SUNCORP

Python School

Lesson 2

30 March 2017

PREVIOUSLY ON PYTHON SCHOOL

WE LOOKED AT

DATA TYPES!

ints : 3

floats : 5.134

strings : "Barry"

lists : [1, 4, 0, 7]

tuples : (1, 4, 0, 7)

dicts : {"key1": 100, "key2": -8}

TRY TO RECALL THEIR...

differences

behaviour

rules

IN PARTICULAR SEQUENCE TYPES

strings : "Barry"

---

lists : [1, 4, 0, 7]

tuples : (1, 4, 0, 7)

dicts : {"key1": 100, "key2": -8}

PREVIOUSLY...

we learnt how to manipulate sequence types, using:

slicing → `[:]`

&

inbuilt methods → `.append() etc.`

WORKING WITH SEQUENCES

in a repetitive manner

possible when our sequences are small

almost impossible when sequences are large

how do we work with sequences of any size?

EPISODE 2

FLOW CONTROL

What if I want to operate on **all** of the values?

What if I only want to operate on **some** of the values?

What if the operation is repetitive?

What if the number of items is huge?

What if?

What IF?

WHAT IF?

...Zzzzz...

...ZZZZZ...

RATHER

I want a process that can do this many times

```
# PSEUDO CODE
#
# repeat // for all items in my list:
#    1. then for each item
#    2. do something with it
```

which leads us to...

our first concept of flow control

ITERATION

"the process of repeating a task many times"

ITERATION IS POWERFUL

it allow us to:

MOVE THROUGH & MODIFY

sequences

and:

BUILD

new ones

Python provides two control processes for iteration

for	while

BOTH OF WHICH

use position and size

```
    ["A", "l", "e", "j", "a", "n", "d", "r", "o"]
# ----------------------------------------------> #
    [ 0 ,  1 ,  2 ,  3 ,  4 ,  5 ,  6 ,  7 ,  8 ]

                size = 9 # items
```

attributes which all **lists**, **tuples** and **dicts** have

we will now look at how each process handles flow control

FOR

Takes a collection of items

```
my_list = [99, 22, 1, 93, 6, 3, 1, 1]

for item in my_list:

    # print item to screen
    print(item)
```

and **for** each **item** in the sequence

executes a block of code

so I can move through the list like so...

```
my_list = [99, 22, 1, 93, 6, 3, 1, 1]

for item in my_list:

    # print item to screen
    print(item)
```

and print out every item

```
99
22
1
93
6
3
1
1
```

NEW SYTNAX ALERT!

Python uses a **colon** (:) to specify **code blocks**

```
for item in my_list:

    # print item to screen
    print(item)
```

Python uses a **colon** (:) to specify **code blocks**

```
for item in my_list:

    # print item to screen
    print(item)
```

anything after a colon **MUST** be indented

this defines ownership to the process

WARNING: a common syntax error

Say I want to add 10 to a list of arbitrary numbers

```
mega_list = [1, 3, 4, 1, 4, 7, 9, 999, 343 ,-1, ...]
```

You wouldn't want to do this

```
x = 10

item0 = mega_list[0] + x
item1 = mega_list[1] + x
item2 = mega_list[2] + x
item3 = mega_list[3] + x
# ad nauseum

list_plusx = [item0, item1, item2, item3, ...]
```

repetitive!

instead, we can use a `for` loop

```
mega_list = [1, 3, 4, 1, 4, 7, 9, 999, 343 ,-1, ...]
```

which will do it for us

```
for item in mega_list:

    # add 10
    new_value = item + 10

    # print to screen
    print(new_value)
```

but how do we modify the current value?

