mfre_summer_python_prep2

May 22, 2024

Part B Now, let's move forward into more of the basics. We'll look at the building blocks of how Python stores data in memory and implements fundamental concepts such as program logic.

0.0.1 Lists

The basic data container in Python is a *list*. Lists act as a general purpose container for Python objects - they can hold any combination of data of different types.

A *list* is created as an ordered sequence of data enclosed in square brackets, with each element separated by a comma. For example:

[ ]:
```
l = [2, 3, 5, 7]
```

Lists can contain multiple data types as entries:

[ ]:
```
m = [10, 'ES', -1.234, True]
```

and can even have other lists as entries:

[ ]:
```
J = [21, 5, ['apples', 12, False]]
```

[ ]:
```
You can create an empty list by defining a variable name to brackets ```[]```:
```

[ ]:
```
empty_list = []
# Calling it will produce the empty list
empty_list
```

Or also using the `list()` function:

[ ]:
```
print(J[0])
print(J[2][0])
```

As mentioned, we can nest lists inside of other lists. This allows us to effectively create *matrices* with lists (we'll be speaking more about matrices when we get to the lesson on statistics and linear algebra). In our list of lists, the internal lists represent the *rows* of the matrix. For example, we can represent the matrix:

$$B = \begin{bmatrix} 1 & 2 \\ 0 & 4 \end{bmatrix}$$

as:

[ ]:
```
B = [[1, 2], [0, 4]]
```

The element 2 in the matrix B is in the first row, second column. How would we retrieve that from the list object `B` using indices?

[ ]:
```
B
```

0.0.2 0 vs 1 indexing

Python is a 0 indexed language. This means if I want the first object in a list, I need to tell Python that I want the 0th object. R, Matlab, Mathematica, and some other programming languages are 1 indexed, where the first item in an array is array[1], as that is how humans have counted for thousands of years. 0 indexed languages are more common, and have an obscure history - in fact, there are even claims that it is due to yacht racing in the 1960s.

Off-by-one errors are one of the most common bugs in computing - and a little bit of UX/UI considerations might have significantly reduced them. Some of your instructors switch between two languages: Python and R. Python is 0 indexed and R is 1 indexed - keep and eye out for these errors.

As well as indexing individual elements, we can take slices using a `:`, to indicate a range of elements to return.

0.0.3 Slicing list elements

If we wanted to access a slice of list elements, we would use the same syntax for slicing string elements: for a given list `l`, to slice out the elements from index `n` up to (but not including) index `m`, we would write: `l[n:m]` As an example, if we consider the following list:

[ ]:
```
sl = ['apples', 'pears', 'oranges', 'cherries', 'grapefruits', 'pomegranates']
```

and wanted elements 1 through 3, we would slice:

[ ]:
```
sl[1:4]
```

Leaving the first and/or last index call blank will make Python assume you mean "from the beginning" or "to the end".

[ ]:
```
sl[2:]
```

[ ]:
```
This slices from element 2 until the end.<br>

We can use the step feature as well, with a second colon:
```

[ ]:
```
sl[1::2]
```

This slice starts from element 1 goes until the end, only returning every second item.

So far we've been accessing list elements by index, but what if we want to find out what the index is of a particular list element? (Particularly handy if we have very long lists.) We accomplish this using the `index()` command, with the syntax `listname.index(element_name)`. If we wanted to know what the index of `'cherries'` from the list `sl` is, we would write:

[ ]:
```
sl.index('cherries')
```

One issue with this is that it will only return the index of the *first* occurrence of the item you ask for.

[ ]:
```
repeat = [14, 15, 16, 16, 17]
repeat.index(16)
```

How would we write code that would give us a list of all of the indices of the occurrences of 16 in the list called `repeat`?

0.0.4 Replacing and adding list elements

We access a list element by calling `listname[index_number]`, but we can also use this syntax to redefine list elements. If we use the list `l` above, and wanted to replace the second element with the string `'lantern'`, we would write:

[ ]:
```
l[1] = 'lantern'
l
```

This is the same syntax we use for defining a variable. But what if we wanted to add something new to the list? There are a few options here: 1. We can use Python's `append` command to tack on whatever we want to the end of the list. This operation is performed "in place", meaning our list is redefined to be a list with all of our old elements, and then with the new element at the end. 2. We can concatenate two lists together using `+`, i.e.: we can define a new list which is the old list + another list containing the new element(s) we want added.

[ ]:
```
l = [1, 2, 3, 4]
l.append(5)
l
```

[ ]:
```
m = [1, 2, 3, 4]
newlist = m + [5]
newlist
```

It's important to remember that the element `5` we're adding to the list must itself be in a list. If we tried to just add the number 5, we get the error message:

[ ]:
```
m+5
```

We can also concatenate copies of a list onto itself a set number of times by using *:

[ ]:
```
m * 3
```

This is especially handy when you want a list of size `n` that all contain the same element. For example, if we wanted a list of all 1s, whose length was 50, we could easily write:

[ ]:
```
ones = [1]*50
print(ones)
```

Let's do an exercise where we expand our list `sl` to include kiwis, pineapples, grapes, and tomatoes.

