



Module 1 : Introduction to OR, Data Science and Programming

Session 1C : Python Programming Part 1

Dr Daniel Chalk

"And now for something completely different"

Why Python?

Python is now the most widely used programming language in the world

It is easy to use and has extensive *libraries* of code for common things we may want to do

It allows us to get *quick results* and build and test *iteratively*

Hello World!

There is a tradition in Computer Science that the first program anyone should write in a Programming Language is one that writes the words “Hello World!” to the screen.

(Look, back in my day (the 80s), getting a machine to display *anything* on your TV was pretty amazing stuff!)

Let's write a Hello World program in Python

The print() Function

The print() function displays text on the screen. The input to the function is what we want it to print.

Writing the Hello World! Program

In your script window (on the left), type the following :

```
print ("Hello World!")
```

Push F5 (or the big green Play button on the top toolbar) to run the program. Observe the output. Exciting, yes? :)

(We could have also written the print command in the Interactive Console, and it would have run the command immediately. Give it a try!)

Variable declarations

In many programming languages, we need to create a variable by *declaring* it. This means specifying its type alongside its name.

In Python, variables are *dynamic*, so we don't need to do this. Instead, we simply assign a value to a name to create the variable, and the type will be automatically set based on the value. We assign values in Python using the = operator. It basically says "Let *this* have a value of *this*".

e.g.

```
age = 52  
name = "Bob"
```

What types will Python assign to these variables?

Dealing with Spaces in Names

In programming languages, a space indicates a separation between instructions, values etc. So if we want to name something (such as a variable) with multiple words (as we do frequently) then we can't use spaces. There are two main conventions for dealing with spaces :



camelCase



snake_case

You can use whichever one you prefer. My preference is for snake_case and so that's what you'll see in most of the course.

Variable Types

Let's remind ourselves of the main variable types, and see how we use them in Python :

Numbers

int – integer numbers (whole numbers) `my_int = 27`

float – floating point numbers (numbers with up to 15 decimal places) `my_float = 32.451`

Strings

Sequences of characters. `my_str_1 = "Apple"`

Denoted using " or '.

`my_str_2 = 'Dan is the best teacher'`

List

Ordered sequences of items denoted using [].

`my_list = [1, 5, 3, "Matthew", True]`

Variable Types

Set

Unordered sequences of *unique* items denoted using { }.

```
my_set = {4, 1, 3, 5, 2}
```

Dictionary

Unordered collection of key-value pairs denoted using {x:y}

```
my_dict = {"Name": "Dan", "Age": 39}
```

Tuple

Ordered sequences of items, like a list, but immutable (once created, cannot be changed). Denoted using ()

```
my_tuple = ("Test", 1, 7, True, "Orange")
```

Boolean

Boolean values are either *True* or *False*. `my_boolean = False`

fStrings

There's a really nice feature in newer Python versions (3.6 onwards) called *fStrings* – “Formatted String Literals”.

These are strings where we can include formatting within the string to define where we want dynamic text. We often want to do this where we want to insert the value of a variable into a string of text.

e.g. “Her name is <<insert name here>>”

fStrings

To use fStrings, we simply put the character *f* ahead of the “ at the start of our string.

Then, we use { } to denote where we want to include a variable value, by giving the name of the variable.

For example :

```
f"The patient lives in {home_town}"
```

The Hello Dan Program

Let's now amend our Hello World program. Instead of saying hello to the world, we're going to make it more personal. We're going to say hello to you, personally!

Change your script to the following :

```
my_name = "Dan"  
print (f"Hello {my_name}")
```

Now run the program.

More fString features

You can do more than just put variable names in the curly brackets in fStrings. You can put any instruction that results in something that could be interpreted as a string (e.g. numbers).

```
print (f"The answer is {2+2}")

my_float = 4.678235468594900127458

# .2f here means "a floating point with a precision of 2 decimal places"
print (f"My number rounded to 2 decimal places is {my_float:.2f}")

frequency = "Sometimes"
hsma_master = "Dan"
the_enemy = "THE LINE"

print (f"{frequency} we want to write very long fStrings that need to span",
      f"multiple lines if we don't want {hsma_master} to tell us off for",
      f"going over... ... {the_enemy}")
```

```
The answer is 4
My number rounded to 2 decimal places is 4.68
```

```
Sometimes we want to write very long strings that need to span multiple lines if we don't want Dan to tell us off for going over... ... THE LINE
```

The old method

Although the use of fStrings is the preferred method for including dynamic text in print statements in newer versions of Python, you will likely see examples of the older method being used too.

```
my_name = "Dan"  
print ("Hello ", my_name)
```

```
my_name = "Dan"  
my_age = 40  
print ("Happy birthday ", my_name, ", you are ", my_age, " today.", sep="")
```

User Input

Sometimes we need to ask the user to input something in order to continue with the program.

In Python, getting input from the user is easy.

We simply tell Python that we need an input, any message we want to display to the user, and the name of the variable in which we want to store the input.

User Input Example

```
age = input("How old are you? : ")
```

The above will store the user input. But there might be a problem with it if we're storing something like an age. Does anybody know what it might be?

Casting

By default, inputs are read in as *strings* – sequences of characters that are treated as text. But for something like age, we probably want to work with it as a number.

To do this, we need to *cast* the variable as an *integer*. We do this by specifying the type to which we want to cast, and then wrapping the value we want to translate (or “cast”) in brackets.

We could read in the input and do this separately after :

```
age = input("How old are you? : ")  
age = int(age)
```

or we could just do it on the same line we take the input :

```
age = int(input("How old are you? : "))
```

Exercise 1

Using what you've learned so far, write a program that asks for the user's name, then asks for their age, before printing the message :

“Happy birthday <<name>>, you are <<age>> today”

You'll have 10 minutes, so feel free to stretch your legs if you finish early!