introducing `enumerate(...)`

`enumerate(...)` belongs to a special class of data type

aptly called an **iterator**

---

iterator types are designed to work with iteration

`enumerate(...)` works as follows

for each item, also return its index

```
my_list = [4, 6, 1, 0]

for (i, item) in enumerate(my_list):

    # print to screen
    print("value at index %d = %d" % (i, item))
```

index and value is always returned as a tuple

```
"value at index 0 = 4"
"value at index 1 = 6"
"value at index 2 = 1"
"value at index 3 = 0"
```

now we can modify our list

```
mega_list = [1, 3, 4, 1, 4, 7, 9, 999, 343 ,-1, ...]
```

retrieve current value → add 10 → replace

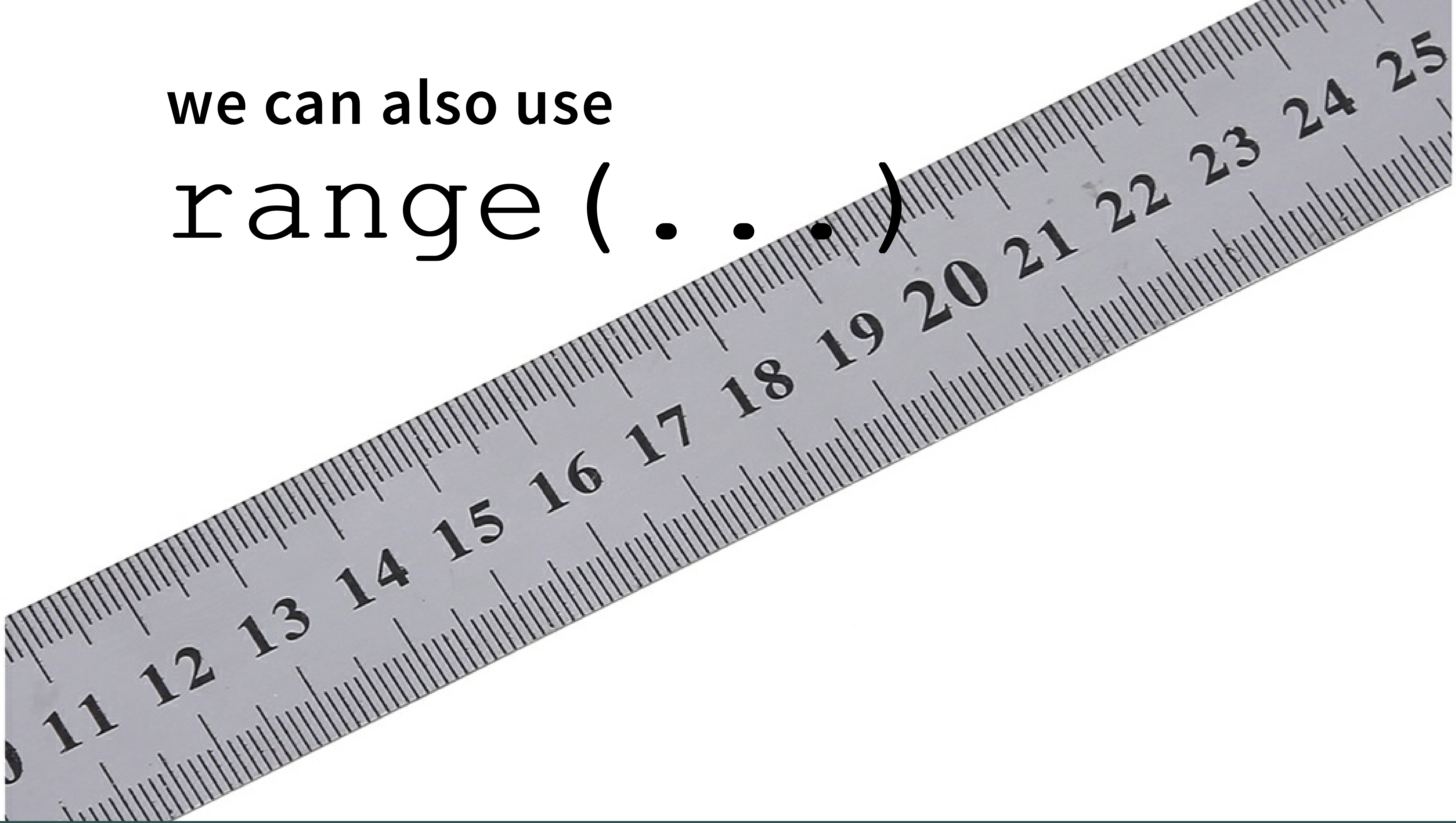
```
for (i, item) in enumerate(mega_list):

    # add 10 to item value
    new_value = item + 10

    # replace old value with new value
    mega_list[i] = new_value
```

