MATH40006: An Introduction To Computation
Module notes, Section 4

These notes, together with all the other resources you'll need, are available on Blackboard, at

`https://bb.imperial.ac.uk/`

---

4 Branching; iterable objects; comprehensions

4.1 `if` and branching

One of the main ideas of programming is **iteration**: getting the computer to do the same thing over and over again. The other big idea to get to grips with is **branching**: getting the computer to make a choice about what to do dependent on some condition or other.

> **Challenge 1**: write a program that checks whether 8 is a factor of 24, and if it is, prints an appropriate message. Repeat for 9 instead of 8.

Clearly, 8 *is* a factor of 24: if we type

```
24 % 8
```

we get 0, and if we type

```
24 % 8 == 0
```

we get `True`. (Remember, to check whether two quantities are equal in Python, we use the double equals sign, `==`; the single equals sign is reserved for **assigning a value to a variable**.)

So: what we want is for Python to run a check to find out whether 8 is a factor of 24, and if it is, print something (and if it isn't, do nothing). We might do that like this.

```
if 24 % 8 == 0:
    print("Hooray! 8 is a factor of 24")
```

```
Hooray! 8 is a factor of 24
```

If you replace the 8s with 9s...

```
if 24 % 9 == 0:
    print("Hooray! 9 is a factor of 24")
```

... you get no output at all.

Note: don't get `if` mixed up with `while`!

In some ways, `if` and `while` are a bit similar. Both involve the computer checking the value of a condition, and if that value is `True`, executing some code. But with `if`, the code is executed just once; with `while`, the code is executed repeatedly until the condition becomes `False`. If the program was

```
while 24 % 8 == 0:
    print("Hooray! 8 is a factor of 24")
```

it would run forever, printing the string again and again until we force-broke it. So this can be quite a costly mistake to make!

In this first example, the logic was "If the condition is `True`, execute this code; it it isn't, do nothing at all." We can also build programs for which the logic is "If this condition is true, execute this piece of code, and if it isn't, execute this other piece of code."

Challenge 2: write a program that checks whether 8 is a factor of 24, prints an appropriate message depending on whether it is or not. Repeat for 9 instead of 8.

First for 8:

```
if 24 % 8 == 0:
    print("Hooray! 8 is a factor of 24")
else:
    print("Oh no! 8 is not a factor of 24")
```

```
Hooray! 8 is a factor of 24
```

Now for 9:

```
if 24 % 9 == 0:
    print("Hooray! 9 is a factor of 24")
else:
    print("Oh no! 9 is not a factor of 24")
```

```
Oh no! 9 is not a factor of 24
```

Now let's embed a branch in a loop.

Challenge 3: write a program that iterates through the integers between 1 and 24, printing the values of any that are factors of 24.

We need to be a bit careful with our `range` object here. Remember that we want those integers that are greater than or equal to 1, and strictly less than 25; the right way to get those is `range(1, 25)`. Here's a program for printing all of them:

```
for n in range(1, 25):
    print(n)
```

```
1
2
3
4
5
6

...

24
```

That's not what we want, though; we want to print only those numbers that are factors of 24. For that we need an `if`, inside the `for`:

```
for n in range(1, 25):
    if 24 % n == 0:
        print(n)
```

```
1
2
3
4
6
8
12
24
```

Notice the double-indenting for the line of code inside the `if`, inside the `for`.

Challenge 4: write a program that iterates through the integers between 1 and 24, appending the values of any that are factors of 24 to a list.

```
fac_list = []
for n in range(1, 25):
    if 24 % n == 0:
        fac_list.append(n)
print(fac_list)
```

```
[1, 2, 3, 4, 6, 8, 12, 24]
```

Challenge 5: write a program that iterates through the integers between 1 and 24, appending the values of any that are factors of 24 to one list, and those of any that *aren't* factors to another.

```
fac_list = []
nonfac_list = []
for n in range(1, 25):
    if 24 % n == 0:
        fac_list.append(n)
    else:
        nonfac_list.append(n)
print(fac_list)
print(nonfac_list)
```

```
[1, 2, 3, 4, 6, 8, 12, 24]
[5, 7, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]
```

Finally, let's write a program for calculating the maximum of a list of numbers.