Comments

When writing programs, it is important to *comment* our code (leave notes explaining what various sections of code are doing) to make it easier for others to follow.

There are different ways to comment in Python :

```
# this is a comment, until we start a new line  
# at which point we need another hashtag
```

```
"""This is also a comment, and will continue to be so until  
the three speech marks close it again."""
```

Mathematical Operators

It's usually the case we want to manipulate our variable values in some way. When dealing with numbers, we will likely want to apply common mathematical functions to them :

```
a + b # a plus b  
a - b # a minus b  
a * b # a multiplied by b  
a / b # a divided by b  
a % b # a modulus b (return the remainder from a divided by b)  
a ** b # a to the power of b
```

```
a += 1 # this is shorthand for a = a + 1  
a -= 1 # this is shorthand for a = a - 1
```


Conditional Logic

In *Principles of Programming*, we talked a lot about Conditional Logic.

In Python, we can apply Conditional Logic using *IF / ELIF / ELSE* statements.

Let's consider an example.

Conditional Logic Example

Let's imagine we need to print a different message to the user depending on whether or not someone is under 60 or aged 60+.

We might write the following :

```
if age < 60:  
    print("Please go to room A")  
else:  
    print("Please go to room B")
```

Note the tab indentations – these are important in Python to indicate blocks of code, such as the block within a condition in an if statement

Comparison Operators

There are many comparison operators in Python. They help form relational statements that return a *Boolean* value of *True* or *False*.

```
a == b # value of a is equal to value of b
a != b # value of a is not equal to value of b
a > b # value of a is greater than value of b
a < b # value of a is less than value of b
a >= b # value of a is greater than or equal to value of b
a <= b # value of a is less than or equal to value of b
a == b and a < c # value of a is equal to value of b AND less than value of c
a == b or a < c # value of a is equal to value of b OR less than value of c
```

elif

Sometimes we don't just have two conditions to check. If we have more than two, it can be useful to use the “else if” command – *elif*

```
if age < 16:  
    print ("Please go to the children's room")  
elif age < 60:  
    print ("Please go to room A")  
else:  
    print ("Please go to room B")
```


Exercise 2

Let's reflect on what you've learned so far :

- Printing to the display
- Requesting and storing inputs from the user
- Variables and casting variables
- Mathematical operators
- Conditional Logic using IF, ELIF and ELSE

Switch to Jupyter Lab. Spend the next **35 minutes (+ 5 minute break)** working through the Jupyter Notebook “python_prog_workbook_1.ipynb”. Remember, you can test if your code cell is working by running the cell (CTRL + Enter)

If you're unable to use Jupyter Lab, you can either :

- open Google CoLab and import the notebook to work on it in there, or
- open the file on the GitHub repository to see the questions, and write your solutions in Spyder

You should each work through the exercises, but you'll be in your groups so please do help each other!

Loops

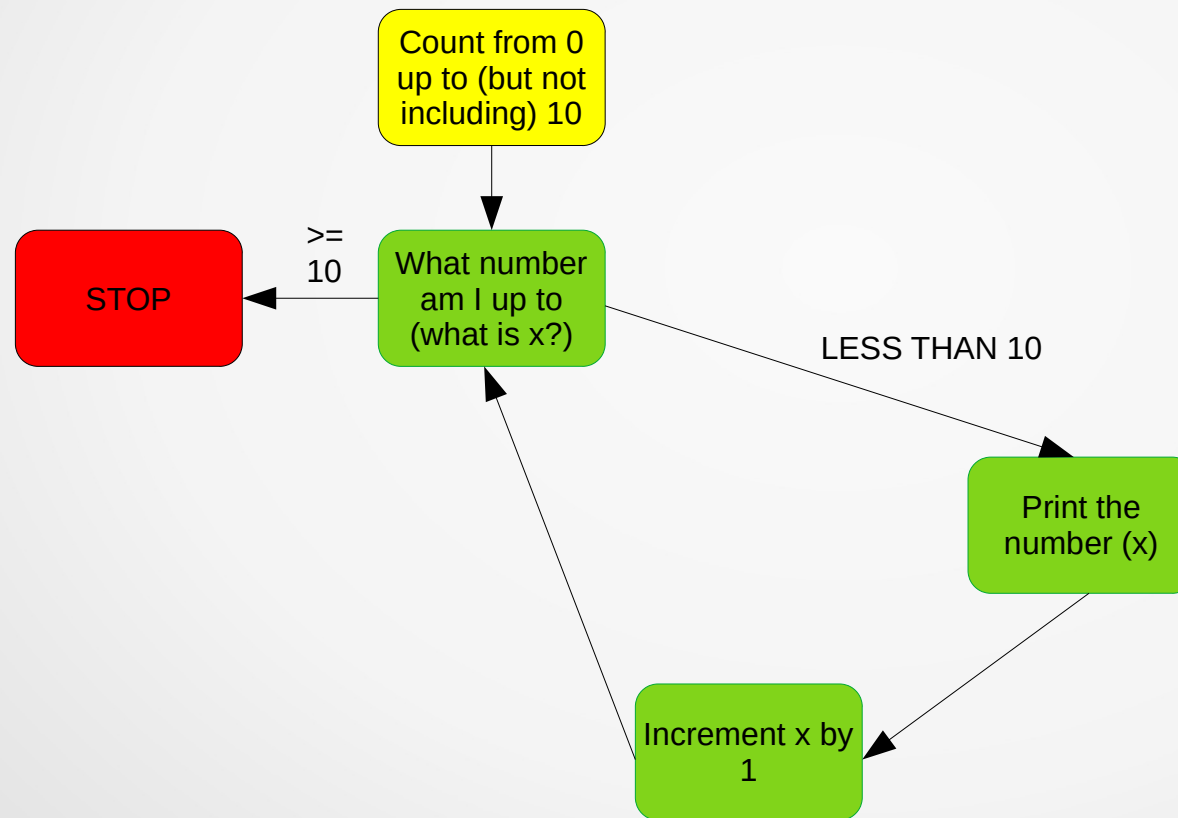
Remember from our Principles of Programming session we said that, often, we need to perform the same operation a number of times. It might be that :

