Python_Variable_Review

January 26, 2023

1 Python Variable Review

In this notebook, some of the code cells are blank and are for you to enter code. Others already have code in them. Make sure you run these as you go, even if it doesn't explicitly say to do so.

1.1 Everything is an object

1.1.1 Objects in Python

In Python, everything is an object. An object is a single unique unit of information that occupies a chunk of computer memory (the amount of memory depends on the object).

Objects are generally created by assigning them a name. Like this:

[1]:
```
anObject = 42
```

After you run this cell (run it now!), there is an object in memory that has the name `anOject` assigned to it and the object's value 42.

Objects are like fancy museum display cases. Think about the display case of, for example, the U.S. Declaration of Independence. It has the area beneath clear glass that displays the document itself, but there are also all kinds of fancy bits to it, like lights and sensors and stuff.

Our object's value, 42, is the thing (the noun) that is "on display", which we can see by typing a name currently assigned to it. Type `anObject` in the cell below and run the cell.

[2]:
```
anObject
```

[2]:
```
42
```

The object comes with other built-in nouns and even some verbs too. One of the most fundamental *properties* (nouns) of an object is its *type*. We can get the the type with the `type()` function, which takes one argument, the name of the object. Use the `type()` to get the type of our object in the cell below.

[4]:
```
type(anObject)
```

[4]:
```
int
```

The verbs are accessed by using a period (`.`) after the objects current name and then typing the verb. One of the verbs is getting the bit length – how many "bits" (a contraction of BInary digITS

coined by John Tukey) it takes to represent the value in binary notation. Lets, see, for 42… 2, 4, 8, 16, 32, 64, and done! So it should take 6 bits to represent 42 in binary. Let's see using the *method* (verb) `bit_length()`:

[5]:
```
anObject.bit_length()
```

[5]:
```
6
```

Were we right? Try some other numbers!

1.1.2 Objects and Names

Objects and their names are not the same! Notice that above we said "… *a* name currently assigned…" and "… the object's current name…" A single object can have more than one name.

[6]:
```
notAnotherObject = 42
```

Running that cell does *not* create a second object containing 42. Rather, the "42" object that already exists in memory is simply assigned a second name!

Another *property* (noun) that all objects have is a unique identification number. The ID number is independent of any names that may be assigned to the object!

Python has an `id()` function that we can use this to see what names are referring to what objects. Like `type()`, it takes one argument, a name of an object. Try it to get the id number of `anObject`.

[7]:
```
id(anObject)
```

[7]:
```
4334224912
```

Now use it get the id number of `notAnotherObject`.

[8]:
```
id(notAnotherObject)
```

[8]:
```
4334224912
```

See? The two names refer to the same object, which saves on memory!

If we do something like this:

[9]:
```
anObject = 11
```

Now an new object storing the value 11 *has* been created, and the name "anObject" now goes to 11. Let's check the id numbers of our two names now, and warm up our `print()` muscles while doing so!

[15]:
```
print(f"The ids are {id(anObject)} for anObject \
and {id(notAnotherObject)} for notAnotherObject")
```

```
The ids are 4334223920 for anObject and 4334224912 for notAnotherObject
```

Aside `print()` is awesome - it is useful for quick & dirty debugging

"f-strings" are also awesome, and relatively new to Python. You make an f-string just like a string, except you put an "f" in front of the opening quote mark.

An f-string lets you directly enclose an expression (like `id(anObject)` in the string enclosed in curly braces, and python will compute the expression, convert it to a string, and pop it right into the f-string!

The backslash (""), called the"line continuation character" in this context, tells Python that you want to continue the current line on the next line. You generally do *not* need to use it inside `lists` and such, but you do need it above.

Try running the `print()` above without the `\`.

Now lets make a list (see below), but with each list element on a new line:

[16]:
```
aList = [1,
         3,
         5]
aList
```

[16]:
```
[1, 3, 5]
```

No "" needed!

Let's make a new variable "eleven" holding the value 11:

[17]:
```
eleven = 11
```

[19]:
```
whos
```

```
Variable           Type    Data/Info
-----------------------------------
aList              list    n=3
anObject           int     11
eleven             int     11
notAnotherObject   int     42
```

We have now created a few variables in this Python session, so… **Quick quiz!**

How many names do we currently have? 4

How many unique objects do we currently have? 3

1.2 Data Types

As we've seen objects in Python have a type (that describes themselves, not the kind of other objects they are attracted to). Later on, we'll learn that we can create our very own custom types but, for now, let's take a look at the common built-in data types.

Remember that all objects you create by naming them will have both *attributes* or *properties* (their nouns) and *methods* (their verbs), but the particular nouns and verbs they have will depend on their type.