Challenge 6: write a program to calculate the maximum of the numbers

```
x_list = [43, 62, 53, 75, 53, 3, 97, 44, 69, 47, 84, 63, 90, 80,
   34, 6, 29, 74, 86, 23]
```

One thing to realise is that we can run a `for` loop not only over a `range`, but also over a list. So we're allowed to type

```
for x in x_list
```

and the variable `x` will take on, successively, the values 43, 62, 53 and so on.

So here's our tactic. Set up a variable, representing the "maximum so far"; give it the initial value 0, which is less than all the positive integers in this list. Then, loop through `x_list`, checking whether each value is greater than the "maximum so far". If it is, update the value of the "maximum so far", making it this new number. If it isn't, do nothing.

Here's the code:

```
x_list = [43, 62, 53, 75, 53, 3, 97, 44, 69, 47, 84, 63, 90, 80,
       34, 6, 29, 74, 86, 23]
max_x = 0
for x in x_list:
    if x > max_x:
        max_x = x
print(max_x)
```

97

4.2 Iterable objects

We've now written quite a few `for` loops. Most of them have been over `range` objects, but the last one we wrote was over a **list**.

The structure of a `for` loop is always

```
for x in <something>:
    <do a thing>
```

What we're concerned with here is what the allowable `<something>`s are!

In fact, we can write a `for` loop over a `range` object, or a list, or a string; and that's just the start of it. Lists first:

Challenge 1: for the list

```
x_list = [43, 62, 53, 75, 53, 3, 97, 44, 69, 47, 84, 63, 90, 80,
   34, 6, 29, 74, 86, 23]
```

write a `for` loop that prints out all the square roots.

```
from math import sqrt
x_list = [43, 62, 53, 75, 53, 3, 97, 44, 69, 47, 84, 63, 90, 80,
       34, 6, 29, 74, 86, 23]
for x in x_list:
    print(sqrt(x))
```

```
6.557438524302
7.874007874011811
7.280109889280518
8.660254037844387
7.280109889280518

...

4.795831523312719
```

Now strings. For this next example, you need to know that the function called `ord` associates a unique numerical **character code** with each alphanumeric character; for example, `ord('n')` returns 110.

Challenge 2: for the list

```
spy_string = 'My name is Bond, James Bond.'
```

write a `for` loop that prints out each character code.

```
spy_string = 'My name is Bond, James Bond.'
for character in spy_string:
    print(ord(character))
```

```
77
121
32
110

...

46
```

(Notice that I've used the variable name `character`. Now, I needn't have; I could have called this variable `x`, or `ian`, or `spoon`. But this variable stands for a character, so calling it `character` helps with the readability of our code; helps to make it **self-documenting**.)

Here's our final challenge in this section.

Challenge 3: for each of $x = 33, 35, 37, 39, 41, 43, 45, 47$, print the integer value of

$$\frac{3\,x+1}{2}.$$

Do this in five different ways.

One way is by cheekily copying and pasting from the notes (this is rather frowned upon, but let's go crazy).

```
x_list = [33, 35, 37, 39, 41, 43, 45, 47]
for x in x_list:
    print((3*x+1)//2)
```

```
50
53
```

```
56
59
62
65
68
71
```

However, these numbers are equally spaced, meaning we could create `x_list` like this:

```
x_list = list(range(33, 48, 2))
```

So here's a second implementation:

```
x_list = list(range(33, 48, 2))
for x in x_list:
    print((3*x+1)//2)
```

```
50
53
56
59
62
65
68
71
```

(You could argue this isn't really different from the first implementation, but let's let that slide.)