- we need to do the same thing for a given number of iterations
- we need to continue to do something whilst a certain condition is true

We can use two different types of loop to achieve the above – *for* loops and *while* loops

For Loops

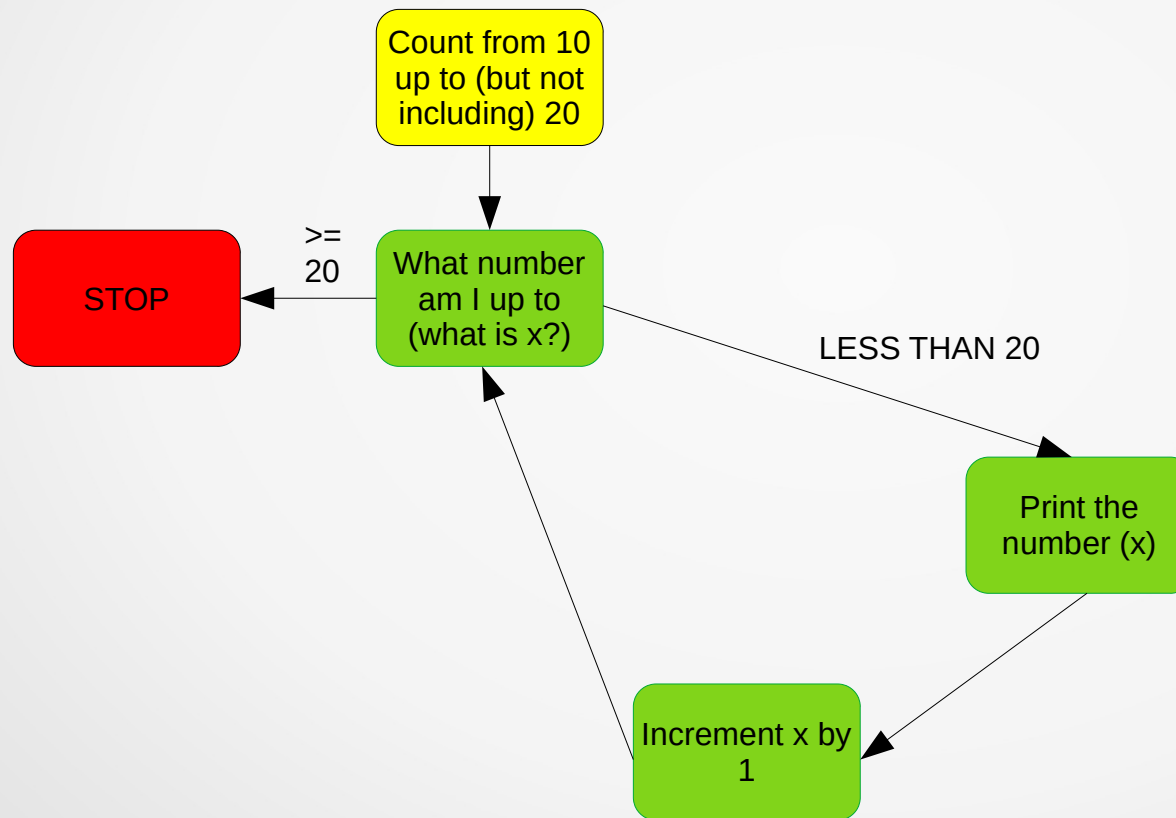
```
for x in range(10):  
    print(x)
```



0
1
2
3
4
5
6
7
8
9

For Loops – Custom Range

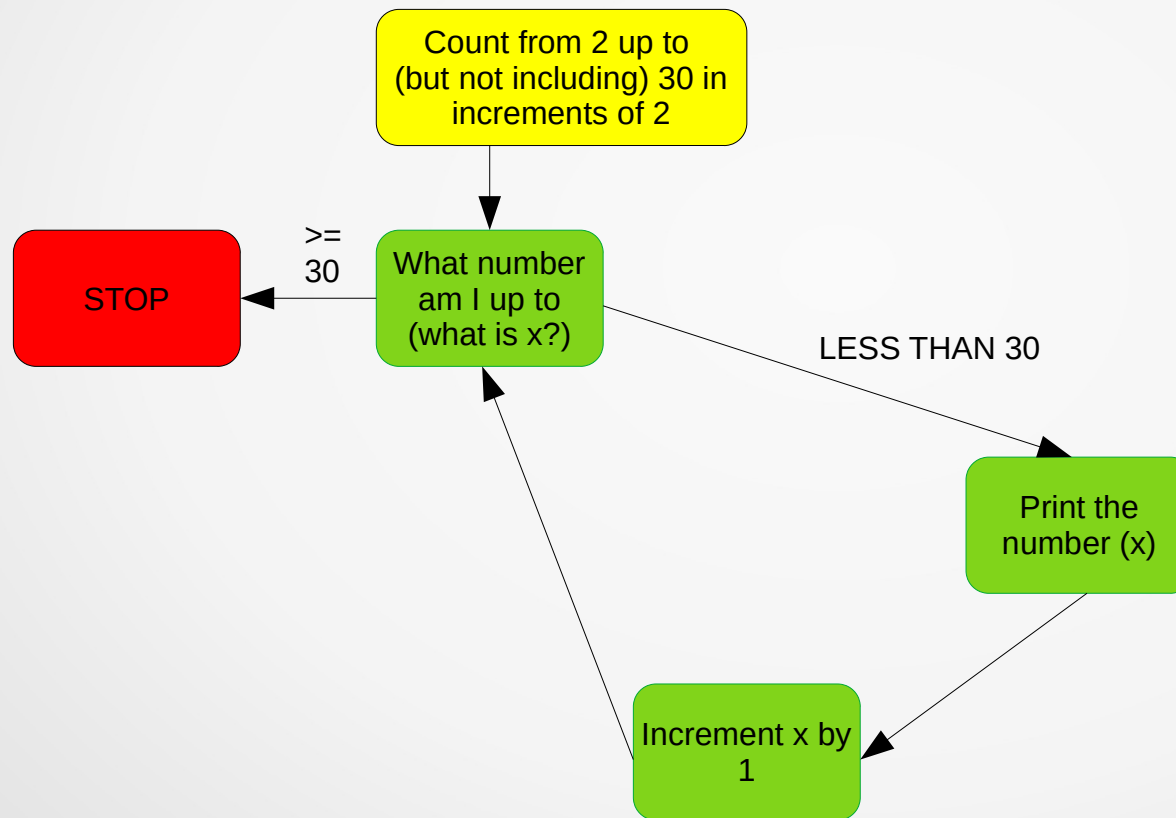
```
for x in range(10, 20):  
    print(x)
```



