# MATH40006: An Introduction To Computation
## Module notes, Section 2

## 2 Data in Python

### 2.1 Types of number

An important thing to understand in Computing is the idea of **data types**. Computers store data as binary code: as sequences of ones and zeros. There are then interfaces to convert that binary data into a human-readable form and vice versa, and also algorithms for operating with the data: for adding and multiplying two numbers, for example.

Exactly how the interfacing works, and exactly what the algorithms do, depends on the **type** of data we're talking about: computers don't set about adding two integers using the same algorithm as for adding two approximate decimals, for example.

In Python, this starts with three different types of **number**: `int`, `float` and `complex`.

- `int` is short for "**integer**";
- `float` is short for "**floating point number**"; this is your computer's way of representing what humans read as approximate decimals; except, of course, the internal representation is *binary*;
- `complex` is self-explanatory; a `complex` has a real part and an imaginary part.

Let's start with `int`. Typing

```
int1 = 345
int2 = 456
```

and Shift-Returning, sets up two variables, called `int1` and `int2`, with integer values. Then typing

```
type(int1)
```

should return the output `int`. Tihis tells us how the computer is storing the number, and what algorithms are used when we do arithmetic with it; try

```
int1 + int2
```

```
801
```

```
int1 - int2
```

```
-111
```

```
int1 * int2
```

157320

etc.

Remember that

```
int2 / int1
```

returns an approximate decimal (actually what we're now calling a `float`). To get the **integer quotient**, which is an `int`, you'd type

```
int2 // int1
```

1

The command `int2 % int1` returns the residue of `int2` modulo `int1`, which is `111`. Finally, `int1 ** int2` returns the value of $345^{456}$, which is the rather hefty integer

```
175997846325664414674033971221357038437100610683785278935124804372532 0
669755869321930748488415713851566344649275244159145011241646626616105 1
479230493334156255307739720847087765794478116736839562726727256906201 5
761487146937633581777327692566969482497081923341439823184421944047618 6
039590851246555318581079782137566513217991662102917260164246643117696 3
351584034018558113169085771276288402427763125673952147190047654184791 6
100560762473933684719315854353631007339600925268320502339335235730095 7
549495404151895835838613658621838701693522287406439595027462217030787 0
261518401789781188903912439331247690262779515803013069667135295225961 2
681608875244926028792875699055361366841804170365821665336291970970327 1
803809379049990420380620500111140136112288958083082639589970305996030 7
149230193867232087759903206273650133471173463551443535596719085348548 2
034749594613456672614497180521111036775072408110223718920315175719415 4
793007739668551009055616780796594190622520893702890479854387765602748 3
490475700657584782227039301120017582504922074128056015356548878729272 6
283534749796495191833357791865544606786478268218119460474603948083906 2
130179684118047589436173439025878906251
```

Notice that Python neither tries to round this number nor reports an error: an `int` can be as large as your computer will handle!

Now for `floats`. Let's set up

```
float1 = 345.0
float2 = 456.0
float3 = 23.456
```

and check that `type(float1)`, say, does indeed return the value `float`. We can then do arithmetic in much the same way as with `ints`, although behind the scenes the algorithms the computer is using will be different.

There are other, more obvious differences. If you try, for example

```
float3 ** float3
```

you should get the output

```
1.3825368655381606e+32
```

This is computer-ese for $1.3825368655381606 \times 10^{+32}$; floats that are very big, or very close to zero, will be represented like this. As you see, Python is rounding, and representing the number in "scientific notation" (or "standard form", as you may know it); this is not something it ever does with `ints`. What's more, if you type

```
float2 ** float2
```

you'll get an **overflow error**: Python allows `ints` of arbitrary size, but for `floats` there's a size limit.

You can convert `ints` to `floats` and vice versa:

```
float(int1)
```

```
345.0
```

```
int(float1)
```

```
345
```

Typing `int(float3)` rounds down to 23.

Finally, `complex`. You can either define a complex number explicitly, like this:

```
comp1 = 2 - 1j
comp2 = 2.4 - 4j
```

or in terms of existing `ints` and `floats`, like this:

```
comp3 = complex(float1, float2)
print(comp3)
```

Something to notice is that

```
comp1.real
```

and

```
comp1.imag
```

return `2.0` and `-1.0` respectively; we might have expected `ints` here, but what we get is always `floats`.