voila

```
In : mega_list
Out: [11, 13, 14, 11, 14, 17, 19, 1009, 353 ,9, ...]
```

we can also use

`range(...)`



range(...) is also an iterator type

and aptly creates a sequence of numbers

```
for i in range(5):
    print(i)
```

```
Out: 0, 1, 2, 3, 4
```

given the upper limit (assumes `start` = 0)

```
for i in range(5, 10, 2):
    print(i)
```

```
Out: 5, 7, 9
```

or explicity give `start`, `end` and `step` to create any sequence we like

in the context of modifying our list...

```
mega_list = [1, 3, 4, 1, 4, 7, 9, 999, 343 ,-1, ...]

list_size = len(mega_list)
```

create an iterator from the list size

```
for i in range(list_size):

    # replace old value with new value
    mega_list[i] += 10
```

then modify it via index

you can also loop through multiple lists...

USING
ZIP(...)

zip(...)

joins two lists together for simultaneously looping!

```
my_list1 = [99, 22, 1, 93, 6, 3, 1, 1]
my_list2 = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

zip_lists = zip(my_list1, my_list2)
```

transforms into a list of tuples

```
In : list(zip_lists)[:3]
Out: [(99, 'a'), (22, 'b'), (1, 'c')]
```

where items at each index belong to a tuple group

so you can now operate on them together

```
for (number, letter) in zip(my_list1, my_list2):

    # do something with it
    print("%d%s" % (number, letter))
```

```
Out: '99a'
Out: '22b'
Out: '1c'
...
```

you can `zip()` multiple lists together

however, be wary of individual `list` size

will **chop** on the smallest one

finally we can dynamically

GROW LISTS

LIST CAN BE GROWN AS FOLLOWS

first, initialise an empty list

```
my_list = []

for i in range(10):

    # add a new item to the list
    my_list.append(i)
```

set a list size → add new item(s)

```
In : my_list
Out: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

tuples iterate in the same way as lists

you just can't modify them

WHAT ABOUT DICTIONARIES?

function differently due to their **key:value** referencing

key:value pairs need to be retrieved using the `.items()` method

```
my_dict = {"Jake": 35, "Finn": 16, "Gunter": 7}

for (key, value) in my_dict.items():

    # print pair
    print("value for %s = %d" % (key, value))
```

just remember that entry order != storage order

```
Out: "value for Gunter = 7"
Out: "value for Jake = 35"
Out: "value for Finn = 16"
```

BTW **`enumerate()`** is quite versatile

you can grab indices for `zip`-ped lists

```
for (i, (item1, item2)) in zip(list_1, list_2):
  # stuff happens
```

and `dicts`

```
for (i, (key, value)) in my_dict.items():
  # stuff happens
```

just note that in these cases you are dealing with a nested tuple

NESTED SEQUENCES?

Just add another **for** loop statement with an indent

EXAMPLE

looping through a two dimensional list

```
list_2d = [[0, 2, 1, 0], [1, 9, 4, 0]]

for level_1 in list_2d:

    for item in level_1:

        # do something
```

goes through each sub-list

then each `item` of that sublist

iteration allows movement through a list

HOWEVER...

we frequently want to manipulate this movement

which introduces our next concept

IF ELSE

conditional statements

`if else` statements allow us to control the flow of a process

---

using logic gates → (**AND**, **OR**)

and simple binary outcomes → (**True**, **False**)

EXAMPLE

we use conditionals to perform simple tests

```
# syntax to close a door
door_open = True

if door_open == True:
    print("closing door")
    door_open = False

else:
    pass
    print("already closed")
```

somtimes our tests may require a joint condition to pass

```
x = 15

if (x > 10) and (x < 20):
    print("within bounds")

else:
    print("outside bounds")
```

and we can chain together different logic combinations

```
if ((x > 10) and (x < 20)) and (isinstance(x, int)):

    print("is an integer within bounds")
```

we can also have many outcomes (non-binary) using the `elif` statement

