Chapter 3

Python values are objects

In Python each type of value (like integers, floating-points and strings) is not only holds data but is also packaged with a lot of meta-information and functionality. The extra functionality comes in the form of methods. You can think of methods as functions that are packaged together with the value. You already know one method quite well already. In chapter 1 we worked with this string:

"I am {} percent taller than Dan", and to substitute {} with the percentage you used the `format` method.

```
"I am {} percent taller than Dan".format(height_difference_in_percent)
```

To emphasize that the method operates on the string value, we call the method by connecting the value and the method call with a dot. So to call a method on a value you do the following:

1. Write the value.
2. Then you write a `.`.
3. Then you write the name of the method.
4. Then you write two parentheses. If the method takes any arguments like `format` does, then you write those values between the parentheses with commas in between.

You can see that the method call itself looks just like a function call and in many ways calling a method works much like calling a function. The difference is that when we call a function we say “Hey function, format that string!” When we call a method we say “Hey string, format yourself!”

So why do we need methods? Can’t we just do with functions? Yes we can, but it turns out that it is very handy that the functionality specific to some type of data is packaged with the data it works on. You will start to appreciate that sooner than you think.

Now write and run these examples:
```python
statement = "Methods Are Cool"
print(statement)

shout = statement.upper()
print(shout)

whisper = statement.lower()
print(whisper)

new_statement = statement.replace("Cool", "Fantastic")
print(new_statement)
```

You can see what these methods do. For example, `upper` returns an uppercased copy of the string. So in the example above `statement.upper()` is substituted by the value `'METHODS ARE COOL'`.

Maybe you think: “Didn’t he just say that you call a method by writing it after a `value`?” In the example above he writes it after a `variable`. And then I say: remember oath two! When Python evaluates an expression like `statement.upper()` it first substitutes `statement` with its `value` ("Methods Are Cool") and then it calls the method `upper` for that value. If this is not completely clear, go through chapter 1 again.

**Exercises**

**Exercise 3.1:** Write and run the following code. What do you think it does?

```python
line = '\n	Some text\n'
print("\n{}\n".format(line))

line = line.strip()
print("\n{}\n".format(line))
```

Make sure you do the substitution and reduction steps in your head. Be especially careful about the third line of code. Also, what do you think the special `\t` character is?

**Exercise 3.2:** The string methods you have tried so far have all returned a string. Try this example:

```python
'Donald Trump'.startswith('Don')
```

Using the Python documentation

Now that you are well underway to becoming a programmer, you should know your way around the Python documentation. Especially the part called the Python standard library. There is a lot of things in there that
we do not cover in this course. These are mainly are tools and techniques that for writing more efficient, extensible, robust and flexible code. The parts we cover in this course are is the minimal set that will allow you to write a program that can do anything.

Exercises

**Exercise 3.3:** There is also a string method that returns a secret agent:

```python
print('7'.zfill(3))
```

If you do not believe me you can look it up in the Python documentation.

**Exercise 3.4:** Try to find the documentation for the append method for lists.

**Exercise 3.5:** Try to find the documentation for how slicing of lists and strings work.

Lists

Often when you have some data, the order of things is important. The order of characters is important for the meaning of text in a string. Sometimes we want to specify the order of other things than characters - like numbers or strings. A list is useful when the relative order of items in the list has some meaning. It could be a grocery list where you have listed the things to buy in the order you get to them in the supermarket. When you print a list it nicely prints all the values it contains.

```python
grocery_list = ["salad", "canned beans", "milk", 'beer', 'candy']
print(grocery_list)
```

Unlike strings that are sequences of characters, lists can have any kind of values in them, and you can mix different types of values any way you like. Here is a list that contains an integer, a boolean, a string and a list:

```python
mixed_list = [42, True, 'programming', [1, 2, 3]]
```

By now you have probably guessed that you make a list with two square brackets. A list can have values with commas in between like the ones above or it can be empty like this:

```python
my_list = []
```

You can add single values to the end of a list using the `append` method of lists:
desserts = []
print(desserts)
desserts.append('Crepe suzette')
print(desserts)
desserts.append('Tiramisu')
print(desserts)
desserts.append('Creme brulee')
print(desserts)

If you you have a list you want to add to the end of another list you use the `extend` method:

cheeses = ['Gorgonzola', 'Emmentaler', 'Camembert']
desserts.extend(cheeses)
print(desserts)

Notice how appending, extending modifies the list instead of producing a new list.

