Variables

Variables are symbolic names associated with information, values, which are paired with storage locations on the computer. In other words, if you wish to be able to access a piece of information, you store that information in memory and access it using this symbolic name, the variable, that has been linked to that location.

We assign values to variables using the $=$ opperator.

Variable names:

- They can **only** contain letters, digits, or underscores `_`
- They **cannot** start with digits
- Variable names that start with underscores like `__special` have a special meaning and should be avoided until you understand the convention.

Displaying variable values

You can display these values using the built-in python function, `print`, that prints things as text. To do this:

- Call the function (i.e., tell Python to run it) by using its name
- Provide values to the function (i.e., the things to print) in parentheses
- The values passed to the function are called **arguments**

Accessing variables

Accessing variables

A variable must be given a value or it is "undefined" and cannot be used yet in other operations.

During a single interactive session, or jupyter notebook, variables persist - maintain their most recent assigned value - beyond the cell (individual *code block*) they are defined within.

Another way to say this is that variables defined in this way are *global* in scope. *Scope* is an important concept in programming that we will return to once we have been introduced to the idea of *code blocks*.

Data types

Values in a program will have a specific **type** which determines the types of operations (methods) that can be performed on that value.

There is a built-in method: `type` which can be used to determine the type of a variable.

Common data types

The three examples above cover the three main basic types we will encounter: `int`, `float`, and `string`

strings

- A string is a character array, meaning unlike ints and floats it is not a single value.
    - Although strings are often manipulated as if they were single values, there a few fundamental differences that we will encounter resulting from the non-singular nature of the `string` type.

strings

NOTE: You may have noticed that when working with strings thus far, we have defined them using either single (`'Hello World!'`) or double (`"Hello World!"`) quotes. In some languages, single and double quotes do actually create different data types, but they are interchangeable in Python.

However, the reason to use double (single) quotes some cases is that it allows for single (double) quotes to be included in the text of the string.

Challenge

How would you create a string that included both single and double quotes in the body of the text?

Assigning type in python

You may have noticed that when checking type in earlier examples, the same variable can be assigned values of any of the three types we've dicussed thus far.

So, we see that when we assign a value to a variable in Python that sets the type for that variable **but** that type is *not fixed*.

We can change the type for a variable by assigning it a value of a different type. This is not the case for many languages - in those cases you generally assign the type as part of the variable declaration, e.g. `int a = 5`.

Operations

A value's type determines what operations can be performed on that value. Some of the most basic operations are things like adding (`+`), subtracting (`-`), and multiplying (`*`).

When talking about numbers, all of these operations seem pretty straight forward, and indeed all are possible for `int` and `float` types. `strings` are a bit different.

Addition and multiplication make sense.

Subtraction is less obvious. A string is an array of characters, subtracting one string from another is not a simple process that would always work the same way. For example, how would we interpret `'Hello World!' - 'l'`? There are three `l` characters in the first string, which should we remove? Let's try anyway:

A few more operations

We've touched on some of the more basic operations we can perform on data, including type conversion, but there are many other operations. Not all operations will make sense for every data type, as we've seen in the case of string subtraction.

Division (`/`) also generally only makes sense for number types

- minor caveat: we will later see that number arrays can also have mathematical operations performed on them. *

NOTE: in Python 3, division will always return a float, even when the integer division would not require rounding.

Another useful numerical operation is getting the remainder, this can be done with `%`. This will return the same numerical type as the input: again, if you mix floats and ints the result will be a float.

Combining variables

We've discussed performing operations directly on values of various types, but what about on variables?

In [38]:
```
a = 10
b = a + 5
a = 5
print(b)
```

```
15
```

Will changing the value of one of these variables now change the value of the other?

This may be surprising if you are used to working with spreadsheets instead of programming languages. In a spreadsheet, if we make one cell depend on another, changing the latter will change the former as well. So what's happening here?

When we assign a variable (say `a`) a specific value what are we doing?

We are assigning the name `a` to a location in the computer memory where the value is stored. We can think of `a` as a reference (or pointer) to that location. If we change `a`, we are assigning that name to a *different* location in memory where the new value is stored.

But what are we doing when we say `b = a` or `b = a + 5`?

In the first case, we are assigning `b` to the same location in memory as `a`. In the second case, we are assigning `b` to the location in memory for the *value* resulting from `a + 5`.