```
if age < 20:
    print("Person is young")

elif (age >= 20) and (age < 30):
    print("Person is young-ish")

elif (age >= 30) and (age < 40):
    print("Still quite young")

elif (age >= 40) and (age < 60):
    print("Person is in their prime")

else:
    print("Person is of good vintage")
```

YOU CAN TEST ALMOST ANYTHING

```
if len(my_list) > 10:

if type(my_list) == list:

if value is None:

# etc
```

as long as the outcome is binary → [0, 1]

finally `if else` statements can be nested!

```
if age < 25:
    print("Check ID")

    if age > 18:
        print("Person can drink")

    else:
        print("Get lost kid")

else:
    print("Person can drink")
```

NOW LET'S LOOK AT HOW WE CAN COMBINE

`if else` with `loops`

FILTERING

imagine a list of random numbers where you're only interested in values **greater than 10**

I don't want to destroy the list though

HOW ????????

EASY!

just throw an **`if`** statement into your list

```
my_list = [5, 3, 1, 77, 31, 24, 2, 4, -1, 100]

for item in my_list:

    if item > 10:
        # do something with it
        print(item)
```

`else` is only necessary if there is an alternate state

EASY!

just throw an **`if`** statement into your list

```
my_list = [5, 3, 1, 77, 31, 24, 2, 4, -1, 100]

for item in my_list:

    if item > 10:
    # do something with it
    print(item)
```

`else` is only necessary if there is an alternate state

SIMPLE
CATEGORISATION

What if we want to do something a bit more complicated?

like compare the current value with the previous one?

our good friend **`enumerate()`** returns to help!

```
for (i, item) in enumerate(my_list):

    # first element can't be compared
    if i > 0:

        if my_list[i-1] < item:
            print("bigger")

        else:
            print("smaller")
```

```
Out: "smaller"
Out: "smaller"
Out: "bigger"
Out: "smaller"
Out: "smaller"
Out: "smaller"
Out: "smaller"
```

`if else` allows precise control in our program using simple logic

---

you will almost always use it in your programs & scripts

WHILE

runs **indefinitely** as long as some condition remains true

```
i = 0
limit = 5

while i < limit:

    print("Hello, world!")

    i = i + 1
```

output:

```
"Hello, world!"
"Hello, world!"
"Hello, world!"
"Hello, world!"
"Hello, world!"
```

Now lets use the `while` statement with

SEQUENCES

let's write some code to print all the items in a `list`

```
# feel free to define your own
my_list = [2, 5, 1, 5, 4, 4, 4]

# get the length of your list
list_length = len(my_list)
# define and set a counter
i = 0

while i < list_length:
    # store the current item
    item = my_list[i]
    # print this item to the screen
    print(item)
    # increment counter by one
    i = i + 1
```

let's write some code to print all the items in a `list`

```
# feel free to define your own
my_list = [2, 5, 1, 5, 4, 4, 4]

# get the length of your list
list_length = len(my_list)
# define and set a counter
i = 0

while i < list_length:
    # store the current item
    item = my_list[i]
    # print this item to the screen
    print(item)
    # increment the count by one
    i = i + 1
```

EXERCISE

PRINT LIST ITEMS IN REVERSE ORDER

you can also use `while` loops like `for` loops

```
mega_list = [1, 3, 4, 1, 4, 7, 9, 999, 343 ,-1, ...]
```

again, we modify our list to be +10

```
while i < len(mega_list):

    # add 10
    item_add10 = mega_list[i] + 10

    # change to new value
    mega_list[i] = item_add10

    # increment counter
    i += 1
```

PITFALLS TO BE WARY OF

the edge condition is correct

the counter iterates

here's a more complicated exampled

with the cookie monster

SUMMARY

when to use `for` and when to use `while`

a good rule of thumb

use `for` loops for anything data-driven

•

reading in datasets

analysing datasets

anything finite in sequence

a good rule of thumb

use `while` loops for anything process-based

•

simulations that have a state

working with data-streams (real-time data)

anything infinite in sequence

we ♥ for

we ♥ while

we ♥ if else statements

we ♥ Python

NEXT TIME

$$y = f(x)$$