10
11
12
13
14
15
16
17
18
19

For Loops – Custom Increment

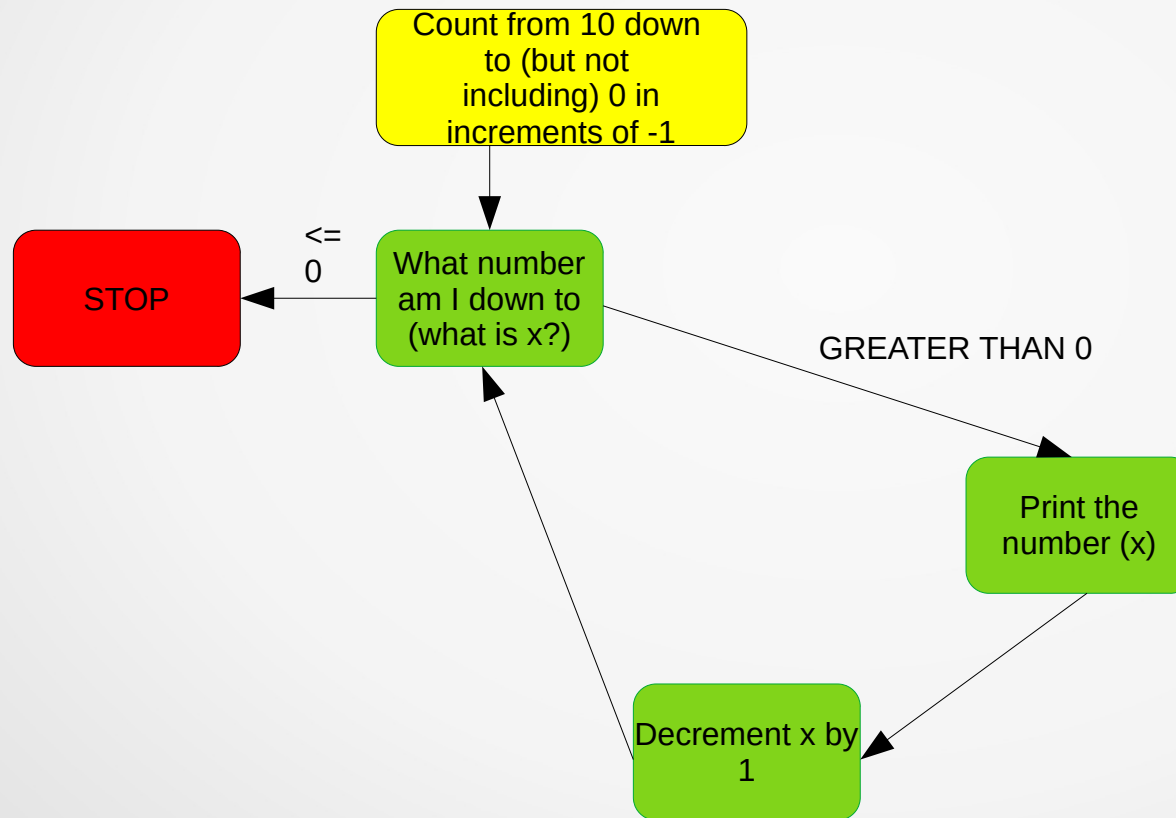
```
for x in range(2,30,2):  
    print(x)
```



2
4
6
8
10
12
14
16
18
20
22
24
26
28

For Loops – Negative Increment (Decrement)