NOTE: You can also think of this as: we are putting the value resulting from `a+5` into a location in memory that was previously unused, and `b` is referencing that new location. From our perspective, these are the same thing.

In either case, when we change `a` this will not change `b` because we are not changing the value `b` references, we are only changing the value `a` references.

Mixing types and type conversion

What about performing operations that mix types?

What would it mean to add a number and a string?

For example, how should we interpret `1 + "2"` ? Is it the number `3` or the string `"12"` ?

Converting to/from string

To clear up this ambiguity, we must first convert one of the values to be the same type as the other. Standard types like these can be converted by using the type name as a function:

Converting from string

NOTE: When converting a string, this will only work if the type you are attempting to convert to is valid.

List-like data types

We have actually already worked with one list-like data type - the `string` - which is actually a set of characters that make up that string.

List-like objects contain multiple values, so they have a length. For example, what would the lenght of the string `"Hello World"` be?

Determining length

Because strings have a length, there is a built-in method to determine their length: `len`.

This method actually works on any list-like type, as we will see.

More list-like data types

- strings
- tuples
- lists
- numpy arrays

Let's take a look at how to declare these different types, and what we can put in them. For example, strings are declared using single or double quotes around text: `'my string'` or `"my string"` . Only characters can go in strings, but any symbol or number can be represented as a character, so in principle we can store any type of data as strings, it just won't generally be the most efficient way of doing so. Now we'll explore the other types.

Tuples

From these examples we have a few takeaways:

- Tuples are declared using rounded brackets
- Tuples can be filled with any of the standard data types we have learned about so far, or any mixture of those types. In fact, each element of a tuple can be any type, including tuples or even your own methods or classes.

Lists

From these examples the takaways are:

- Lists are declared using square brackets: `[]`
- Lists can also be filled with any of the standard data types we have learned about so far, or any mixture of those types, etc.

Arrays

In Python, when we talk about arrays, we are talking about a specific implementation: `numpy arrays`. Python does not have a built-in array data type, it uses lists for everything. Lists are very powerful, but because they can contain arbitrary data-types, they are less efficient when it comes to the manipulation of large data sets. Because there is no built in array-type, we hvae to `import` the module/library we wish to use. We will discuss the syntax for importing libraries more in a future section.

A few notes on some new concepts:

- Properties of the array can be accessed with array.property, i.e. `array.shape` or `array.dtype`:
    - This is how properties of any class are accessed - we will discuss classes more in another lesson
    - `dtype`: this is the type for the elements in the numpy array
    - `shape`: this is a tuple giving the number of rows and columns: if we have an *N*-dimensional array, the lenght of the shape tuple is *N* and each element of the shape tuple givens the length of that dimension.

- New types:
    - `int64`: that simply means that these are 64-bit (8-byte) integers, rather than the 4-byte default
    - `float64`: again, 64-bit floats (double precision) rather than 32-bit.
    - new string types: here the `U` indicates `unicode`, which is the particular way of encoding symbols and characters, `U1` is a 1-byte `char` - the usual - and `U32` is a 32-bit (4-byte) `char`. This larger size is chosen when floats are represented by strings because floats can generally be much larger in size than single characters.

These examples also leave us with a few take-aways:

- You create a numpy array with the array() method, giving it the input list of elements as an argument to the method.
- Numpy can take any valid list and convert it to a numpy array, however they do have some unique behavior.
    - If you create a numpy array with a mixture of integers and floats, it will convert all integers to floats. If you create the array with a mixture of strings and numbers, it will convert all elements to strings.
    - Numpy arrays created with a list of lists of the same length will create a multi-dimentional array.
    - Arrays created with a mixture of lists of different lenght and individual numbers will treat each element as an arbitrary `object` type

As we go forward, we will find that many of the numpy array specific methods we encounter only work with arrays of numbers. Generally, it is not useful to use arrays rather than the built-in list type when working with arbitrary object types.

We will now explore some of the methods available for manipulating list-like data types.

Accessing elements

- An item in a list is called an element. Whenever we treat a string as if it were a list, the string's elements are its individual characters.
- This works the same way for all list-like data types.
- Elements of a list-like type are accessed using their index (numerical position in the list).
- The list index starts at `0` (unlike in MatLab, or Fortran, where it starts at `1`).
- You can access elements from the end of the list using negitive integers: `-1` is the last element, `-2` second to last, etc.

