Rhys Whitley • Natural Perils Pricing • March 2017

THE OBJECTIVE OF THESE CLASSES

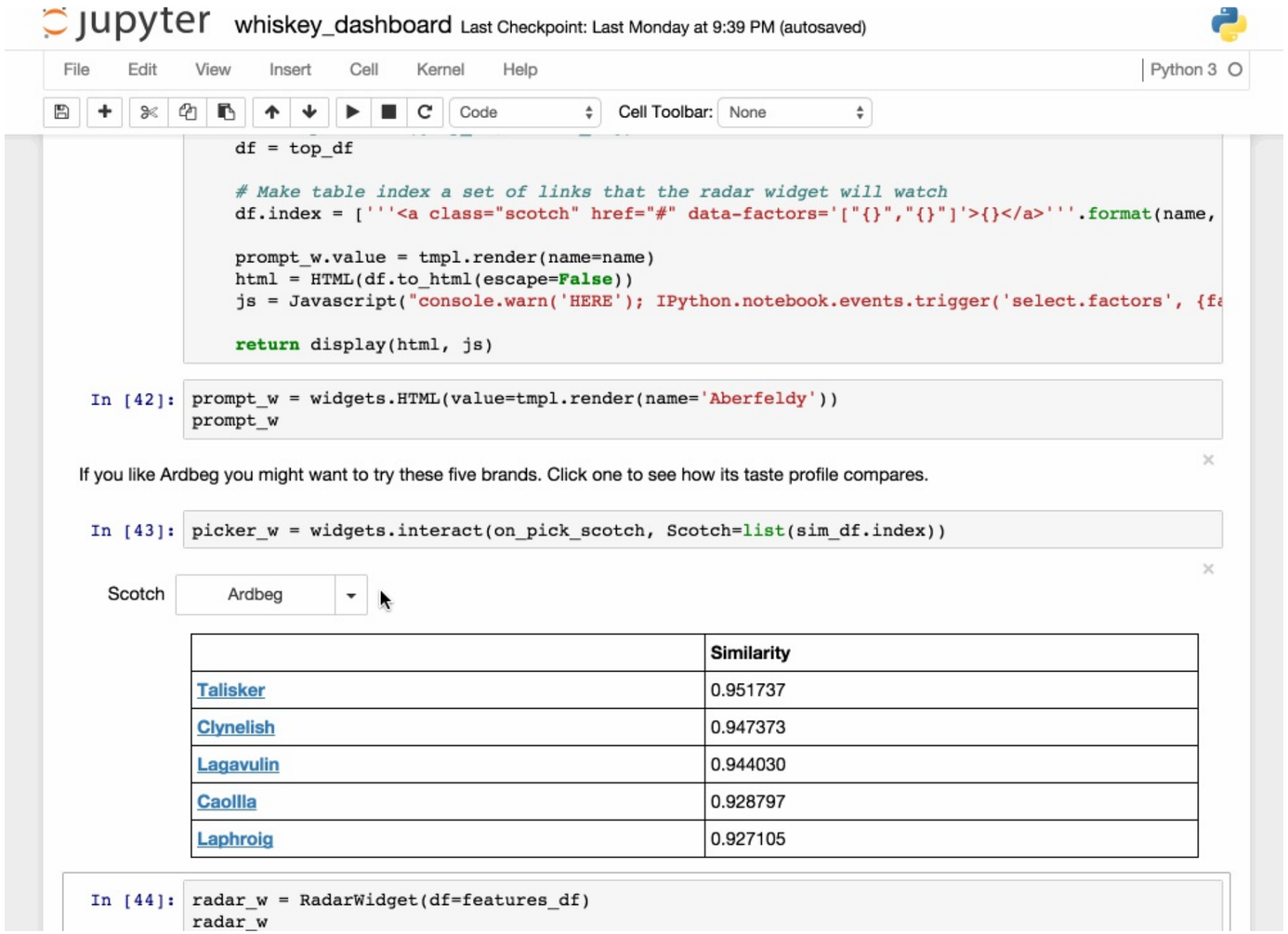
is to introduce you to the Python language and all that it entails