**Exercises**

**Exercise 3.6:** If you want to test if a value is in a list you use the `in` operator. Try this:

```python
print('Tiramisu' in desserts)
print('Meatloaf' in desserts)
```

The `in` operator works on more than lists. What do you think is printed here? Try.

```python
print('sip' in 'mississippi')
```

**Exercise 3.7:** You can concatenate two lists. Make sure you figure out how this works before you try it out. Then experiment with changing the lists. Can you concatenate two empty lists?

```python
some_list = [1, 2, 3]
another_list = [7, 8, 9]
merged_list = some_list + another_list
print(merged_list)
```

**Exercise 3.8:**

What do you think is printed here? Make sure you figure out how this works before you try it out.
Indexing and slicing

Now you know how to make lists, but to work with the values in lists you must also know how to access the individual values that a list contains. Each value in a list is identified by its index in the list. To access a value in a list you write brackets after the list and between those brackets you specify the index of the value you want. The first value has index 0, the second has index 1 and so on.

```python
my_list = []
x = my_list.append(7)
print(x)
```

numbers = [7, 4, 6, 2, 8, 1]
```python
print("first element is", numbers[0])
print("second element is", numbers[1])
print("third element is", numbers[2])
print("fourth element is", numbers[3])
```

You may wonder the index of the first value is zero and not one but that is simply the convention in programming. Over time you will start to find this useful rather than annoying. You should think of the index as the offset from the start of the list.

If you want a sub-sequence of values from a list (we call that a slice) you specify a start index and an end index separated by a colon:

```python
print(numbers[1:4])
```

When you run that you can see that numbers[1:4] is substituted for [4, 6, 2], so the slicing operation produces a list of the specified values.

You may wonder why the value at index 4 is not in the resulting list. That is another programming convention: intervals are "ends exclusive". So when you specify an interval with a start index of 1 and an end index of 4 it represents all the numbers starting from 1 and up to, but not including, 4. So the slice 1:4 corresponds to the values at indexes 1, 2 and 3. The reason programmers handle intervals in this way is that it makes it easier to write clear and simple code as you will see in the exercises.

**Exercises**

**Heads up:** All the indexing and slicing operations also work on strings. I have thrown in some string exercises below to help you figure this out by yourself.

**Exercise 3.9:** What do these two expressions reduce to?
and

"Donald Trump"[7]

Exercise 3.10: What is printed here? Do all the substitution and reduction steps and compare to the exercise above.

```python
l = [11, 12, 13, 14, 15, 16, 17]
print(l[2])
```

and

```python
s = "Donald Trump"
print(s[7])
```

Exercise 3.11: What is printed here? Do all the substitution and reduction steps - and do it twice. Next week you will be happy you did.

```python
numbers = [1, 2, 3]
i = 0
print(numbers[i])
i = 1
print(numbers[i])
i = 2
print(numbers[i])
```

Exercise 3.12: What do you think happens here? Make up your mind and try out the code below:

```python
l = [11, 12, 13, 14, 15, 16, 17]
l[4] = "Donald"
print(l)
```

Exercise 3.13: What do you think happens here? Make up your mind and try out the code below:

```python
s = "Donald Trump"
s[7] = 'D'
```
Did you see that comming? Strings are *immutable*, which means that you cannot change them once you have made them. If you want "Donald Drump" you need to produce a new string with that value. Try to figure out how to do that with the `replace` method of strings.

**Exercise 3.14:** When you do not specify the start and/or the end of a slice Python will assume sensible defaults for the start and end indexes. What do you think they are? Make up your mind and try out the code below:

```python
l = [11, 12, 13, 14, 15, 16, 17]
print(l[::3])
print(l[3:])
print(l[:])

s = 'abcdefghijlmnopqrstuvwxyz'
print(s[:11])
print(s[11:])
print(s[:])
```

**Exercise 3.15:** What do you think happens when you specify an index that does not correspond to a value in the list:

```python
l = [11, 12, 13, 14, 15, 16, 17]
print(l[7])

s = 'abcdefghijlmnopqrstuvwxyz'
print(s[99])
```

Read and understand the error message.

**Exercise 3.16:** Do you think you also get an error when you specify a slice where the end is too high? Try it out:

```python
l = [11, 12, 13, 14, 15, 16, 17]
print(l[3:99])

s = 'abcdefghijlmnopqrstuvwxyz'
print(s[13:99])
```

I guess that is worth remembering, right?

**Exercise 3.17:** What do you think the builtin `len` function does?
Exercise 3.18: Which value in a list does this expression reduce to?

```
l[len(l)-1]
```

Exercise 3.19: If you do not like Emmentaler you can just delete it. What do you think the `del` keyword does?

```
cheeses = ['Gorgonzola', 'Emmentaler', 'Camembert']
print(cheeses)
del cheeses[1]
print(cheeses)
```

Exercise 3.20: Because intervals are “ends exclusive” we can compute the length of a slice as

```
end - start:
```

```
l = [7, 4, 6, 2, 8, 1]
start = 1
end = 4
print("{} has length {}".format(l[start:end], end-start))
```

Think about what this code would look like if ends were included in intervals.

Exercise 3.21: Another advantage of “ends exclusive” intervals is that you only need one index to split a list in two:

```
l[:i] + l[i:] == l
```

If ends were included in intervals this would not be as simple.

