list))
viz_zip
```

Out[86]:
```
{(1, 1.0), (2, 3.0), (3, 5.0), (4, 4.0), (5, 2.0)}
```

In [87]:
```
type(viz_zip)
```

Out[87]:
```
set
```

Exercise 5: Dictionary search

`Given the dictionary `my_dict` that we made earlier, find all odd values, print the keys associated with this set of values, and then assign to the same key the value multiplied by 2.`

In [88]:
```
## Your code here
print(f'{my_dict=}')

for key, value in my_dict.items():
    if value % 2 != 0: # find all odd values
        print(key)
        my_dict[key] = 2*value

print(f'{my_dict=}')
```

```
my_dict={1: 1.0, 2: 3.0, 3: 5.0, 4: 4.0, 5: 2.0}
1
2
3
my_dict={1: 2.0, 2: 6.0, 3: 10.0, 4: 4.0, 5: 2.0}
```