Manipulate and move around the language; create programs and scripts

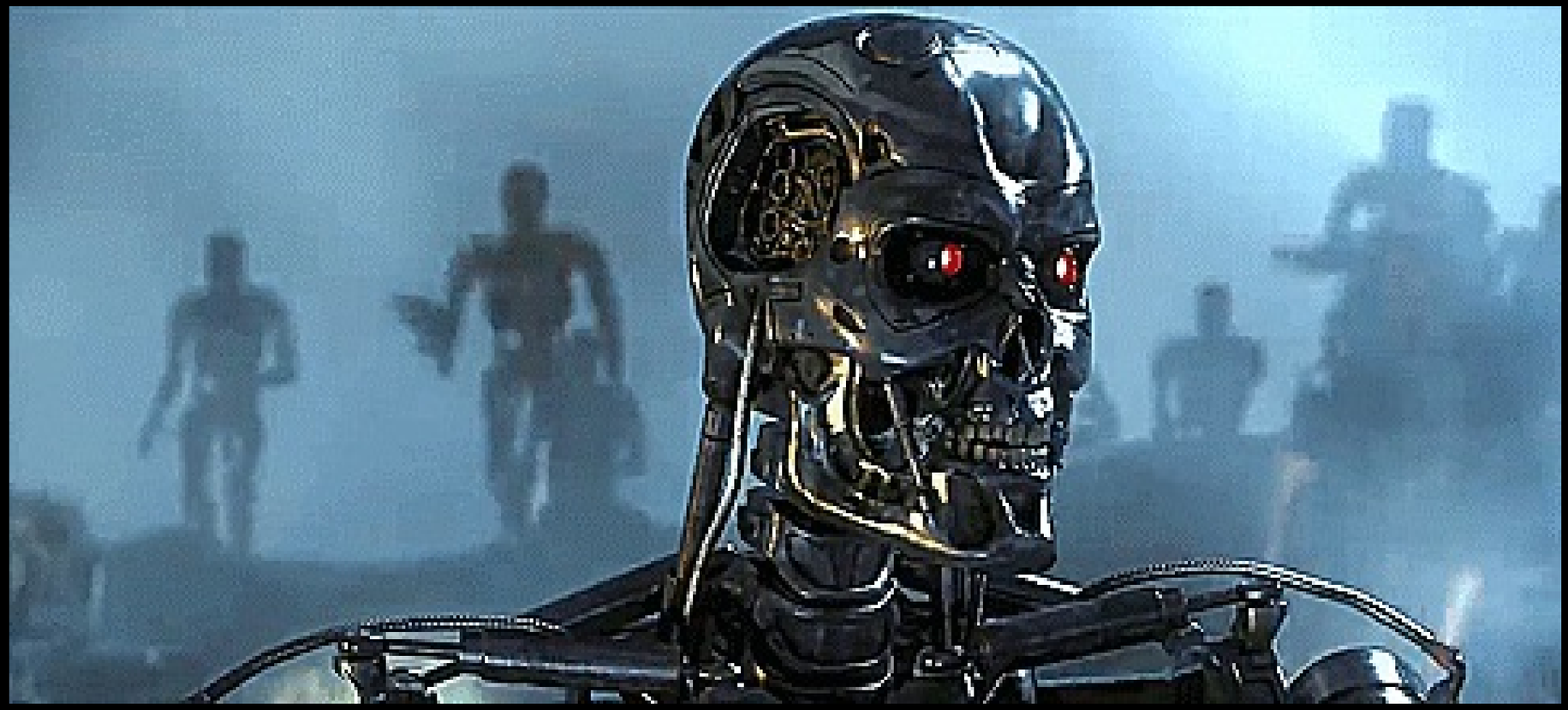
README.md | .travis.yml | python-tool... | python-tool... | tools.py | Settings

```
49              sys.path.insert(0, path)
50
51          script = jedi.api.Script(
52              source=request['source'],
53              line=request['line'] + 1,
54              column=request['col'],
55              path=request.get('path', ''),
56          )
57
58          if request['type'] == 'usages':
59              self._write_response(self._serialize('usages', script.usages()))
60          elif request['type'] == 'gotoDef':
61              self._write_response(self._serialize('gotoDef', script.goto_definitions()))
62
63      def _write_response(self, response):
64          sys.stdout.write(response + '\n')
65          sys.stdout.flush()
66
67      def watch(self):
68          while True:
69              try:
70                  data = self._input.readline()
```