Exercise 3.22: Do all the substitution and reduction steps in your head (or on paper) before you write any of the following code. Think carefully and make up your mind what you think will be printed below. Remember that the value of a list is a container that holds other values in it. Then write the code and see if you were right. If you were not, make sure you figure out what led you to the wrong conclusion.
Exercise 3.23: Did you look up the details of how slicing works? If so you should be able to explain what happens here:

```
Exercise 3.23: Did you look up the details of how slicing works? If so you should be able to explain what happens here:

l = [1, 2, 3, 4, 5]
s = 'abcdefghi'
print(l[:3])
print(s[:3])
```

### Dictionaries

Lists are useful for storing values when the order of the values is important but lists have one drawback: you can only access a value in a list using the index of the value.

A dictionary, called `dict` in Python, is a much more flexible data type. Like a list, a dictionary is a container for other values, but dictionaries do not store values in sequence. They work more like a database that lets you store individual values. When you store a value you assign it to a key that you can then use to access the stored value. Now create your first dictionary:

```
person = {'name': 'Donald Trump', 'age': 70, 'job': 'President'}
```

This dictionary has three values ("President", "Donald Trump" and 70) and each values is associated with a key. Here "age" is the key for the value 70. So when defining a dictionary you should note the following:

1. You make a dictionary using braces.
2. Between you braces you put key-value pairs separated by a colon.
3. The key-value pairs are separated by commas.
4. To make an empty dictionary you just write the braces with nothing between them: `{}`.

To access a value in the dictionary you
"{} is a {} year old {}".format(person['name'], person['age'], person['job'])

Here we used strings as keys, but you can also use many types of values as keys (Python will give you an error if you try to use a type that is not allowed):

```python
misc_dict = {42: "Meaning of life", "pi": 3.14159, True: 7}
```

A dictionary stores key-value pairs but do not keep track of their order. So when you print a dictionary the order of the key-values pairs is arbitrary.

If you have a dictionary you can add key-value pairs in this way:

```python
person['age'] = 71
person['personality'] = ['narcissist', 'sexist', 'megalomaniac']
print(person)
```

Notice that if you assign a value (71) to a key that is already in the dictionary ('age'), then the old value (70) is replaced.

**Exercises**

**Exercise 3.24:** What does this expression evaluate to?

```python
{'name': 'Donald Trump', 'age': 70, 'job': 'President'}['name']
```

**Exercise 3.25:** Assuming the definition of the person dictionary above, what does this expression evaluate to? Compare to the expression in the previous exercise.

```python
person['name']
```

**Exercise 3.26:** Write and run this code
Exercise 3.27: The *in* operator also works with dictionaries. Look at what these expressions reduce to and then try to figure out what *in* does when applied to a dictionary:

```python
'name' in person
'age' in person
'job' in person
70 in person
'President' in person
'Donald Trump' in person
```

Exercise 3.28: Write and run this code with different values of `key` and read any error messages.

```python
key = 3
# key = 'banana'
# key = 3.14159
# key = True
# key = {}
# key = []
d = {}
d[key] = 7
```

Exercise 3.29: Do you think this will work?

```python
person = {'name': 'Donald Trump',
          'age': 70,
          'job': 'President'}
print(person)
```

Python keywords

By now you know that some of the words in your code have specific purposes. *def* defines functions,
return returns value from a function, and is a logical operator etc. Here is a list of the ones you will see in this course: and, assert, break, continue, def, del, elif, else, False, for, from, if, import, in, is, not, or, pass, return, while, True, None (you can see a full list here)

These words are reserved for their special purposes in Python and you will not be allowed to assign values to them. Try this to see for yourself:

```
None = 4
```
or this:

```
and = 2
```

**Chapter exercises**

Start by making dictionaries for (some of) the Trump family:

```python
donald = {'name': 'Donald Trump', 'age': 70, 'job': 'President'}
melania = {'name': 'Melania Trump', 'age': 70, 'job': 'First lady'}
tiffany = {'name': 'Tiffany Trump', 'age': 23, 'job': 'Internet personality'}
ivanka = {'name': 'Ivanka Trump', 'age': 35, 'job': 'Top aide'}
```

**Exercise 3.30:** What do you think the following code produces? Do all of the substitution and reduction steps in your head, and only then try out the code.

```python
donald['child'] = tiffany
melania['husband'] = donald

print(melania)
print(melania['husband']['child'])
```

**Exercise 3.31:** A dictionary can contain any kind of Python values, even lists or dictionaries. Consider the code below where we add a list of ex-wives to the Trump persona. Can you see why we need to check if the 'ex-wives' key before we add to the list of ex-wives?
```python
donald = {'name': 'Donald Trump', 'age': 70, 'job': 'President'}

if 'ex-wives' not in donald:
    donald['ex-wives'] = []
donald['ex-wives'].append('Marla Maples')
donald['ex-wives'].append('Ivana Trump')

print(donald)

Exercise 3.32: Lists can also contain any kind of value. Consider this example. What do you think the following code produces? Do all the substitution and reduction steps in your head, and only then try out the code.

```