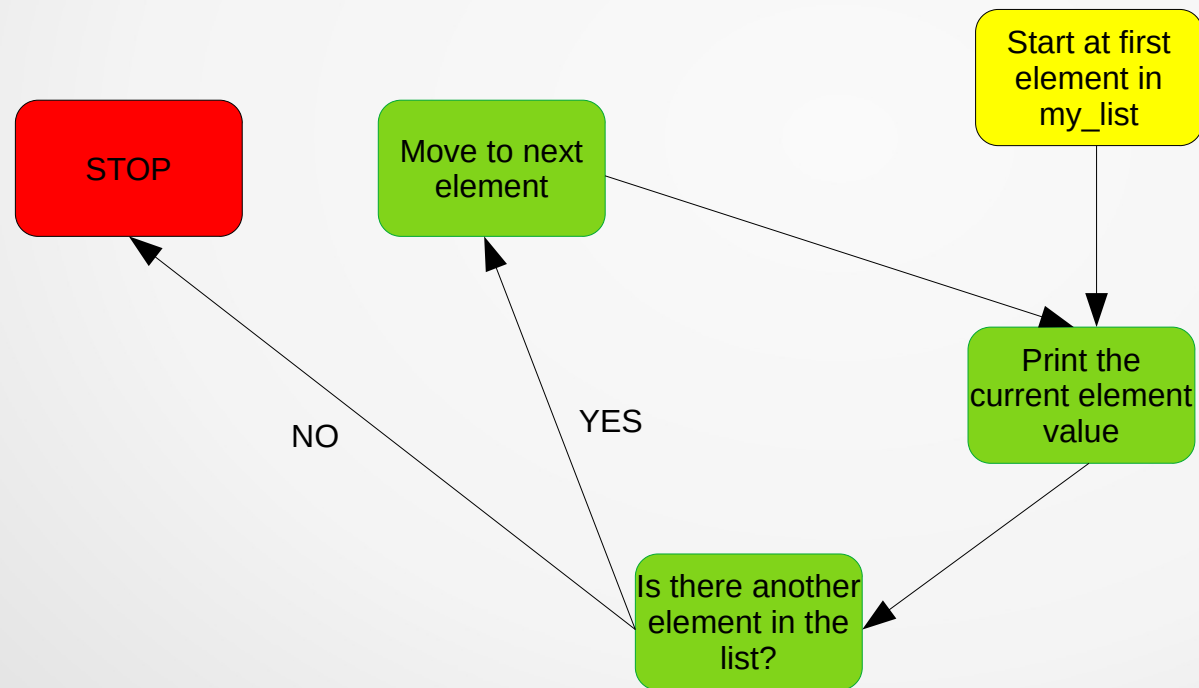
```
for x in range(10,0,-1):  
    print(x)
```



10
9
8
7
6
5
4
3
2
1

For Loops – Iterate through a List

```
my_list = [1,10,"Dan", True, "HSMA"]  
  
for element in my_list:  
    print(element)
```