Obviously, one thing that varies with type is literally type.

There is a function corresponding to each data type that converts its argument to that data type. The function name is the same as the type as returned by `type()`. Thus, from the "int" from above tells us that there is a function `int()` that will convert things to integers if it can.

1.2.1 Booleans (bool)

A Boolean variable (named for the mathematician and logician George Boole) can only have the values True and False.

[20]:
```
aBool = True
```

[21]:
```
anotherBool = False
```

Check the type of these variables.

[33]:
```
type(aBool)
```

[33]:
```
bool
```

[34]:
```
type(anotherBool)
```

[34]:
```
bool
```

Note that this also tells us that there is a `bool()` function that will convert things to Boolean.

In the cell below, create several Boolean variable, some True, some False.

[26]:
```
a = True
b = False
c = False
d = True
```

Now, in the next code cell, examine the id numbers of the objects corresponding to your names.

[35]:
```
id(a)
```

[35]:
```
4340673416
```

[36]:
```
id(b)
```

[36]:
```
4340672784
```

[37]:
```
id(c)
```

[37]:
```
4340672784
```

[38]:
```
id(d)
```

[38]:
```
4340673416
```

In this markdown cell , describe what's going on here in your own words. What's going on with your Boolean variables/objects?

There are only two unique objects for the Boolean type. So, regardless of the multiple names assigned there are only two id numbers.

Booleans values map to the values of other variables and vice versa. Let's explore this with the two conversion function `int()` and `bool()`.

In the cell below, convert to two possible Boolean values to integers, and convert both zero and some positive and negative non-zero values to Boolean.

[41]:
```
int(True)
```

[41]:
```
1
```

[42]:
```
int(False)
```

[42]:
```
0
```

[43]:
```
bool(0)
```

[43]:
```
False
```

[44]:
```
bool(2)
```

[44]:
```
True
```

[45]:
```
bool(-2)
```

[45]:
```
True
```

In this cell, describe the mapping of Booleans to integers and vice versa.

We actually use Boolean values and Boolean logic quite a bit in programming. And, of course, all information on computers is ultimately in the form of bits, so all computers are ultimately Boolean processing machines.

1.2.2 Integers (int)

Integers are all the numbers without a fractional component which, in practice, means all numbers without a decimal point. As we saw above, creating a variable using an integer automatically creates an object of type `int`.

Create another integer variable now in the cell below and check it's type.

[49]:
```
anInt = 8
print(anInt)
type(anInt)
```

```
8
```

[49]:
```
int
```

Now create another variable with the same value, except put a ".00" after it. So if you integer above was 3, make this new variable equal to 3.00.

[51]:
```
anotherVar = 8.00
```

Now check the type of your new variable:

[52]:
```
type(anotherVar)
```

[52]:
```
float
```

1.2.3 Floating Point Numbers (float)

Floating point numbers are the computer implementation of the real numbers (remember the good ol' number line from school?). They are so named because, unlike a fixed point number, the decimal place is allowed be anywhere in the number, or "float". For example, 1.234, 12.34, and 123.4 are allowed floating point number but, in a fixed point system, only 12.34 would be allowed. In some applications, fixed point numbers have advantages, but we don't have to worry about them, it's just good to know where the name comes from!

Floating point values always have at least 1 digit on either side of the decimal point. Going in other direction, the largest or smallest number you can represent with a `float` (or the largest `int` you can have) is determined by your computer's hardware and operating system.

1.2.4 Character Strings (str)

Character strings (or just "strings") contain "characters", which are basically the things that you could possible create with your keyboard.

Strings are enclosed in quotes. Run the code cell below, and this notebook will say hello to you.

[53]:
```
aStr = "Hello!"
aStr
```

[53]:
```
'Hello!'
```

Anything enclosed in quotes is a string, even numbers and operators (like "+").

Make a variable containing a number (e.g. `e_num = 2.7182`), and make another using the same number in quotation marks (e.g. `e_str = '2.7182'`), and check their types:

[56]:
```
d_num = 3.8282
type(d_num)
```

[56]:
```
str
```

[57]:
```
d_str = '3.8282'
type(d_str)
```

[57]:
```
str
```

Python is string-friendly. Multiple strings can be concatenated just by using the plus sign ("+").

