# Python Recap

15 November 2024

**Python has three principal layers of functionality:**

1. Built-ins, which are always available, eg `print`.
2. Standard library modules, which are installed with Python but need to be imported before they can be used, eg `re`.
3. Third-party modules, which need to be installed and then imported, eg `requests`. Note that the Anaconda Python distribution includes more than the standard installation.

When you create a *variable* Python infers its type, which then affects what you can do with it. You can use string methods on strings, as we did in week 1, but you have to use integer methods on integers.

The main types are: *string*, *integer*, *float*, *Boolean*.

[ ]:
```
name = "Taylor Swift"
age = 34
shoe_size = 9.0
is_famous = True
```

[ ]:
```
print(type(name))
print(type(age))
print(type(shoe_size))
print(type(is_famous))
```

Remember that you can get help with the available methods in Jupyter Notebook.

[ ]:
```
name.
```

Python also has container datatypes known as *Collections*. The ones to be comfortable with are *list*, *tuple*, *set* and *dictionary*.

**Lists** Lists go in square brackets. They're a good data type to learn first because they match our natural intuition for lists. If you have a to-do list you can delete completed items and add new ones. In Python's terminology, this means they are *mutable*.

[ ]:
```
beatles = ["John", "Paul", "George", "Ringo"]
```

You can get parts of a list, which Python calls *slices* with more square brackets. But remember that Python starts counting from zero.

What will `beatles[3]` return?

[ ]:
```
print(beatles[3])
```

[ ]:
```
print(beatles[1:2])
print(beatles[2:])
print(beatles[-1])
```

Note in the first example, that Python uses the second value of a range as a stopping value, so it's not included in the result.

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

Lists are particularly suitable for a *comprehension*, where you loop over all the members of a collection. This is the simplest form:

[ ]:
```
for beatle in beatles:
    print(beatle)
```

But you can use fstrings to produce something a bit more complicated. Put `f` before the first set of quotation marks and then within the quotation marks you can refer to the value held by a variable by putting the variable in curly brackets:

[ ]:
```
for beatle in beatles:
    print(f"{beatle} was a member of The Beatles.")
```

**Logic** You can use `if`, `else` and `elif` to do different things depending on the value of something:

[ ]:
```
if len(beatles) > 4:
    print("Some say George Martin was the fifth Beatle.")
elif len(beatles) < 4:
    print("Some say McCartney died in 1965 and there was a cover-up")
else:
    print("Actually there were four Beatles.")
```

We can combine a list comprehension and an if statement to filter lists. First we'll need to create new, empty lists:

[ ]:
```
long_name_beatles = []
short_name_beatles = []
```

[ ]:
```
for beatle in beatles:
    if len(beatle) > 4:
        long_name_beatles.append(beatle)
    else:
        short_name_beatles.append(beatle)
```

[ ]:
```
print(long_name_beatles)
print(short_name_beatles)
```

**Tuples** Tuples are like lists in that they are *ordered* but they are unlike lists in that they are *immutable*. Tuples go in round brackets:

```
[ ]: immutable_beatles = ("John", "Paul", "George", "Ringo")
```

As long as we're not changing the tuple, we can do exactly the same things as with lists, eg:

```
[ ]: for beatle in immutable_beatles:
         print(f"{beatle} was an immutable beatle")
```

But you can't add or delete. If you try you'll get an error:

```
[ ]: immutable_beatles.append("George Martin")
```

This makes tuples more efficient. But why?

**Sets** Sets are *mutable* (like lists) but not *ordered*. Sets cannot contain duplicates. This is useful for guaranteeing uniqueness:

```
[ ]: mylist = [23, 42, 96, 42, 47, 91, 57, 11, 98, 47, 32]
     myset = set(mylist)
     print(myset)
```

Another nice feature of sets is that you can compare two or more sets and find: - the *intersection* (which elements are common to all) `&` - the *union* (all elements in all sets) `|` - the *difference* (elements in set 1 which are not in set 2) -

```
[ ]: set1 = set([81011, 91011, 70101, 91100, 91111, 91001, 81100, 81111, 81001])
     set2 = set([70110, 91000, 90111, 91001, 81011, 81100, 71011, 81110, 81001,␣
      ↪91111])
```

```
[ ]: print(f"These numbers are in both sets {set1 & set2}.\n")
     print(f"These numbers are in at least one set {set1 | set2}.\n")
     print(f"These numbers are in set 1 only: {set1 - set2}.\n")
     print(f"These numbers are in set 2 only: {set2 - set1}.\n")
```

**Dictionaries** Dictionaries are pairs of keys and values separated by colons. The syntax is very similar (but not identical) to JSON. Keys must be unique within a dictionary. Values can be any of the data types above, or other dictionaries. So values can be lists or dictionaries can be nested inside dictionaries.

```
[ ]: dictbeatles = {"John": 1940, "Paul": 1942, "George": 1943, "Ringo": 1940}
```

```
[ ]: print(f"Paul was born in {dictbeatles['Paul']}")
```