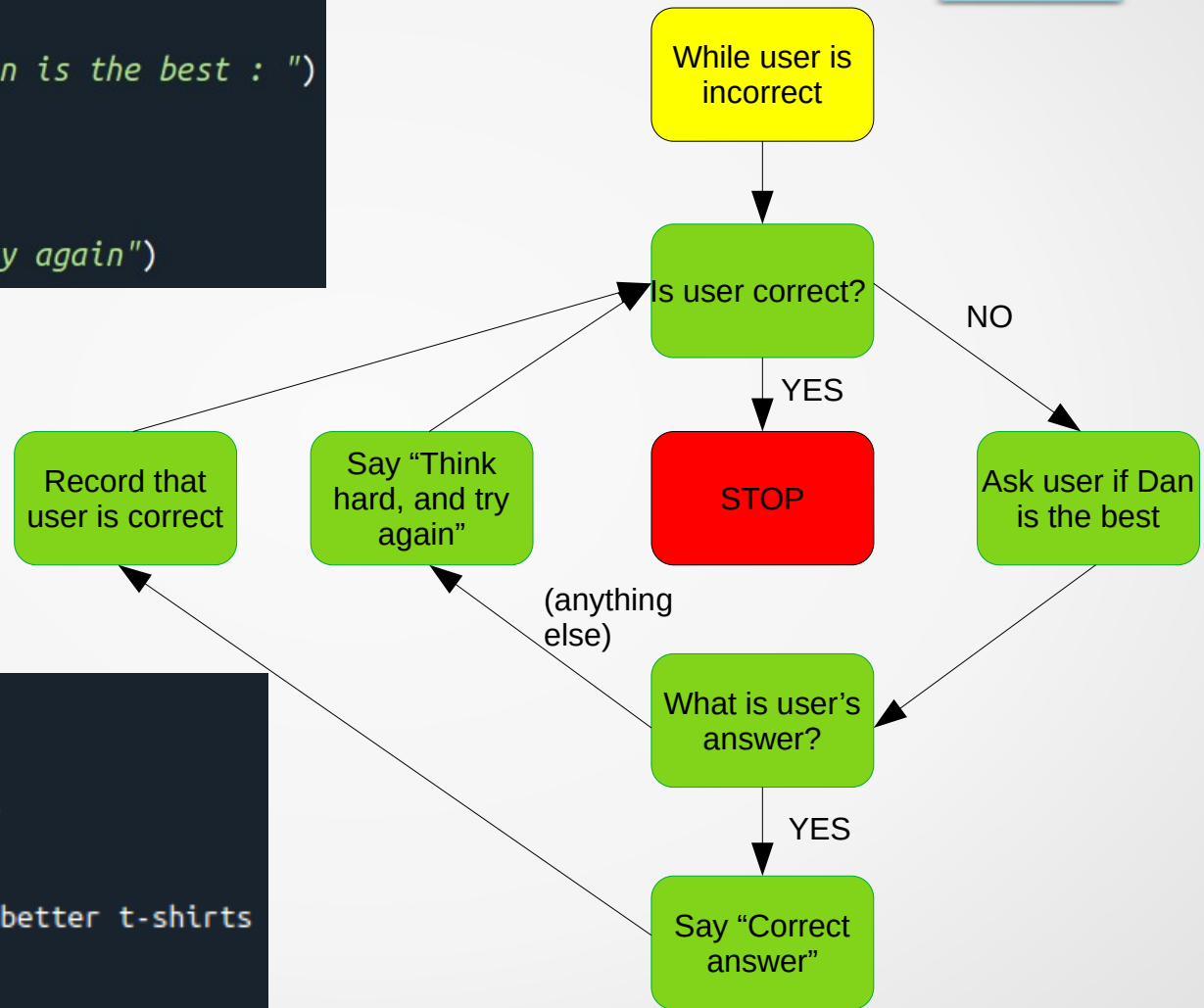


1
10
Dan
True
HSMA

While Loops

```
correct = False

while correct == False:
    answer = input("Type YES if Dan is the best : ")
    if answer == "YES":
        print("Correct answer")
        correct = True
    else:
        print("Think hard, and try again")
```



```
Type YES if Dan is the best : NO
Think hard, and try again
```

```
Type YES if Dan is the best : It's Mike
Think hard, and try again
```

```
Type YES if Dan is the best : He's got better t-shirts
Think hard, and try again
```

```
Type YES if Dan is the best : Sighs...
Think hard, and try again
```

```
Type YES if Dan is the best : YES
Correct answer
```

Breaking from a loop

Sometimes we want to break out of a loop mid-flow. For example, we may have a for loop, and want to break out of it when a condition has been met.

We can do this using the *break* command, which *immediately* breaks out of the loop and continues as though the for loop had completed.

Example :

```
total = 0
for x in range(10):
    total = total + x
    print (total)

    if total > 5:
        break
```

0
1
3
6

Infinite Loops

Sometimes you can accidentally write an *infinite loop* – one that will never end. Anybody who used to play with BASIC in the old days will remember an example of an infinite loop :

```
10 PRINT "HELLO WORLD"  
20 GOTO 10
```

More modern examples include setting up a while loop that can never stop :

```
x = 3  
while (x >= 3):  
    print(x)  
    x += 1
```

```
while True:  
    print("Dan is the best")
```

If this happens to you (and it will at some point) – don't worry; just hit CTRL + C to interrupt and terminate the program (make sure you're in the iPython console sub-window if you're in Spyder). Sometimes, an infinite loop can be useful – for example you may want the user to stay in the system making inputs until they choose to close the system, at which point you can use a break statement.

Exercise 3

You've now learned about :

- For Loops
- While Loops
- Breaking from Loops

Switch to Jupyter Lab. Spend the next 30 minutes working through the Jupyter Notebook “python_prog_workbook_2”. Remember, you can test if your code cell is working by running the cell (CTRL + Enter whilst in the cell)

Working with Lists : Create and Append

Lists are very useful ways to store multiple items of data, to which we can refer later.

```
# Define a new empty list
my_empty_list = []

# Define a new list with some starting elements
my_list = [3, 4, 7, "hello"]

# To add an item to a list, we use the append function
my_list.append(2)

print (my_list)
```

```
[3, 4, 7, 'hello', 2]
```

Working with Lists : Extend

We can also use the extend function to add multiple items to a list – we give it a list, and it adds all the items in that list to the list to which we want to add :

```
my_list = [1,2,3,4,5]  
my_list.extend([6,7,8,9,10])  
print(my_list)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Working with Lists : Indices

To refer to a specific item in a list, we use `[x]` notation, where `x` is the index of the element we want to refer to.

REMEMBER : In Python, as with most programming languages, we start counting from 0. So the second element would have an index of 1.

```
print (my_list[2])
```



7

0	1	2	3	4						
[3	,	4	,	7	,	'hello'	,	2]

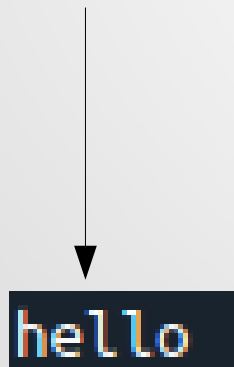


Working with Lists : Negative Indices

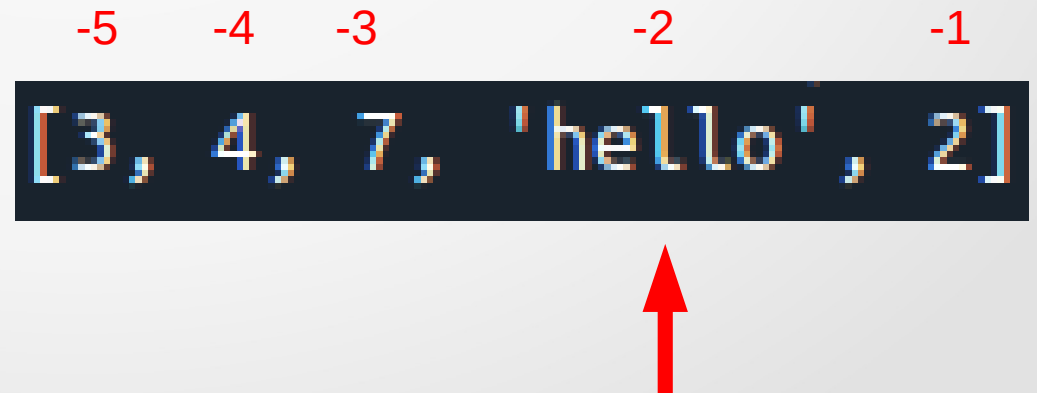
We can also use negative indexing to refer to an item based on its position from the end of the list.

An index of -1 would give the last item in the list.
An index of -2 would give the penultimate item in the list etc

```
print (my_list[-2])
```



hello



-5 -4 -3 -2 -1

[3, 4, 7, 'hello', 2]

Working with Lists : Index Ranges

We can refer to multiple items in a list, rather than a single element.