## 2.2 `True` and `False`

So that's three numerical data types: `int`, `float` and `complex`. But not all data consists of numbers.

The Boolean data type consists of two values, `True` and `False`. These values serve as the outputs for certain kinds of input in Python, including those based on the operators <, >, <=, >= and ==:

```
print(2 < 4)
print(4 < 2)
print(5 < 5)
print(5 <= 5)
print(5 == 5)
```

```
True
False
False
True
True
```

Just as we can combine numerical data using operators such as +, −, *, /, // and %, so we can combine Boolean data. The two most important operators are `and` and `or`. Typing

`<expr1> and <expr2>`

returns `True` if both `<expr1>` and `<expr2>` are `True`, and `False` otherwise.

```
print(2 < 4 and 4 % 2 == 0)
print(2 > 4 and 4 % 2 == 0)
print(2 < 4 and 4 % 2 == 1)
print(2 > 4 and 4 % 2 == 1)
```

```
True
False
False
False
```

Typing

`<expr1> or <expr2>`

returns `True` if one or other of `<expr1>` or `<expr2>`, or both, are `True`, and `False` otherwise.

```
print(2 < 4 or 4 % 2 == 0)
print(2 > 4 or 4 % 2 == 0)
print(2 < 4 or 4 % 2 == 1)
print(2 > 4 or 4 % 2 == 1)
```

```
True
True
True
False
```

There's also `not`, which changes `True` into `False` and vice versa.

```
print(not 2 < 4)
print(not 2 > 4)
```

```
False
True
```

Something whose value can be `True` or `False` is called a **Boolean expression**.

## 2.3 Data structures

We can think of a a number or a Boolean valua as a single piece of data; I now want to look at *collections* of data. A collection of data that we hold together as a single thing is called a **data structure**. Python has many kinds of data structure, and you'll meet several more on this module, but for now let's focus on three: **strings**, **lists** and **tuples**.

### 2.3.1 Strings

First strings. A string is an ordered collection (what in Python is called a **sequence**) of **characters**. Strings in Python can be represented using either single quotes...

```
string1 = 'Python makes me feel dumb'
```

... or double quotes:

```
string2 = "struck with admiration"
```

This flexibility allows us to embed quotes within a string:

```
string3 = "; I 'really mean' that!"
```

We join strings in Python using the + operator:

```
string1 + string2 + string3
```

It's possible to pull out individual characters from a string, using what's called **indexing**. If you type

```
string1[1]
```

what gets returned is the character of `string1` whose index is 1. One might think that that would be `'P'`, but actually, in Python the indexing starts from zero, so `string1[1]` is `'y'`. To get `'P'`, you'd need to type `string1[0]`.

We can also extract substrings; this is called **slicing**. If you type

```
string1[0:6]
```

this returns the substring `'Python'`.

But hang on a minute. This substring consists of six characters, meaning that the indexes represented are 0, 1, 2, 3, 4 and 5. This is also, perhaps, a bit unexpected: typing `string1[0:6]` returns the characters whose indexes are **greater than or equal to** 0, but **strictly less than** 6. This asymmetry in the indexing convention for slicing can take a bit of getting used to.

The input

```
string1[0:10:2]
```

returns `'Pto a`. This slice consists of the characters with indexes 0, 2, 4, 6 and 8: that is, those whose indexes are greater than or equal to 0, and strictly less than 10, **going up in steps of 2**.

To get the number of characters in a string, type

```
len(string1)
```

```
25
```

Something that turns out to be surprisingly useful is the ability to *split* a string at all occurrences of a certain character. To see what that does, type

```
splitstrings = string1.split('e')
print(splitstrings)
```

```
['Python mak', 's m', ' f', '', 'l dumb']
```

The output is a collection of substrings (actually a **list** of substrings; more about lists very soon). The boundaries of the substrings are where the character `'e'` occurs in the original string. If you don't specify a character in this way, the space character is used.

Notice that we *don't* type `split(string1, 'e')`, which is what we might expect. Something like `len` is called a Python **function**; something like `split`, which comes after a dot in this way, is called a **method**. More about functions later, and more about methods quite a lot later.