But lists aren't the only things we're allowed to loop over; there are also `range` objects. Now, a `range` object isn't the same thing as a list. You can tell that if you type

```
x_range = range(33, 48, 2)
print(x_range)
```

```
range(33, 48, 2)
```

The difference is that a list's contents are all explicit and visible. A `range` object, by contrast, keeps its contents hidden; indeed, it doesn't even create them till they're needed. That means that ranges take up much less memory, which is why it's a good idea to use them when we can.

Here's our third implementation, using a `range`:

```
x_range = range(33, 48, 2)
for x in x_range:
    print((3*x+1)//2)
```

```
50
53
56
59
62
65
68
71
```

It's even possible to iterate across objects that are, at least potentially, **infinite**. One such construct is the `count` object, which needs to be important from a module called `itertools`:

```
from itertools import count
x_count = count(33, 2)
```

Now, this object contains the integers 33, 35, 37, 39, etc, **going on to infinity**. However, we don't need infinite memory; each number is created only when it's needed, so this infinity is merely a *potential* one.

But it still has to be treated with care; it can still cause an infinite loop if mishandled. Here's a program that uses `count`, forcing a break when we've got to our final number:

```
from itertools import count
x_count = count(33, 2)
for x in x_count:
    if x > 47:
        break
    else:
        print((3*x+1)//2)
```

```
50
53
56
59
62
65
68
71
```

That's four implementations so far. For the last one, we'll import another function from `itertools`, called `islice`. This takes a **slice** out of a `count` object, making it finite again; here, we want to slice out the first eight elements of `count(33, 2)`.

```
from itertools import count, islice
x_slice = islice(count(33, 2), 8)
for x in x_slice:
     print((3*x+1)//2)
```

```
50
53
56
59
62
65
68
71
```

Challenge 4: for the lists

```
x_list = [43, 62, 53, 75, 53, 3, 97, 44, 69, 47, 84, 63, 90, 80,
   34, 6, 29, 74, 86, 23]
y_list =[79, 70, 5, 0, 98, 96, 34, 42, 57, 46, 58, 66, 49, 17,
   80, 95, 58, 51, 95, 76]
```

write a `for` loop that prints out the absolute values of the differences.

```
for pair in zip(x_list, y_list):
    print(abs(pair[0]-pair[1]))
```

```
36
8
48
75
45
...
53
```

So, to summarise, it's possible to write `for` loops over include:

- ranges;
- lists;
- tuples;
- strings;
- `count` objects;

- `islice` objects;
- `zip` objects.

These types of object all belong to a Python class called `Iterable`.

As an illustration, let's set up six iterable objects:

```
s = 'Hello World'
r = range(10)
l = list(r)
t = tuple(r)
c = count()
i = islice(c, 10)
z = zip(l,c)
```

Then we can check the values of these things:

```
print(s)
print(r)
print(l)
print(t)
print(c)
print(i)
print(z)
```

```
Hello World
range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
count(0)
<itertools.islice object at 0x000001412367F650>
<zip object at 0x0000014126F13780>
```

We can check that they all belong to the Python **collection** known as `Iterable`, as follows:

```
from collections.abc import Iterable
print(isinstance(s, Iterable))
print(isinstance(r, Iterable))
print(isinstance(l, Iterable))
print(isinstance(t, Iterable))
print(isinstance(c, Iterable))
print(isinstance(i, Iterable))
print(isinstance(z, Iterable))
```

```
True
True
True
True
True
True
True
```

This means, as we've seen, that `for` loops can be written across them all; the same, as you'll see soon, is true of the powerful Python-specific programming construct called the **comprehension**. The first four are all members of the Python class `Sequence`:

```
from collections.abc import Sequence
print(isinstance(s, Sequence))
print(isinstance(r, Sequence))
print(isinstance(l, Sequence))
print(isinstance(t, Sequence))
print(isinstance(c, Sequence))
print(isinstance(i, Sequence))
print(isinstance(z, Sequence))
```

```
True
True
True
True
False
False
False
```