Slicing

- A slice is a part of a a list-like thing.
- We take a slice by using [start:stop], where start is replaced with the index of the first element we want and stop is replaced with the index of the element just after the last element we want.
- Mathematically, you might say that a slice selects [start:stop).
- The difference between stop and start is the slice's length.
- Taking a slice does not change the contents of the original list-like object. Instead, the slice is a copy of part of the original.
- In the case of strings, a part of a string is called a substring. A substring can be as short as a single character.

- What do you think `a[:4]` will do?
- What does `a[2:]` do?
- What does `a[3:-2]` do?
- What about `a[:]`?

How do we slice multidemensional arrays?

First, **how do we access elements?** Based on the way the `shape` is reported, it's not a bad guess to assume that we can access the row and column index using two comma-separated indices.

What will these return:

- `a[0]`?
- `a[0, 1]`?
- `a[1, 2]`?

Slicing will work the same way:

Modifying List-like types

We will now take a minute to explore how to modify or add to the values stored in list-like data types.

We have seen how to access elements from list-like types. **Can we change them?**

Let's try lists, strings, and tuples.

To explore why we are able to assign new values to individual 'items' of the list, but not strings or tuples, lets step back and look again at some of the features of variable assignment we've already discussed.

When discussing types initially we learned about how to add two strings together. We can do the same with tuples and lists:

Now, in our discussion of creating new variables that depend on previously defined variables, following the example above, we learned that once you define `b`, you can change `a` and it will not change `b`. Let's try a few examples of this with list-like objects.

So far so good, but in all of these cases we set `a` equal to a new value, meaning we changed the location in memory that `a` is referencing. However, we learned above that we can change individual elements of a list, **what happens if we only change part of `a` ?**

Mutable and Immutable types

In the example above, when we changed only a part of list `a`, we actually changed the values being stored in the location in memory that `a` references. The line `b = a` created the variable `b` as a reference to the *same memory location* as `a`. So, when we change the values being stored at that memory location using `a` as the reference, `b` also changes. This is a fundamental property of **mutable** data types. Lists are **mutable**, meaning that the data the list, or any mutable type, stores can be changed.

Numbers are an example of an **immutable** type, meaning you cannot change the data. This seems obvious in this case. If you say `a = 2`, so `a` is a reference to the value `2`, you cannot change `2`, `2` is always `2`. All you can do is change `a` to reference a different value. This is also true for strings and tuples: they **immutable**. This is why even though we can access individual elements of strings and tuples, we cannot change those elements. Once a string or tuple is created, that string or tuple value cannot be changed. You instead have to tell your variable to reference a new string value.

So how do we make variables with mutable values independent?

Here we used the `list` function to make a copy of list `a` - a new memory location - that the new variable `b` is now references. So now when we change `a` that does not change `b`: they are not referencing the same memory location in this case.

Bonus question: We said that to change a variable with an immutable value, we have to set it equal to a different value, meaning it now references a different memory location on the

computer. What happens to the old value that no-longer has a variable to reference it?

Functions and Methods

A function is a block of code which only runs when it is called. You can pass data, into a function and a function can return data as a result.

Built-in Functions

We've already seen several examples of functions: `print`, `len`, `type`, and even the `int`, `float`, and `string` functions that changed an existing value from one type to another. These are all built-in functions (meaning they are part of the basic Python interpreter and do not require the `import` of a particular library) that apply broadly, although `len` only applies to list-like types.

Here are a few general rules about functions:

- an *argument* is a value passed into a function
- Functions can take zero arguments
- To call a function, you must use round brackets, even if you give no arguments, so that Python knows a function is being called: `(<argument>)`
- Functions can take multiple arguments, separated by commas: `function(arg1, arg2, ...)`
- Functions always return a value. If no return value is provided within the function code block, it will return None

Methods

Methods are a particular type of function which is associated with a specific object (or class) and can only be called on that object: in other words methods are functions that belong to objects. Methods also have access to the data stored by the object, and in some cases can be used to modify that data, while in general Functions cannot modify the data stored by an object.

List-like type methods

All of the built-in list-like types in Python have a few fundamental methods in common:

- `count()`: returns the number of times a specified value occurs
- `index()`: returns the index for the first element with the specified value

Because tuples are immutable, these are the only methods available. These methods access the data stored in the tuple, but do not attempt to modify that data. Strings are also limited, but many text-editing related methods exist for string objects, the key point is that they all *return a new string*.