[ ]:
```
sl = ['apples', 'pears', 'oranges', 'cherries', 'grapefruits', 'pomegranates']
```

0.0.5 List sorting

The order of elements in a list matters, but what if it's not in the order we want it? Python comes with a few list sorting options, that usually follows alphabetical or numerical rules. For Python 3, the common tool is the `sorted()` function. If we pass a list of comparable elements (can we compare strings and numbers?), it will produce a list in numerical or alphabetical order.

[ ]:
```
ln = [2, -5, 10, 7, 3, 5]
sorted(ln)
```

[ ]:
```
lm = ['apples', 'pears', 'oranges', 'cherries', 'grapefruits', 'pomegranates']
sorted(lm)
```

[ ]:
```
lm
```

We can also call `.sort()` on a list directly, which operates in place:

[ ]:
```
lm.sort()
lm
```

What would happen if we tried to sort the following two lists?

[ ]:
```
test1 = [1, 3, 0, True, 10, 3.5, False]
```

[ ]:
```
test2 = [1, '2', 3, '4', 5, 'A']
```

For test2, what could we do to create a new list which first has the sorted integers, and then has the sorted strings?

[ ]:
```
test2_sorted =
```

0.0.6 Mutability

Lists are mutable, meaning that we can change objects. We need to be careful when assigning lists to variables. By default, Python does not copy the list, but instead points to the same list in memory:

[ ]:
```
my_list = [1, 2, 3]
a = my_list
```

```
my_list[1] = 5
print(my_list)
print(a)
```

Some objects are immutable, meaning they cannot be modified. Strings have this property:

[ ]:
```
my_str = 'abc'
my_str[1] = 'd'
```

As we saw, we can add to lists, using the append method. Be aware, methods on mutable objects will often operate in place. There are many more list methods we will cover as we continue - you can take a look using tab completion in notebooks.

This means we don't have to reassign the object, but also that we have side effects, as we saw above:

[ ]:
```
mylist = [1, 2, 3]
mylist2 = mylist.append(4)
print(mylist2)  # huh, None
print(mylist)
```

0.1 Dictionaries

Dictionaries are similar to lists in that they're indexed collections of data. The main difference, however, is that order does not matter in a dictionary. So if order doesn't matter, how do we keep track of the data? The answer is with *keys*. Dictionaries are enclosed in braces `{}`, and follow the syntax: `dictname = {key1:value1, key2:,value2, ...}`

Here, the `keys` are the index we're choosing, which can be integers, floats, or strings. The `values` can be any other Python object we want, like numbers, lists, strings, other dictionaries, dataframes, etc. One restriction is that keys must be unique; values can be repeated as many times as you like. Here is an example dictionary, where the keys are the names of students, and the values are their midterm marks.

[ ]:
```
midterm_marks = {
    'Olivier': 86,
    'Juan': 95,
    'Yasi': 92,
    'Xiao': 97,
    'Janelle': 89
}
```

If we wanted to access the value of a particular key, the syntax is the same as calling the index of a list item: `dictname[key]`. So to access Ivan's mark, we write:

[ ]:
```
midterm_marks['Yasi']
```

If we had a dictionary with many keys and wanted to access them, we can use the `dictname.keys()` command. Similarly, we can call the values with `dictname.values()`:

[ ]:
```
midterm_marks.keys()
```

[ ]:
```
midterm_marks.values()
```

If we wanted both the keys and values, we can use the `.items()` command:

[ ]:
```
midterm_marks.items()
```

This returns a special `dict_items` object, which is basically a list of tuples, where each tuple is a key-value pair. *(Note: A tuple is another Python container type, like a list, but its values cannot be modified after creation. It is denoted with round brackets instead of square ones.)*

[ ]:
```
list(midterm_marks.items())
```

0.1.1 Updating key values, adding new key values

Adding new entries, and updating previous entries are handled via the same syntax as with lists. If we wanted to add Tyler's mark of 90 to the midterm marks, we'd write:

[ ]:
```
midterm_marks['Tyler'] = 90
midterm_marks
```

If it turns out that Patrick's mark was actually 89, how would we change it?

We can use the `sorted()` function with dictionaries too. Calling it on the keys or the values of the dictionary will produce a list that's sorted, assuming you can compare the entries:

[ ]:
```
sorted(midterm_marks)
```

[ ]:
```
sorted(midterm_marks.values())
```

Say we wanted to find the name of the person with the highest mark. How would we do this? (There are lots of right answers.)