This essentially means we can do indexing and slicing on them:

```
print(s[3])
print(r[3])
```

```
3
3
```

4.3 Comprehensions

You've now met the two key components of the style of computer programming known as **procedural**, namely iteration and branching. There are some programming languages that aren't procedural, and that rely instead on other constructs, but many, many languages are procedural, and have either `for` and `while` loops, and things like `if`, or something very similar. So none of that stuff is really unique to Python.

There are certain tasks, though, that, while they can certainly be done with loops, are more easily done *in Python* using a construct that some other languages don't have, called a **comprehension**. Comprehensions are appropriate when you start with an **iterable sequence**, and want to do the same thing to each of its elements, ending up with a list.

Challenge 1: create a list containing 10 copies of the string "Hello World!".

We've already done this by appending a copy of the string to an empty list ten times using a `for` loop. Here's another way of doing it, which I think you'll agree is a bit neater.

```
hw_list = ['Hello World!' for n in range(10)]
print(hw_list)
```

```
['Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!', 'Hello World!']
```

Challenge 2: create a list of all the squares of the integers between 0 and 9.

```
sq_list = [n**2 for n in range(10)]
print(sq_list)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Now let's do our summation that converges slowly to π.

Challenge 3: calculate

$$\sum_{n=0}^{100} \frac{4 \times (-1)^n}{2\,n + 1}.$$

```
terms = [(4*(-1)**n)/(2*n + 1) for n in range(101)]
print(sum(terms))
```

```
3.1514934010709914
```

This may actually not be the best way to do this; the `for` loop version we've already done has the advantage that it doesn't have to create a list of 101 terms, but only a running total.

You can set up a comprehension over any iterable sequence: that is, over anything you can do a `for` loop over.

Challenge 4: create a list containing all the character codes in the string

```
spy_string = 'My name is Bond, James Bond.'
```

```
spy_string_ords = [ord(character) for character in spy_string]
print(spy_string_ords)
```

```
[77, 121, 32, 110, 97, 109, 101, 32, 105, 115, 32, 66, 111, 110,
100, 44, 32, 74, 97, 109, 101, 115, 32, 66, 111, 110, 100, 46]
```

Note, though, that the output from a comprehension is always a list. (Actually, that's not strictly true, as you'll see later in the module; but let's for the moment act as if it is: certainly, the output from a comprehension can't be a string, or a tuple, or a `range` object, or anything like that.)

4.3.1 Filtering conditions

It's possible to attach a **condition** to a comprehension, which means it only operates on, and the list at the end is only constructed from, some of the elements.

Challenge 5: create a list containing all the character codes in the string

```
spy_string = 'My name is Bond, James Bond.'
```

but only those in lower case.

```
spy_string_ords = [ord(character) for character in spy_string
                   if character.islower()]
print(spy_string_ords)
```

```
[121, 110, 97, 109, 101, 32, 105, 115, 111, 110, 100, 44, 97, 109, 101, 115,
111, 110, 100, 46]
```

Often, all we want to do is filter, and the comprehension doesn't actually do anything to the data.

Challenge 6: create a list containing all the lower case characters in the string

```
spy_string = 'My name is Bond, James Bond.'
```

```
chars = [character for character in spy_string
               if character.islower()]
print(chars)
```

```
['y', 'n', 'a', 'm', 'e', 'i', 's', 'o', 'n', 'd', 'a', 'm', 'e', 's',
 'o', 'n', 'd']
```

Challenge 7: create a list consisting of all the integers between 2 and 100 inclusive, then remove all the even numbers other than 2.

```
ints = list(range(2,101))
ints = [n for n in ints if n==2 or n % 2 > 0]
print(ints)
```

```
[2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37,
39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71,
73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99]
```