Using notebooks to prototype code, test analysis, produce reports



Automate your tasks // use the power of the machines



OVERALL

Convince you of its utility in business - analytics, data viz, work-flow management

Instil good coding and curation practices

Empower you with with the knowledge to migrate from archaic tools like MS Office and SAS

Give you the confidence to go out and learn on your own, so you can even teach me!

WHAT I PLAN TO COVER

Basic Concepts

Functions & Classes

Data Analytics

Data Viz

Geospatial Analysis

Machine Learning

Scripts, Jobs & Optimisation

Team Hack'athon

LESSONS ARE INTERACTIVE

Play with the code as I talk

Feel free to ask questions as we go

Help the person next to you

I may throw around a lot of technical jargon, so just stop and ask me what the hell I'm talking about.

MOST IMPORTANTLY...

REMEMBER TO HAVE FUN!

Python (and coding in general) is here to make your life easier!

WHY PYTHON?

YOU WANT TO WRITE POWERFUL AND EFFECTIVE CODE

without becoming a software engineer

created by Guido von Rossum (BDFL)

"intended to make programming accessible to intelligent people"

A HIGH-LEVEL LANGUAGE

? o_O ?

---

what you write is closer to human language

significant areas of the computer system are automated for you

code is interpretted

QUICK COMPARISON

Fortran (low-level)

```
! need to define the program
program example

    implicit none
    ! need to do this
    integer(kind=1) :: i
    real(kind=8) :: x
    character(len=4) :: word
    ! before you can do this
    i = 0
    x = 14.23434324591
    word = "noun"

    write(*,*) word
end program example
```

Python (high-level)

```
# done for you
i = 0
x = 14.23434324591
word = "noun"

print(word)
```

QUICK COMPARISON

Fortran (compiled)

```
# compile program ...
$ gfortran -o example example.f

# to exectute
$ ./example
$ noun
```

Python (interpreted)

```
# just execute
$ python example.py
$ noun
```

DESIGN PHILOSOPHY

free-form programming style that allows you to explore and make mistakes

quick programming - you have an idea, so try it out

supportive community - newbies always welcome

all encapsulated in the Zen of Python

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
```

AND IN CASE YOU WERE WONDERING ABOUT THE NAME

IT COMES FROM MONTY PYTHON

THE PYTHON INTERPRETER

the interpreter allows communication between you and the machine

reads and executes commands interactively (input/output)

two interpreters currently exist for Python

2.7.13	**3.6.0**
end of life: 2020	now standard

there are some slight differences behaviour

were going to be using **3.6.0**

all nicely packaged together in

if you haven't already, download from

https://www.continuum.io/downloads

and install it on your H: driver

SETUP WITH JUPYTER NOTEBOOK

type this into your notebook