```
my_list = [3, 4, 7, "hello", 2]

# Print everything from 2nd element up to (but not including) 4th element
print (my_list[1:3])

# Print everything up to (but not including) 3rd element
print (my_list[:2])

# Print everything from 4th element onwards
print (my_list[3:])
```

```
[4, 7]
[3, 4]
['hello', 2]
```


Working with Lists : Length and Pop

```
# We can remove items from a list using either remove() or pop()
# remove() removes a specified item
my_list.remove("hello")
print (my_list)

# Let's restore the list
my_list = [3, 4, 7, "hello", 2]

# pop() removes an item specified by its index in the list
# or removes the last item in the list if no index is specified
my_list.pop(3)
print (my_list)

# Both of the above remove "hello" from the list
```

```
[3, 4, 7, 2]
[3, 4, 7, 2]
```

Working with Lists : Checking existence

We can check whether an item is in a list using the *in* command.

```
my_list = [3, 4, 7, "hello", 2]
if 2 in my_list:
    print("Already there")
```

```
Already there
```

Working with Lists : Copying Lists

It may be tempting to copy a list by saying :

```
my_list = [3, 4, 7, "hello", 2]  
copy_of_my_list = my_list
```

BUT this won't work as expected. What do you think will happen?

Working with Lists : Copying Lists

The previous example will simply make another reference to the *same list*.

To create a separate copy of the list, we need to use the *copy()* function.

```
my_list = [3, 4, 7, "hello", 2]  
copy_of_my_list = my_list.copy()
```

Now we have *two lists*, and can work with them separately.

List Comprehension

Python has a really useful feature called “List Comprehension” that allows us to easily create lists based off other lists using a single line of code.

Let's consider an example. Let's imagine we have a list of numbers, and we want to set up a second list containing all the numbers of the first list, but doubled.

Here's one way we could do it :

```
list_a = [1,2,3,4,5]
list_b = []

for number in list_a:
    list_b.append(number*2)

print (list_b)
```

Or we could just use a list comprehension :

```
list_a = [1,2,3,4,5]
list_b = [x*2 for x in list_a]

print (list_b)
```



[2, 4, 6, 8, 10]

List Comprehension

We can also use conditional logic in a list comprehension, so that we form a list from another list where the elements meet certain criteria :

```
list_a = [1,2,3,4,5,6,7,8,9,10]
list_b = []

for number in list_a:
    if number % 2 == 0:
        list_b.append(number)

print (list_b)
```

```
list_a = [1,2,3,4,5,6,7,8,9,10]
list_b = [num for num in list_a if num % 2 == 0]

print (list_b)
```

[2, 4, 6, 8, 10]

Dictionaries

A dictionary is a really useful way to store values associated with an index value. We may want to store various information about a patient in a dictionary so it becomes easier to identify to what each element refers :

```
my_patient_dictionary = {"patient_ID":1674234,
                        "name":"Bob Smith",
                        "year_of_birth":1961,
                        "readmission":True
                        }
```

```
print (my_patient_dictionary)
```

```
# We can easily grab a particular piece of information
# about our patient # using [] notation
```

```
print (my_patient_dictionary["name"])
```

```
# And we can change dictionary values easily
my_patient_dictionary["year_of_birth"] = 1960
print (my_patient_dictionary)
```

```
# Or even add new key-value pairs
my_patient_dictionary["month_of_birth"] = 6
print (my_patient_dictionary)
```

```
# Or delete one we no longer need
del my_patient_dictionary["readmission"]
print (my_patient_dictionary)
```

```
{'patient_ID': 1674234, 'name': 'Bob Smith', 'year_of_birth': 1961, 'readmission': True}
```

```
Bob Smith
```

```
{'patient_ID': 1674234, 'name': 'Bob Smith', 'year_of_birth': 1960, 'readmission': True}
```

```
{'patient_ID': 1674234, 'name': 'Bob Smith', 'year_of_birth': 1960, 'readmission': True, 'month_of_birth': 6}
```

```
{'patient_ID': 1674234, 'name': 'Bob Smith', 'year_of_birth': 1960, 'month_of_birth': 6}
```

Sets and Tuples

As discussed earlier, Python also has Sets and Tuples. We won't cover them today, but you should read about them on pythonhealthcare.org :

<https://pythonhealthcare.org/2018/03/15/5-sets/>

<https://pythonhealthcare.org/2018/03/14/4-python-basics-tuples/>

Exercise 4 and Lunch

We'll now take a break until 1.30pm. Have some lunch, stretch your legs and take a well-deserved break!

At 1.30, you'll come back and spend 40 minutes working through “python_prog_workbook_3” in Jupyter Lab.

I'll keep the Zoom call open, so you don't need to disconnect – just mute and switch off your video.

Libraries and Imports

Libraries are collections of functions that have been written by other people, containing useful bits of code so we don't have to reinvent the wheel every time we write a program. You'll learn about functions in the next session.

To use libraries, we need to *import* them into our program. We usually write import statements at the start of our program.