Add the string to itself (e.g. `e_str + e_str`):

[58]:
```
d_str + d_str
```

[58]:
```
'3.82823.8282'
```

Add the number to itself:

[59]:
```
d_num + d_num
```

[59]:
```
7.6564
```

Now try adding the string to the number and vice versa:

[61]:
```
d_str + d_num
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [61], line 1
----> 1 d_str + d_num

TypeError: can only concatenate str (not "float") to str
```

[62]:
```
d_num + d_str
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [62], line 1
----> 1 d_num + d_str

TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

(Note that the errors differ.)

A string is actually the first object we're talking about here that is actually a collection of other things. This means we can get the individual things via *indexing* using square brackets:

[63]:
```
aStr[0]
```

[63]:
```
'H'
```

But what are the things? That is, what are the elements of a string in Python? In the code cell below, see what type the `aStr[0]` (the "H" in "Hello!" is.

[105]:
```
type(aStr[0])
```

[105]:
```
str
```

Use this cell to explain what this tells you about what multi-element strings are.

I think this indicates that every element is a string in itself, and that the multi-element string is just a bunch of singular element strings put together. I'm not completely sure though.

1.2.5 Tuples (tuple)

Tuples are a collection of objects assigned to a single name. Like this:

[65]:
```
myTup = (9, 8, 7)
myTup
```

[65]:
```
(9, 8, 7)
```

You create it with parentheses (that's what tells Python that you are making a tuple).

In Python, we access the individual items of all multi-element objects using square brackets. Like this:

[66]:
```
myTup[0]
```

[66]:
```
9
```

Recall that we can get multiple things using the colon:

[67]:
```
myTup[0:2]
```

[67]:
```
(9, 8)
```

Tuples can contain objects of different types. Python doesn't care:

[68]:
```
myTup = (3, '4', 'five', True)
myTup
```

[68]:
```
(3, '4', 'five', True)
```

One thing you *can't* do is mess with a tuple. Try changing the 3rd value ('five') to 5:

[69]:
```
myTup[2] = 5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [69], line 1
----> 1 myTup[2] = 5

TypeError: 'tuple' object does not support item assignment
```

Why would you want to use a tuple if you can't change the values? If you are working with actual data, for example, you *never* want *anyone* changing the values, so perhaps a tuple is in order.

Also, for large "hungry" applications, tuple are faster and more memory-efficient than lists.

1.2.6 Lists (list)

Lists are much like tuples except that we can change the individual elements after the list has been created (as much as we want). Lists are created with square brackets. Like this:

[73]:
```
myList = [True, 'Hey', 7, 4.5]
myList
```

[73]:
```
[True, 'Hey', 7, 4.5]
```

Try changing one of these values in the cell below:

[75]:
```
myList[0] = False
myList
```

[75]:
```
[False, 'Hey', 7, 4.5]
```

1.2.7 Dictionaries (dict)

The most complicated (in some sense) type in Python is the dictionary.

Dictionaries are perhaps poorly named, but they are incredibly useful. A real world dictionary is pretty specific; it yields definitions of words and that's it.

Python dictionaries are much more general and flexible. They allow you to store many elements of any type of data and let you access the data by name. If you've programmed in another language, dictionaries are essentially the same as structs (structures).

Here's a dictionary; notice the curly braces.

[76]:
```
myDict = {'name': 'Mary',
          'age': 21,
          'school': 'UT_Austin',
          'gpa': 3.89}

myDict
```

[76]:
```
{'name': 'Mary', 'age': 21, 'school': 'UT_Austin', 'gpa': 3.89}
```

Dicts are a nice way to store a variety of information about a single logical entity, in this case a hypothetical student named "Mary".

Each entry in the dict is a key:value pair. The keys are strings, and the values can be anything.

We retrieve the information using the keys. Like this:

[77]:
```
myDict['name']
```

[77]:
```
'Mary'
```

Here's a dict to store hypothetical data from an experiment along with "metadata" about the experiment:

[78]:
```
expDict = {'date': '10July2023',
           'time': '1620CST',
           'experimenter': 'FGauss',
           'data': [245.3, 232.9, 238.6, 222.2, 250.1]}
```

Notice that one of the elements in the dict is a list! If we grab it, it's just a list like any other:

[79]:
```
just_a_list = expDict['data']
just_a_list
```

[79]:
```
[245.3, 232.9, 238.6, 222.2, 250.1]
```

The more you hang around Python, the more you realize how useful dicts are.

1.3 Mutability

Each type of Python object can either be mutable, meaning that the object itself can be modified, or immutable, meaning that, once created, the object cannot be changed in any way.

1.3.1 Immutable object types

The common immutable object types are

- bool
- int
- float
- str
- tupple

If you create an immutable object, like a `str`, by running the cell below…

[80]:
```
immStr = 'You cannot change me!'
immStr
```

[80]:
```
'You cannot change me!'
```

You can't change it. Try changing the 3rd element (the element at index 2) to an 'o' to spell 'You' as 'Yoo'.

[82]:
```
immStr[2] = 'o'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [82], line 1
----> 1 immStr[2] = 'o'