The method that undoes the effect of `split` is called, unsurprisingly, `join`, and it works like this:

```
 'e'.join(splitstrings)
```

```
'Python makes me feel dumb'
```

The `replace` method works like this:

```
 string1.replace('dumb', 'smart')
```

```
'Python makes me feel smart'
```

Typing `string1.lower()` converts all letters to lower case, and `string1.upper()` gives all caps.
Python has a feature called **string formatting**. Here's an illustration of how that works:

```
 template = 'The radius of {} is {} metres.'
```

```
 template.format('Jupiter', 69911000)
```

```
'The radius of Jupiter is 69911000 metres'
```

```
 template.format('Earth', 6371000)'
```

```
'The radius of Earth is 6371000 metres'
```

This approach to string formatting, using a template together with the `format` method, is ideal if, as in the above example, you're using a template more than once. If you're string formatting as a one-off act, though, Python 3.6 and above have a way of doing it that's a little more intuitive and easy to read.

Suppose I want to generate three random integers and generate a string that reports the numbers themselves and the value of their maximum; suppose, too, that I only want to do this once. Here's how I could do that using `format`:

```
 from random import randint
 a = randint(1,100)
 b = randint(1,100)
 c = randint(1,100)

 'The maximum of {}, {} and {} is {}.'.format(a,b,c,max(a,b,c))
```

```
'The maximum of 38, 20 and 85 is 85.'
```

And here's the modern, readable way to do it, using what's called a **formatted string literal** (or **f-string** for short):

```
from random import randint
a = randint(1,100)
b = randint(1,100)
c = randint(1,100)

f'The maximum of {a}, {b} and {c} is {max(a,b,c)}.'
```

```
'The maximum of 81, 39 and 13 is 81.'
```

(Notice the `f` in front of the string.) If your version of Python is older than 3.6 (unlikely!) then this won't work.

### 2.3.2 Lists

A string is a sequence of characters. A **list** is a sequence of *anything*. Let's start by typing

```
list1 = [1, 2, 3, 4, 5]
```

This is a list of `int`s. If you type

```
list2 = ['one, 'two, 'three', 'four', 'five']
```

you'll have set up a list of strings. And if you type

```
list3 = list1 + list2
```

you get a list some of whose elements are `int`s, and some of which are strings: check it out by typing

```
list3
```

```
[1, 2, 3, 4, 5, 'one, 'two, 'three', 'four', 'five']
```

Literally anything can go in a list: we can even make a list of lists:

```
list4 = [list1, list2, list3]
list4
```

```
[[1, 2, 3, 4, 5'],
 ['one, 'two, 'three', 'four', 'five'],
 [1, 2, 3, 4, 5, 'one, 'two, 'three', 'four', 'five']]
```

We pick out elements by indexing in the same way as with strings:

```
list1[1]
```

2

(Note that the indexing conventions are the same as for strings; the element with index 1 is the second element!)

We get sublists by slicing in the same way as with strings too:

```
list3[0:7]
```

```
[1, 2, 3, 4, 5, 'one, 'two']
```

(these are the elements with indexes 0 to 6), or

```
list3[0:7:2]
```

```
[1, 3, 5, 'two']
```

(these are the elements with indexes 0 to 6, going up in steps of 2).

One neat thing we'll make a lot of use of is **appending**. If you type

```
list1.append('six')
```

you don't get any output; what's happened is that the value of `list1` has changed. You can see this by typing

```
list1
```

```
[1, 2, 3, 4, 5, 'six']
```

You can always, then, lengthen a list by one, by appending an extra element to it.

A special, very useful kind of list is one consisting of equally-spaced integers. We've already typed one of those in by hand, but here's a slicker way:

```
list5 = list(range(6))
```

If you now type

```
list5
```

you'll get the output

```
[0, 1, 2, 3, 4, 5]
```

### 2.3.3 Tuples and immutability

Most computing languages support some kind of data structure in the form of a one-dimensional sequence of pieces of data, though the details of how this works vary enormously from language to language. One of the basic data structures in Python is, as you've seen, the **list**.

Python actually supports another data structure that is in most respects extremely similar to the list, with a few crucial differences (actually, really, with just one crucial difference). This is called the **tuple**.