```
print("Hello world!")
```

Congratulations, you've executed your first piece of Python code

so lets start with most important thing you'll ever learn

EPISODE 1

THE BASICS

DATA TYPES

and variables

Rhys Whitley • Natural Perils Pricing • March 2017

WHAT IS A VARIABLE?

is any type of named reference, which points to a stored value on the computer

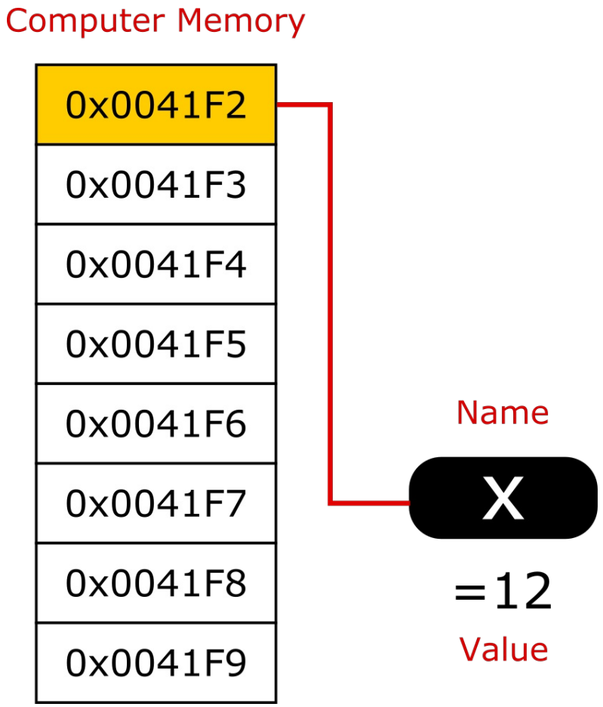
```
x = 12
```

is any type of named reference, that points to some arbitrarily stored value on the computer

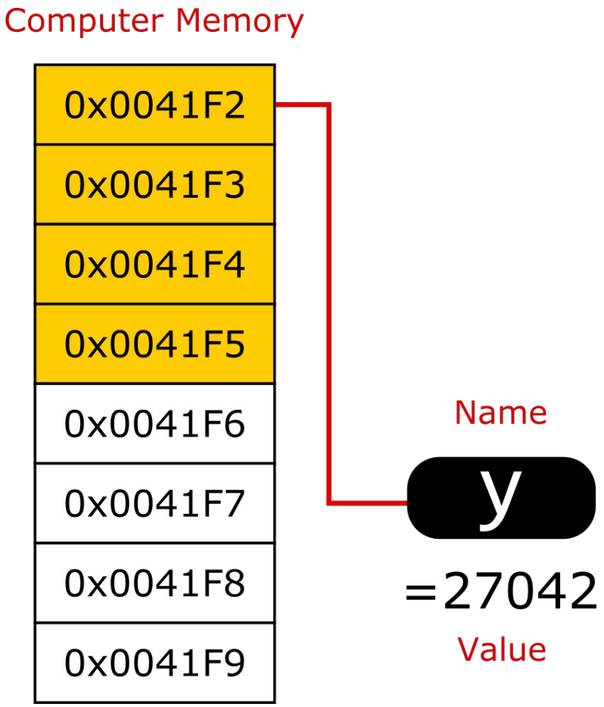
```
x = 12
```

BEHIND THE SCENES

```
x = 12
```




```
y = 27042110
```



CONVENTIONS ON NAMING

keep variables short-ish and relevant to what they store

```
age = 30 # good
my_mothers_uncles_sisters_boyfriends_age = 30 # bad
```

don't use names that also refer to python functions

```
print = 300 # no
sum = 230 # no
def = 'Sally' # no
```

must always start with a character

```
amount = 300 # will work
300amount = 300 # error
```

LET'S LOOK AT SOME OF PYTHON'S DATA TYPES

and how they behave

INTEGER

Is latin for *whole number*; cannot represent a fractional number

```
a = 5
b = -1000000
```

```
In : a // 2
Out: 2
```

Good for counting, ranking and categorising

FLOATING POINT

Approximates a real number to a fixed number of significant digits

```
c = 4.5234351111
d = 131093923.10
```

refers to the moving decimal

```
In : c * 10.0
Out: 45.234351111
```

Good for continuous values, measurements, parameters

BOOLEAN

Represents a binary outcome: true or false - actually a subclass of integer

