logo_CDM

CSC 241 Notes  
Yosef Mendelsohn

With many thanks to Dr. Amber Settle for the original version of these notes.

Prof Settle's notes in turn are based on the Perkovic text.

# Character encoding and strings

For many years the standard encoding for characters in the English language was the ASCII encoding. ASCII defines a numeric code for 128 characters, punctuation, and a few other symbols common in the American English language. For example:

* 'A' is represented by 65
* 'B' by 66
* 'a' by 97
* 'b' by 98
* a dollar sign by 36
* etc

Here is a link to an ASCII table: <http://www.ascii-code.com/>

Python includes two built-in functions to manipulate ASCII encoding that you should know about:

The function **ord**() takes as a parameter a character and returns the ASCII code (i.e. number) of that character:

>>> ord('a')

97

>>> ord('$')

36

The function **chr**() is the inverse function of ord(). It takes as its parameter a numeric code and returns the character corresponding to it:

>>> chr(97)

'a'

>>> chr(45)

'-'

**Problem 1**: Write a function **encoding**() that takes a string as a parameter and prints the ASCII code of every character in it.

It would be used as follows:

>>> encoding('dad')

100 97 100

>>> encoding('mom')

109 111 109

**Problem 2**: Write a function **charCodes**(low, high) that prints the characters corresponding to ASCII decimal code i for all values of i from low up to and including high.

Try it as follows:

>>> charCodes(62,67)

>

?

@

A

B

C

See the solution in this week's solutions file.

# Objects

In Python, every value, whether a simple integer value (like 3) or a more complex value (such as the list [‘hello’, 4, 5]) is stored in memory as something called an **object**.

Objects are extremely important in nearly every programming language in common use. We will introduce a few key concepts today and will build on them in subsequent lectures.

Every Python has:

1. A **type** that indicates what kind of values the object can hold – this is important because the type determines what kind of operations are valid for the object.
2. A **value**, This refers to the contents or the data stored inside the object.

To illustrate the difference, consider the following examples:

>>> type(3)

<class 'int'>

>>> type(3.0)

<class 'float'>

>>> type('hello')

<class 'str'>

>>> type([])

<class 'list'>

>>> 3

3

>>> 3.0

3.0

>>> 'hello'

'hello'

>>> []

[]

There are two types of comparisons you can do between objects:

1. **Object value comparisons**: Compare the values stored inside two objects. Note: To do this, the two objects must be of the same data type.
2. **Object identity comparisons**: Compare the objects themselves. Note: This is a very important point and will require further discussion.

To see how this works, consider some examples.

**NOTE**: Understanding the discussion of objects below will be made much easier – and you will understand the concepts far better -- if you get in the habit of drawing out diagrams of the objects. I will demonstrate in class.

## Object value comparisons

Comparison operators are used to determine the equality between two objects . (NOTE: The two objects must be of the same type) :

>>> 3 == 3

True

>>> 3 == 4

False

>>> [2, 3, 4] == [2, 3, 4]

True

>>> [2, 3, 4] == [3]

False

>>> 3 <= 4 == 4 < 8 != 9

True

## Object identity comparisons

Suppose we have the following two objects:

lst1 = [2,3,4]

lst2 = [2,3,4]

As you can see, these are two entirely separate variables (objects). However, they do happen to be holding the same values. Therefore, typing:

**lst1 == lst2**

will return True

However, suppose we now change one of them slightly:

lst2 = [2,3,**5**]

I hope you recognize that at this time, lst1 == lst2

would return False.

So the comparison operator **==** compares the "insides" (ie the contents) of two objects and returns True if both objects hold the exact same data.

**The is operator:**

Now let's talk about another important operator called **is**

The **is** operator compares two references (e.g. variables) to see if they are pointing to the same object in memory.

For example, suppose we create the two lists as we did earlier:

lst1 = [2,3,4]

lst2 = [2,3,4]

lst1 **==** lst2 #will return True

lst1 **is** lst2 **#will return False!!**

This is important!! Be sure you understand why this happened.

Now let's re-create 'lst2' but instead of assigning it [2,3,4] explicitly, we will assign it 'lst1':

lst2 = lst1

In other words, both objects (lst1 and lst2) are pointing to the exact same object.

NOW if we say:

lst1 **is** lst2

it will return True

**ALSO:** This means that if we were to make a change to lst1, then lst2 would also be changed and vice-versa:

lst1 = [2,3,4]

lst2 = lst1

lst2[0] = 999 #change a value in lst2

print(**lst1**)

#would output: [999, 3, 4]

We have already seen that we can compare the contents of two objects using ==.

lst1 == lst2

However, what if we want to see if two identifiers are referring (pointing) to the same object as is the case with lst1 and lst2 just above?

As we have stated, we do this using the keyword '**is**'.

lstA = [2,3,4]

lstB = [2,3,4]

lstC = lstA

* lstA **==** lstB #returns True
* lstA **is** lstB #**returns False**

Now let's try using the 'is' operator on two variables that point to the same object in memory:

* lstA is lstC #returns True
* lstC is lstA #also returns True

**Diagram this out!**

Another example – again, draw these out:

>>> a = [3, 4]

>>> b = a

>>> a == b

True

>>> a is b

True

>>> b is a

True

>>> c = [3, 4]

>>> a == c

True

>>> a is c

**False**

**Potential Source of Confusion:**

The previous discussion of variables and the memory locations they reference is very important.

However, there is one particular situation where the Python interpreter does something unexpected. When two objects are holding the same integer, Python will have those objects point to the same location in memory. Note: This only applies to integers.

>>> a = 2.0

>>> b = 2.0

>>> a is b

False

>>> a = 2

>>> b = 2

>>> a is b

True

This last one should bother you. 'a' and 'b' are entirely different objects. They both just so happen to be storing the same data.

So why did we get 'True' for the last example? The Python virtual machine decided, on its own, to save some memory and re-use integer objects with value 2. That is, even though a and b are entirely different objects, the interpreter saves a little memory by having both objects (a and b) point to the same location in memory. This is a quirk of Python. It's not a big deal for our purposes, but it can sometimes be a source of confusion. I hope to discuss in a little more detail in lecture.

From a coding perspective, this business of having objects that hold integers point to the same location in memory turns out to be a non-issue since numbers are immutable. We’ll talk later about what this means precisely.