For example, the `capitalize()` method capitalizes the first letter of a string, which sounds like it changes the string, let's give it a try:

List methods

Lists are mutable, so there are many more things we can do with lists to manipulate their data in useful ways.

- `append()`: adds single values to the end of a list
- `extend()`: adds the elements of any list-like type to the end of the current list
- `sort()`: sorts the list
- `pop()`: removes the element at the specified position

And more. In Jupyter, or iPython, you can see all of the available methods by typing the name of your list variable followed by a dot to indicate that you want to use a list method and hitting tab.

Numpy functions

In data analysis, a helpful feature of numpy arrays is that common mathematical operations can be performed on them and the operation will be done element-by-element without you as the programmer having to access each element directly.

The Numpy library is also contains many functions that allow us to perform a variety of calculations on the array or on only one dimension (axis) of the array, allowing for rapid manipulation of large data sets.

One more list-type:

Pandas DataFrame

Creating functions

We will encounter more advanced features of functions as we go forward, but the basic syntax for defining a function is:

```
def <function name>(<parameters>):
    body
```

- The function name, like variable names, can be anything.
- You indicate that you are defining a function with `def`.
- The colon at the end of the first line indicates the start of a *block* of code, in this case the code block that determines what will happen when the function is used.

When definiting the function, we think of inputs as **parameters** used by the function. When we use (call) the function we give it **arguments**. We make this distinction because the parameters within the body of the function do not have a set type or value - they are placeholders for the particular values that will be used any time the function is used. These parameters also have names that are *local* in scope.

What is scope?

Scope describes the region of your code where a variable is defined. A global variable is one accessible anywhere in the code. There are also local variables, those only defined within a code-block. For example, variables defined within functions can only be accessed in that function.

In [ ]:

```
def my_function(my_name):
    p = "This is {}'s function!".format(my_name)
    print(p)

a = "Dr. Hanks"
my_function(a)
print(p)
```

Repeated actions and conditions

”For” loops

A *For loop* allows you to execute a command once for each value in a collection.

- Doing calculations on the values in a list one by one requires writing out the calculation for as many values as there are in the list.
- A for loop tells Python to execute some statements once for each value in a list, a character string, or some other collection.
- “for each thing in this group, do these operations”

In [101]:
```
a = [1,2,3,4,5]
print(a[0])
print(a[1])
print(a[2])
print(a[3])
print(a[4])
```

```
1
2
3
4
5
```

Or I can do this in one loop!

In [102]:
```
for number in a:
    print(number)
```

```
1
2
3
4
5
```

The `for` loop consists of a collection, a loop variable, and a body.

```
for <loop variable> in <collection>:
    body
```

In the above example

- `number` is the loop variable: this is what changes for each iteration of the loop
- `a` is the collection that the loop runs over
- `print(number)` is the body: this specifies what to do for each value in the collection

Loop syntax

The first line of the `for` loop must end with a colon and the body must be indented. This syntax is not unique to `for` loops, but is the basic syntax for code blocks:

- The colon at the end of the first line signals the start of a block of statements.
- Python uses indentation rather than {} or begin/end to show nesting.
    - Any consistent indentation is legal, but almost everyone uses four spaces.
    - Indents are always meaningful

In [103]:
```
for number in a:
print(number)
```

```
  File "<ipython-input-103-463e2dfe2066>", line 2
    print(number)
        ^
IndentationError: expected an indented block
```

In [104]:
```
for number in [2, 3, 5]
    print(number)
```

```
  File "<ipython-input-104-a0e1f235821e>", line 1
    for number in [2, 3, 5]
                           ^
SyntaxError: invalid syntax
```

In [105]:
```
a = [0,1,2]
    b = [3,4,5]
```

```
  File "<ipython-input-105-12d18f9defc7>", line 2
    b = [3,4,5]
    ^
IndentationError: unexpected indent
```

Loop syntax cont'd

- Loop variables can be called anything, they are created on demand and meaningless - except to you and others reading your code
- The body of the loop can contain many statements - the end of the loop is indicated by the first line of code that is not indented to the same level.
- The loop variable and any variables created inside the for loop have *global* scope

NOTE: In many languages, loops and conditionals (for, while; if, else) also act as scope blocks. This is not the case in Python. In Python, scope is set by functions, classes, and modules.