TypeError: 'str' object does not support item assignment
```

Consider the following (run the cells in turn):

[83]:
```
immStrCopy = immStr
```

`immStrCopy` and `immStr` both now refer to the same object.

[84]:
```
immStr = 'You cannot change me either!'
```

`immStr` now refers to a newly created object; `immStrCopy` still points to the original unchanged string.

Verify this in the code cell below using the values and ID numbers of the two strings.

[86]:
```
print(immStr)
id(immStr)
```

```
You cannot change me either!
```

[86]:
```
4401070208
```

[88]:
```
print(immStrCopy)
id(immStrCopy)
```

```
You cannot change me!
```

[88]:
```
4396201296
```

1.3.2 Mutable objects

Mutable objects *can* be changed. This can produce surprising results if you don't appreciate the distinction between mutable and immutable objects.

Common mutable objects are lists and dictionaries.

Consider the following in contrast to the immutable example using strings above.

[89]:
```
mList = ['you', 'can', 'change', 'me']
mListCopy = mList
```

Now change the first (zero index) value of `mList` to 'anyone'.

[90]:
```
mList[0] = 'anyone'
```

Use the code cell below to check the values and ID numbers of `mList` and `mListCopy`

[94]:
```
print(mList)
id(mList)
```

```
['anyone', 'can', 'change', 'me']
```

[94]:
```
4401482688
```

[95]:
```
print(mListCopy)
id(mListCopy)
```

```
['anyone', 'can', 'change', 'me']
```

[95]:
```
4401482688
```

Use this cell to explain how this behavior differs from that of immutable objects.

With the immutable objects (the strings) the copy still pointed to the original unchanged string, but with the lists above, which are mutable, the copy changed as well.

1.4 Inquiring minds want to know

There are some functions that allow us to get information about our objects.

1.4.1 dir()

The `dir()` (for directory) function give us all the verbs for an object.

In the code cell below, make a sting variable containing your name in all lowercase. Then 'dir' it (e.g. `dir(yourNameVar)`).

[97]:
```
myName = 'marifer'
dir(myName)
```

[97]:
```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
```

```
'__getnewargs__',
'__gt__',
'__hash__',
'__init__',
'__init_subclass__',
'__iter__',
'__le__',
'__len__',
'__lt__',
'__mod__',
'__mul__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rmod__',
'__rmul__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'capitalize',
'casefold',
'center',
'count',
'encode',
'endswith',
'expandtabs',
'find',
'format',
'format_map',
'index',
'isalnum',
'isalpha',
'isascii',
'isdecimal',
'isdigit',
'isidentifier',
'islower',
'isnumeric',
'isprintable',
'isspace',
'istitle',
'isupper',
'join',
'ljust',
```

```
 'lower',
 'lstrip',
 'maketrans',
 'partition',
 'removeprefix',
 'removesuffix',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

Scroll down past all the stuff with double underscores, like `__add__`. The first thing after these should be `capitalize`. Try using this verb using the the dot notation (`object.verb()`).

[100]:
```
myName.capitalize()
```

[100]:
```
'Marifer'
```

1.4.2 %who

Type `who` in the code cell below and run it. (if it doesn't work, try `%who`)

[101]:
```
who
```

```
a        aBool   aList   aStr    anInt   anObject        anotherBool
anotherInt      anotherVar
b        c       d       d_num   d_str   eleven  expDict         immStr
immStrCopy
just_a_list      mList   mListCopy       myDict  myList  myName  myTup
notAnotherObject
```

Use this cell to say what `who` does

It lists all of the variable names created in this python session.

1.4.3 %whos

Now try `whos`.

[102]:
```
whos
```

```
Variable           Type     Data/Info
------------------------------------
a                  bool     True
aBool              bool     True
aList              list     n=3
aStr               str      Hello!
anInt              int      8
anObject           int      11
anotherBool        bool     False
anotherInt         float    8.0
anotherVar         float    8.0
b                  bool     False
c                  bool     False
d                  bool     True
d_num              float    3.8282
d_str              str      3.8282
eleven             int      11
expDict            dict     n=4
immStr             str      You cannot change me either!
immStrCopy         str      You cannot change me!
just_a_list        list     n=5
mList              list     n=4
mListCopy          list     n=4
myDict             dict     n=4
myList             list     n=4
myName             str      marifer
myTup              tuple    n=4
notAnotherObject   int      42
```

What does `whos` do?

Lists the variable name, the type, and information about the objects.

Note: who and whos are "magic commands" commands available in iPython (and therefore in Jupyter Notebooks) but are not part of Python *per se*.

[ ]:
```

```