We can iterate over a dictionary but the syntax is slightly more complicated than with a list or tuple comprehension. Conventionally the key and the value are given the names `k` and `v`.

```
[ ]: for k, v in dictbeatles.items():
         print(k, v)
```

We can write `if` statements to filter the keys or the values or both.

```
[ ]: print("The younger Beatles are:\n")
for k, v in dictbeatles.items():
    if v > 1940:
        print(k)
```

You can sort a dictionary by its key:

```
[ ]: print(sorted(dictbeatles.items()))
```

It's a bit more complicated to get things like maximums and minimums from a dictionary, but here's how you might do it.

```
[ ]: print(f"The youngest Beatle was {max(dictbeatles, key=dictbeatles.get)}.")
print(f"The oldest Beatle was {min(dictbeatles, key=dictbeatles.get)}.")
```

In the real world, John was not the oldest Beatle...

**Functions** Functions are repeatable bits of code. You don't have to use them but they make code cleaner and reduce errors and inconsistencies. Once you have a function that works you can use it in different scripts too.

When you read other people's code you will almost certainly see some functions. Functions work a bit like a formula in Excel: `=sum(A1:A2)`

Here `sum` is a function and it has two *arguments*: `A1` and `A2`. In Excel there have to be at least two arguments to sum because it doesn't make sense to sum up one cell, but other Excel formulas do work on one cell. Equally in Python the number of arguments to a function will depend on what it does.

In Python we can write our own functions so let's rewrite the Excel sum formula in Python.

- we have to start the function definition with `def`
- we have to choose a name (people normally use verbs for functions)
- we have to have round brackets followed by a colon
- we should either return or print something

```
[ ]: def replicate_excel(num1, num2):
    return num1 + num2
```

```
[ ]: replicate_excel(23432, 78907)
```

We can run this as often as we like. Although in this case there is already a built-in Python function called `sum`.

It's important to notice that we have to pass *exactly* two arguments to `replicate_excel` or we'll get an error:

```
[ ]: replicate_excel(23432, 78907, 51092)
```

Let write a new function to tell a user they're not allowed to do something. First we'll just print a message, so no arguments are required. But we still need the round brackets:

[ ]:
```
def say_no():
    print("I'm sorry, I can't let you do that.")
```

[ ]:
```
say_no()
```

Now we'll add an argument, so that we can put the username in the message.

[ ]:
```
def say_no(username):
    print(f"I'm sorry, I can't let you do that, {username}.")
```

Now we *must* provide a username.

[ ]:
```
say_no("Dave")
```

You can put logic inside the body of a function, as much as you like, but if a function gets really complicated you should probably think about splitting the function into several functions.

Let's revisit the `if` logic from above and either allow a user to do something or not, according to their status.

[ ]:
```
def say_yes_or_no(username, status):
        if status == 1:
            print(f"Your status is lowly, {username}. Go away!")
        elif status < 5:
            print(f"I'm sorry, {username}: you're not important enough to do␣
 ↪that.")
        else:
            print(f"Right you are, {username}. Happy to be of service.")
```

Now we need to call it with two arguments, in the correct order.

[ ]:
```
say_yes_or_no("Dave", 3)
```

A few weeks ago we look at the Dice similarity score, which compares two sets of values. If two sets are identical the score is 1; if two sets have nothing in common the score is 0.

In LaTeX the formula looks like this:

$$

$$\text{Dice}(A, B) = \frac{2|A \cap B|}{|A| + |B|} \tag{1}$$

$$

How can we implement this in Python?

The top line of the equation is finding the intersection of two sets and then multiplying it by two. We found the intersection of `set1` and `set2` above like this:

[ ]:
```
print(set1 & set2)
```

For the Dice calculation we need the length of the intersection (which we can see in this case is going to be five items), multiplied by two.

[ ]:
```
print(len(set1 & set2) * 2)
```

For the lower half, we need to add the length of `set1` to the length of `set2`:

[ ]:
```
print(len(set1) + len(set2))
```

Breaking it down like this, we can see that the result of the whole equation in this case should be 10/19.

[ ]:
```
print(10 / 19)
```

The whole equation in Python could be done like this. We'll also use fstrings to only give us three decimal places.

[ ]:
```
dice = (len(set1 & set2) * 2) / (len(set1) + len(set2))
print(f"{dice:.3}")
```

This is a good example of where you would want to define a function called something like `calculate_dice` and be able to run it multiple times with different arguments.

**Exercises**

- Write some logic to filter `mylist` into a unique list and a list of the duplicate numbers.
- Modify the dictionary code to print out "John was born in 1940" etc.
- Write the names of the older Beatles to a list.
- From `immutable_beatles`, use a slice to print "Ringo was the fourth immutable Beatle."
- Write your own dictionary and query it (it doesn't have to be about The Beatles).
- Write a function to calculate Dice similarity when given two sets as arguments.