For example:

In [106]:
```
for number in a:
    print(number)
    number_mod = number + 5
    print(number_mod)

print("The for loop has finished at {},{}".format(number, number_mod))
```

```
1
6
2
7
3
8
4
9
5
10
The for loop has finished at 5,10
```

range

You can iterate over a sequence of numbers using the `range` function, which has a few options for input arguments:

- `range(N)` will iterate from `0` to `N-1`
- `range(m,n)` will iterate from `m` to `n-1`
- `range(m,n,s)` will iterate from `m` to `n-1` with a step size of `s` (the default is 1)

In [107]:
```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

In [108]:
```
for i in range(2,5):
    print(i)
```

```
2
3
4
```

In [109]:
```
for i in range(1,10,3):
    print(i)
```

```
1
4
7
```

Conditionals: if/else

If we want to take a different action depending on some condition, we can do this with an `if` statement. We give the `if` statement a condition which will return a boolean (True/False) depending on whether the condition is passed or not. If the condition returns `True` the code block within the `if` statement will be exectued, if it returns `False`, the code block will be skipped.

In [110]:
```
numbers = [1,2,3,4,5]
for a in range(5):
    if a > 2:
        print(a)
```

```
3
4
```

`else` allows you to apply some condition such that one code block will be executed if the condition is passed and another block of code will be executed. Only one or the other block will ever execute.

In [111]:
```
numbers = [1,2,3,4,5]
for a in range(5):
    if a < 3:
        print('less')
    else:
        print('greater')
print('done')
```

```
less
less
less
greater
greater
done
```

Finally, we can chain multiple conditions together with `elif`:

In [112]:
```
for a in range(5):
    if a < 3:
        print('less')
    elif a == 3:
        print('equal')
    else:
        print('greater')
print('done')
```

```
less
less
less
equal
greater
done
```

NOTE: when testing if two values are equal we use `==` rather than `=`, which is used to assign values to variables.

We can also use `and` and `or` to to combine conditions. These work like logical and and or statements, a statement with `and` will hold true only if both conditons are true and a statement with `or` will be true if either statement is true.

In [113]:
```
numbers = [0,1,2,3,4,5]
for a in range(5):
    if (a >= 1) and (a <= 3):
        print("{} is in the range 1 to 3".format(a))
    else:
        print("{} is either less than 1 or greater than 3".format(a))
```

```
0 is either less than 1 or greater than 3
1 is in the range 1 to 3
2 is in the range 1 to 3
3 is in the range 1 to 3
4 is either less than 1 or greater than 3
```

In [114]:
```
for a in range(5):
    if (a < 1) or (a > 3):
        print("{} is outside the range 1 to 3".format(a))
    else:
        print("{} is in the range 1 to 3".format(a))
```

```
0 is outside the range 1 to 3
1 is in the range 1 to 3
2 is in the range 1 to 3
3 is in the range 1 to 3
4 is outside the range 1 to 3
```

Notes on syntax

- We have been throwing around some comparison opperators which I will summarize here:
    - `<` - less than
    - `>` - greater than
    - `<=` - less than or equal to
    - `>=` - greater than or equal to
    - `==` - equal to
- When using multiple conditions with `and` or `or`, you should isolate each condition with parentheses.

"While" loops

Like for loops, *while loops* allow you to execute the same command many times. In this case, the continued exectution of the command is dependent on a *condition*.

In [115]:
```
count = 0
while count < 5:
    print(count)
    count += 1
```

```
0
1
2
3
4
```

Can you think of any dangers with using `while` loops?
Hint: what would happen if you made your while loop with `while True:` ?

Why might you want to use a `while` loop rather than a `for` loop?

break

You can exit a `while` loop early with the `break` command. This is how we can save ourselves from inifite `while` loops!
NOTE: You can also use `break` in `for` loops, it will do the same thing - exit the loop.

In [116]:
```
count = 0
while count < 10:
    print(count)
    if count > 5:
        break
    count += 1
```

```
0
1
2
3
4
5
6
```

continue

You can skip the current block of code and return the start of the `for` or `while` loop with `continue`:

In [117]:
```
for a in range(10):
    if a % 2 == 0:
        continue
    print(a)
```