The most obvious difference between a list and a tuple is in the way they're represented to users: a list uses square brackets and a tuple uses round ones.

**Challenge 1**: create and print a list, `list1` and a tuple, `tuple1`, each containing the data `0, 'one', 2.0, (3+0j) and 'IV'`.

```
list1 = [0, 'one', 2.0, (3+0j), 'IV']
tuple1 = (0, 'one', 2.0, (3+0j), 'IV')
print(list1)
print(tuple1)
```

```
[0, 'one', 2.0, (3+0j), 'IV']
(0, 'one', 2.0, (3+0j), 'IV')
```

That's not a very big difference, though, and in many respects they're extremely similar.

**Challenge 2**: extract and print element number 3 (starting with 0) of `list1` and `tuple1`.

```
print(list1[3])
print(tuple1[3])
```

```
(3+0j)
(3+0j)
```

**Challenge 3**: print a list consisting of elements $0, 2$ and $4$ of `list1`, and a tuple consisting of elements $0, 2$ and $4$ of `tuple1`.

```
print(list1[0:5:2])
print(tuple1[0:5:2])
```

```
[0, 2.0, 'IV']
(0, 2.0, 'IV')
```

So what's the difference between them? Well, here's an illustration.

**Challenge 4**: append the value `'cinq'` to `list1`, then try to do that with `tuple1`

Now, if we go first with the list, it works fine:

```
list1.append('cinq')
print(list1)
```

```
[0, 'one', 2.0, (3+0j), 'IV', 'cinq']
```

But if we try it with the tuple, it all goes wrong:

```
tuple1.append('cinq')
print(tuple1)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-29-34955e41cdef> in <module>()
----> 1 tuple1.append('cinq')
      2 print(tuple1)

AttributeError: 'tuple' object has no attribute 'append'
```

So that's one difference: tuples are like lists except you can't append to them. If we wanted to solve this problem, we'd need to do something like this:

```
tuple1 = tuple1 + ('cinq',)
print(tuple1)
```

```
(0, 'one', 2.0, (3+0j), 'IV', 'cinq')
```

(Note in passing that if we want a tuple with one element, like the tuple containing just `'cinq'`, we need to type `('cinq',)` and not `('cinq')`; the latter, annoyingly, is the same as `'cinq'`.)

Another solution would be to convert our tuple into a list and back again:

```
tuple1 = (0, 'one', 2.0, (3+0j), 'IV')
temp_list = list(tuple1)
temp_list.append('cinq')
tuple1 = tuple(temp_list)
print(tuple1)
```

```
(0, 'one', 2.0, (3+0j), 'IV', 'cinq')
```

The other main difference (actually, it's really another aspect of the same difference) is shown by this example:

**Challenge 5**: with these new six-element definitions of `list1` and `tuple1`, change the final element, `'cinq'`, to `'vijf'`.

Once again, with the list, it works fine:

```
list1[5] = 'vijf'
print(list1)
```

```
[0, 'one', 2.0, (3+0j), 'IV', 'vijf']
```

But again if we try it with the tuple, no such luck:

```
tuple1[5] = 'vijf'
print(tuple1)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-33-c3c524102390> in <module>()
----> 1 tuple1[5] = 6.0
      2 print(tuple1)

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

So tuples are the same as lists, except (a) we can't append to them and (b) we can't give their elements new values. We say that lists are **mutable** and tuples are **immutable**.

But isn't this a bit like saying tuples are the same as lists, except worse? What's the point of having an immutable data type? Unfortunately, the answer to that one is going to have to wait till later in the module; there is an advantage, in certain contexts, in having data whose value you know is stable.

## 2.4 The `zip` function

Suppose you have two collections of data in the form of sequences (lists or tuples, for example), and they're the same length, and the data items correspond in some way. You may want to combine them into a single collection consisting of paired data. One neat way to do this is using the `zip` function.

```
xvals = list(range(6))
yvals = (0,1,4,9,16,25)
combined = zip(xvals,yvals)
```

In Python Version 3, this creates a "lazy" object, whose contents aren't visible:

```
print(combined)
```

```
<zip object at 0x0000014126ED9680>
```

We can get access to the contents by converting this object into a list or a tuple; we see that its contents consist of 2-tuples.

```
print(list(combined))
```

```
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```