```
e = True  # 1
f = False # 0
```

```
In : e == f
Out: False
```

Good for dummy variables/flags and conditions

STRINGS

Represents a single character or grouping of letters, defined by surround quotation marks

```
letter = "x"
word = "old"
name = "Fred"
number = "100"
```

...have a specific behaviour, but we're going to come back to this later on though

THERE ARE ALSO OTHER DATA TYPES THAT YOU WILL COME TO FIND

Buffer

File

Module

*None

SEQUENCES

or containers that hold many items

Rhys Whitley • Natural Perils Pricing • March 2017

PYTHON HAS A NUMBER OF CONTAINER TYPES

Lists

Tuples

Dicts

LISTS

Always denoted by **square** [...] brackets

```
number_list = [1, 2, 6, 1, 2]
```

and are generally used to hold homogeneous data types

```
name_list = ["Alex", "Lena", "Eva", "Jason", "Nancy"]
```

can also store mixed-types, but I would generally avoid

```
mixe    d_list = [5.4, "H", 4, "rope", "k", False]
```

CREATING LISTS

put stuff between [...] and separate with a comma

```
some_list = [1, 2, 0, 1193, -1]
```

thankfully you rarely need to do this

repeating something?

```
In : [6]*6
Out: [6, 6, 6, 6, 6, 6]
```

TOO EASY!

Need a sequence of numbers?

meet our good friend `range()`

```
In : list(range(6))
Out: [0, 1, 2, 3, 4, 5]
```

make sure you wrap this in a `list()` call

---

we ♥ `range()`

but we'll cover that in the next lesson

The parts of a list are referred to as items (or elements)

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

---

the term **index** is used to denote position and *in Python*
always starts at 0

and we can use the index to access parts of the list

```
name = ["A", "l", "e", "j", "a", "n", "d", "r", "o"]
```

specific items

```
In : name[0]
Out: "A"

In : name[5]
Out: "n"
```

or chunks

```
In : name[1:7]
Out: ["l", "e", "j", "a", "n", "d"]

In : name[4:]
Out: ["a", "n", "d", "r", "o"]
```

THIS IS REFERRED TO AS

SLICING

not only can we take parts of a list, but we can also change them

```
In : print(name)
Out: ["A", "l", "e", "j", "a", "n", "d", "r", "o"]
```

```
name[3] = "x"
name[7] = "e"
name[8] = "r"
```

```
In : print(name)
Out: ["A", "l", "e", "x", "a", "n", "d", "e", "r"]
```

Python is also quite funky in that it allows indexing from left to right

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

and from right to left!

```
     ["A", "l", "e", "j", "a", "n", "d", "r", "o"]
# ---------------------------------------------------> #
     [-9 , -8 , -7 , -6 , -5 , -4 , -3 , -2 , -1 ]
```

let's take the last letter

```
In : name[-1]
Out: "o"
```

and the last 4 letters

```
In : name[-4:]
Out: ["a", "n", "d", "r", "o"]
```

Position presuposes dimensioniality, and so lists also have a size attribute

```
In  : len(name)
Out: 9
```

You will see later on that both position and size are powerful reference points in automating processes

MANIPULATING LISTS

lists have a number builtin methods that allow you to easily change their **size**, **contents**, and **order**

```
dir(list)
```

```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__iadd__',
```

ITEMS CAN BE ADDED...

by attaching them to the end of list

```
my_list = [0, 2, 1, 0, 1, 9, 4, 0]
extra_item = 99
```

```
In : my_list.append(extra_item)
Out: [0, 2, 1, 0, 1, 9, 4, 0, 99]
```

or by inserting them somewhere specific

```
extra_item = 99
```

```
In : my_list.insert(4, extra_list)
Out: [0, 2, 1, 0, 99, 1, 9, 4, 0]
```

CAUTION

Python considers whatever is between the brackets (...) as **one** item and will add it in like so

```
my_list = [0, 2, 1, 0, 1, 9, 4, 0]
extra_list = [-1, -8]
```

```
In : my_list.append(extra_list)
Out: [0, 2, 1, 0, 1, 9, 4, 0, [-1, -8]]
```

and

```
In : my_list.insert(4, extra_list)
Out: [0, 2, 1, 0, [-1, -8], 1, 9, 4, 0]
```

if you actually wanted to expand a list somewhere in the middle...