To show issues with trying to define non-unique keys, consider the following example:

[ ]:
```
bad = {'A': 1, 'B': 2, 'A': 3}
```

[ ]:
```
bad['A']
```

Why does it show the value is 3? Because the second time we defined `A`, Python overwrote the original definition. This can be seen by looking at the actual dictionary:

[ ]:
```
bad
```

0.1.2 Looping

Sometimes we want to repeat an action multiple times, or for multiple objects. One way of doing this is simply to copy and paste elements of our code:

[ ]:
```
my_str = 'apple'

print(my_str[0])
print(my_str[1])
print(my_str[2])
print(my_str[3])
print(my_str[4])
```

We can be much more concise and avoid repetitive code by using a `for loop`, (something so common in programming that same exists in nearly every coding language):

[ ]:
```
for i in my_str:  # we can vary the name of i here
    print(i)  # technically it is the `loop variable'

# print(i)
```

The added benefit of this is that we can change the thing we are looping over without having to modify our code.

Python can iterate over anything that is an 'iterable'. We will talk more about different types of iterables later on.

Another common pattern is to make a numeric 'index' and loop over that:

[ ]:
```
my_str2 = 'banana'
myval = 0

for i in range(len(my_str)):
    print(my_str[i])
    print(my_str2[i])
    myval += 1

print(myval)
```

0.1.3 Exercise

1. What does the function `len` do?
2. What does the function `range` do?
3. Take a look at the following code:

```
a, b = 5, 10
b, a = a, b
```

What are a and b equal to?

4. What does enumerate do here?

```
for i,j in enumerate(my_str):
    print(j)
    print(my_str[i])
```

5. Create a loop, which loops over a list to calculate a polynomial:

```
x = 5
factors = [1,2,3]
```

This should return us the output of the following computation: `1*x**0 + 2*x**1 + 3*x**2`

Make it work with `x = 8` and `[8,7,6,5,4,4,2,1]`

0.1.4 Conditionals

We sometimes want to carry out different operations depending on certain conditions.

In this case, we can use `if`, `else` and `while` clauses.

[ ]:
```
for i in [1, 2, 3, 4, 5]:
    if i % 2 == 0:
        print('even')
    else:
        print('odd')
```

In this loop, we set `i` to be equal to each element in turn, find the remainder when divided by 2, and see if the remainder is equal to 1.

We test equality using `==` in python, because a single `=` denotes assignment.

Notice the indentation, this is the same as when we carried out `for loops`. While you can indent indefinitely, it's generally bad practice to end up more than 4 loops in, as you probably can't follow it in your internal python repo anymore.

If we have more conditions, we can continue on using `elif`:

[ ]:
```
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

for i in data:
    if i % 3 == 0 and i % 5 == 0:
        print('fizzbuzz')
    elif i % 3 == 0:
        print('fizz')
    elif i % 5 == 0:
        print('buzz')
    else:
        print(i)
```

We could also have written this as:

[ ]:
```
for i in data:
    if i % 3 == 0:
        if i % 5 == 0:
            print('fizzbuzz')
        else:
            print('fizz')
```

```
    elif i % 5 == 0:
        print('buzz')
    else:
        print(i)
```

This is an answer to the infamous fizzbuzz test, which varying percentages of programmers (depending on who you listen to) cannot answer in an interview.

We are evaluating the clause after `if` or `elif`, to see its boolean representation:

[ ]:
```
print(1 % 2 == 0)
print(1 % 2 == 0 or 2 % 2 == 0)
print(1 % 2 == 0 and 2 % 2 == 0)
print(not 1 % 2 == 0)
```

0.1.5 Functions

So far, we have simply used loops to carry out our analysis. What if we want to make our own functions?

We can define our own functions!

Here is the basic syntax for writing a function in Python:

[ ]:
```
def my_function(my_argument_a, my_argument_b):
    '''
    docstring
    '''
    output = my_argument_a + my_argument_b

    return output
```

Let's write some functions for performing temperature conversions:

[ ]:
```
def convert_f_to_c(degf):
    '''
    converts fahrenheit to celsius
    '''
    degc = (degf - 32) * 5/9
    return degc


def convert_c_to_k(degc):
    '''
    converts celsius to kelvin
    '''
    degk = degc - 273
    return degk
```

[ ]:
```
def convert_f_to_k(degf):
    degc = convert_f_to_c(degf)
    degk = convert_c_to_k(degc)
    return degk
```

We can now access these functions the same as any others in Python. We can also use tab completion, even Python's `help` command will work on our function if we filled out its docstring:

[ ]:
```
help(convert_f_to_c)
```

Let's test our functions to ensure we get sensible values:

[ ]:
```
print(convert_f_to_c(100))
print(convert_f_to_c(0))
print(convert_f_to_c(-40))
print(convert_f_to_k(-40))
```

Functions are 'scoped' locally. This means they won't modify anything outside them unless we try:

[ ]:
```
x = 55


def my_func(z):
    # global x
    # probably don't do this.....
    x = 12
    return z + x


print(my_func(33))
print(x)
```

However, watch out for mutable objects:

[ ]:
```
def my_func(mylist):
    mylist.append(5)
    return 12


x = [1, 2, 3, 4]

my_func(x)
print(x)
```