```
1
3
5
7
9
```

else with for and while

You can add an `else` statement following your `for` or `while` loop which will execute if the `for` or `while` condition is not met just like it does with `if` statements.

What is the condition for a `for` loop?

`for` loops move through elements of a collection using iteration, creating an iterator that steps through the collection until the end of the collection is reached. The implicit condition being evaluated in that process is that the end of the collection has not been reached. The `for` loop terminates when that condition is no-longer met, i.e. when the end of the collection is reached.

The `else` statement will not be exectued if you `break` out of the loop early, because the loop condition was always met, you simply left the loop. This can be especially useful in debugging, evaluating idosyncratic code behavior, or if you want to do something different when the loop reaches the end normally rather than ending early under your `break` condition.

For loops revisited

We've looked at examples of `for` loops using basic numerical lists and the `range` function. But I told you that we can use any type of collection, so we'll now try a few more examples.

In [118]:
```
names = ["Rob","Bob","Bill","Will"]
for name in names:
    print(name)
```

```
Rob
Bob
Bill
Will
```

In [119]:
```
my_collection = ["Hello",5,7.5,["Good","bye"]]
for element in my_collection:
    print(element)
```

```
Hello
5
7.5
['Good', 'bye']
```

What if we wanted to also keep track of the index of the current element as we loop through them?

Well, we could create a counter:

In [120]:
```
names = ["Rob","Bob","Bill","Will"]
i = 0
for name in names:
    print("{} is number {}".format(name,i+1))
    i += 1
```

```
Rob is number 1
Bob is number 2
Bill is number 3
Will is number 4
```

Python as a built in function that can save us some work (and run time) called `enumerate`:

In [121]:
```
names = ["Rob","Bob","Bill","Will"]
for i,name in enumerate(names):
    print("{} is number {}".format(name,i+1))
```

```
Rob is number 1
Bob is number 2
Bill is number 3
Will is number 4
```

How is this working?

`enumerate` is a function that takes the collection you want to iterate over as an argument. It returns an enumerate object which can then be iterated over, returning a tuple of the index and value at that iteration. You can also make a list of those tuples.

In [122]:
```
names = ["Rob","Bob","Bill","Will"]
list(enumerate(names))
```

Out[122]:
```
[(0, 'Rob'), (1, 'Bob'), (2, 'Bill'), (3, 'Will')]
```

In [123]:
```
for el in enumerate(names):
    print(el)
```

```
(0, 'Rob')
(1, 'Bob')
(2, 'Bill')
(3, 'Will')
```

Finally, there is an additional optional argument: the starting value for the index counter - this does not change where in the list you start, it just sets an offset for the index value.

In [124]:
```
names = ["Rob","Bob","Bill","Will"]
for i,name in enumerate(names,10):
    print("{} is number {}".format(name,i))
```

```
Rob is number 10
Bob is number 11
Bill is number 12
Will is number 13
```

List comprehension

In Python in particular, `for` loops are a relatively inefficient process. The more *pythonic* way to do what is typically done in a `for` loop is to use **list comprehension**.

The equivalent code for:

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

is

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

In [125]:
```
[print(name) for name in names if len(name)>3]
```

```
Bill
Will
```

Out[125]:
```
[None, None]
```

In [126]:
```
import pandas as pd
```

In [127]:
```
df = pd.DataFrame(names)
df
```

Out[127]:

	0
0	Rob
1	Bob
2	Bill
3	Will

In [128]:
```
data = {'Name':['Rob', 'Bob', 'Bill', 'Will'],
        'Age':[20, 21, 19, 18],
        'Year':['Freshman','Freshman','Sophomore','Junior']}

df = pd.DataFrame(data)
print(df)
```

```
   Name  Age       Year
0   Rob   20   Freshman
1   Bob   21   Freshman
2  Bill   19  Sophomore
3  Will   18     Junior
```

How do we select certain columns?

In [129]:
```
print(df[['Name', 'Year']])
```

```
   Name       Year
0   Rob   Freshman
1   Bob   Freshman
2  Bill  Sophomore
3  Will     Junior
```

What about specific rows?

In [130]:
```
print(df.loc[0])
```

```
Name          Rob
Age            20
Year     Freshman
Name: 0, dtype: object
```

NOTE: `loc` is accessing an entry by the *index* label, not the actual numerical index for that entry.

In [131]:
```
df = df.set_index('Name')
print(df)
```

```
      Age       Year
Name
Rob    20   Freshman
Bob    21   Freshman
Bill   19  Sophomore
Will   18     Junior
```

In [132]:
```
print(df.loc['Rob'])
```

```
Age           20
Year    Freshman
Name: Rob, dtype: object
```