```
import math # imports the math library
import random # imports the random library
```

The Random Library

Python's "random" library contains a number of functions that allow us to generate random numbers.

This is useful, because we need random numbers if we're going to build *stochastic models*.

Remember our talk about distributions in the very first session...?

Distributions

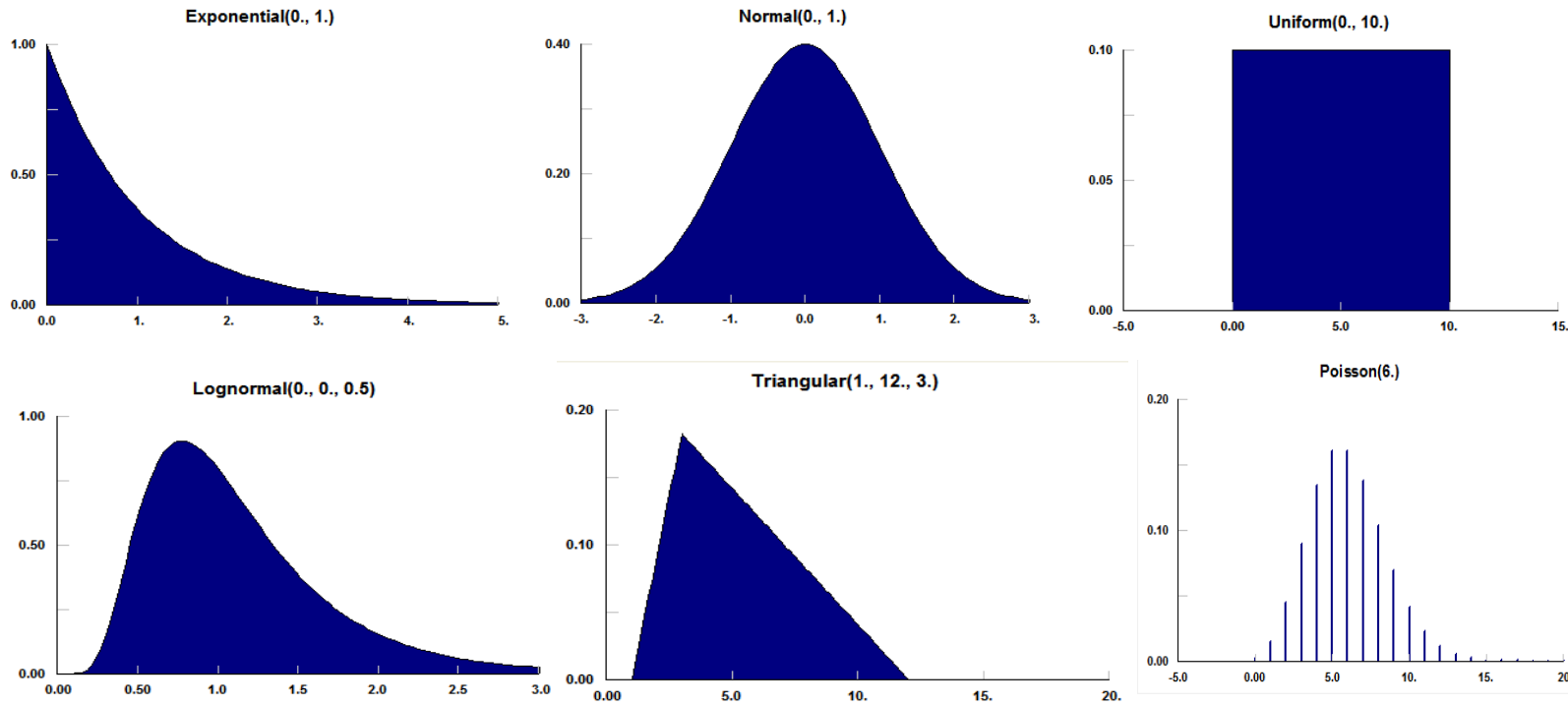
A distribution is a way of representing the variability within data. It provides us with an estimate of the probability of a value occurring in the future.

For example, a distribution might tell us that 30% of patients in the past have spent 6 minutes with the triage nurse. We can then say that, for each new patient coming in, there is a 30% chance that the time they spend with the triage nurse will be 6 minutes.

There are lots of “named distributions” available – distributions that have certain shapes and characteristics. The one you choose will depend on the shape of your real world data – you want to find one that best “fits” the shape of your data.

Each distribution is ‘defined’ by zero to many parameters. These parameters can specify the skew, range etc of the distribution.

Some Distributions



Exponential – common for inter-arrival times

Log Normal – common for process times

Poisson – describes the number of arrivals in any given period if arrival is random

Triangular – useful when data is limited

Random Library Functions

Now we've imported the Random library, let's look at some of the functions we can use. When we call a function from the library, we need to use the format :

library_name.function_name()

```
import random

# Generate a random number between 0 and 1 sampled from a uniform distribution
random.random()

# Generate a random integer (whole number) between 1 and 10 inclusive
random.randint(1, 10)

my_list = ["Dan", "Mike", "Kerry"]

# Choose a random item from the list my_list
random.choice(my_list)
```

Exercise 5 - Are you smarter than Dan as a 4 year old?

When I was 4 years old, I wrote my very first program in BASIC. In it, the computer randomly picked a whole number between 1 and 100, and the user had 10 chances to guess the number. Every time the user guessed a number, they would be told either that the number was “too low”, “too high” or “correct”. If the user used up all 10 chances without guessing correctly, they were told “you lose” and the game would end.

You now have been taught enough to write this program in Python.

You should write the game above (I'd advise using Spyder) along with the following extra features that 4-year old Dan didn't implement :

- a score, which starts at 1000 and which reduces by 100 for every unsuccessful guess, and which is displayed if the user wins
- the user's guesses are stored in a list and printed once the game is over
- the game asks if the player wants to play again after every game ends
- after each game, the player's score is checked against the current high-score (default is 0) and if the last score is higher than the recorded high score then this replaces the high score.

If you do all that with time to spare, add some bells and whistles of your choosing!

You have 45 minutes (+5 minute break), and you should work as a group.

Group Coding Competition

Today, you've learned about the following in Python :

- printing to the display
- variables and casting
- user input
- comments
- mathematical operators
- conditional logic
- for loops
- while loops
- working with lists and list comprehension
- dictionaries
- libraries and imports
- the random library

Your task now, in groups, is to design and build a computer program that uses some or all of the concepts you've learned about today. Your program can do anything you like – be as creative as possible! You will have 40 minutes to design and build your program. Each group will then have 2 minutes to present their program, and a PenCHORD judge will judge the best one. (There's unfortunately no prize, other than the accolade of winning)