```
my_list = [0, 2, 1, 0, 1, 9, 4, 0]
extra_list = [-1, -8]
```

you need to insert at the sliced point of interest

```
my_list[4:4] = extra_list
```

```
In : print(my_list)
Out: [0, 2, 1, 0, -1, -8, 1, 9, 4, 0]
```

...don't know why you would though...

WE CAN ALSO RETRIEVE ITEMS

by considering lists as a stack; i.e. last in / first out

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

```
In : my_list.pop()
Out: 0
```

which permanantly removes the last item from the list

```
In : print(my_list)
Out: [0, 2, 1, 0, 1, 9, 4]
```

and you can keep `pop()`-ping until the list is empty

```
In : my_list.pop()
Out: 4
In : my_list.pop()
Out: 9
In : my_list.pop()
Out: 1
In : my_list.pop()
Out: 0
In : my_list.pop()
Out: 1
In : my_list.pop()
Out: 2
In : my_list.pop()
Out: 0
```

DON'T LIKE SOMETHING IN YOUR LIST

I really hate that 9

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

no problem, you can delete them

by value

```
In : my_list.remove(9)
Out: [0, 2, 1, 0, 1, 4, 0]
```

or index

```
In : del my_list[-3]
Out: [0, 2, 1, 0, 1, 4, 0]
```

WE CAN ALSO SORT LISTS

in numerical

```
num_list = [0, 2, 1, 0, 1, 4, 0]
```

```
In : num_list.sort()
Out: [0, 0, 0, 1, 1, 2, 4]
```

and alphabetical order*

```
name_list = ["S", "t", "e", "v", "e"]
```

```
In : my_list.sort()
Out: ["S", "e", "e", "t", "v"]
```

perhaps you prefer everything backwards...

then just pass a reverse argument to the sort method!

```
In : num_list.sort(reverse=True)
Out: [4, 2, 1, 1, 0, 0, 0]
```

you'll notice that all of these methods, permanantly change the list itself

---

unless you have good reason, then it's best to avoid

---

later I will show how *filtering* can achieve the same thing without destroying your list

THAT'S NOT ALL THOUGH

there are other cool things we can do with lists

WE CAN ALSO JOIN LISTS TOGETHER...

using concatenation...

```
my_list = [0, 2, 1, 0, 1, 9, 4, 0]
other_list = [-1, -8, 1, 0, 20, 55]
```

to make bigger lists!

```
In : my_list.extend(other_list)
Out: [0, 2, 1, 0, 1, 9, 4, 0, -1, -8, 1, 0, 20, 55]
```

neat bit of syntatic sugar

```
In : my_list + other_list
Out: [0, 2, 1, 0, 1, 9, 4, 0, -1, -8, 1, 0, 20, 55]
```

LISTS CAN CONTAIN LISTS!

This is referred to as NESTING, and allows us to create a greater number of dimensions

```
# 1 dimension (or flat)
list_1d = [0, 2, 1, 0, 1, 9, 4, 0]

# 2 dimensions
list_2d = [[0, 2, 1, 0], [1, 9, 4, 0]]

# 3 dimensions
list_3d = [[[0, 2], [1, 0]], [[1, 9], [4, 0]]]
```

Items in nested lists are accessed similarly to flat lists

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

```
In : list_2d[0]
Out: [0, 2, 1, 0]
```

just use extra brackets ([...]) to delve within

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

```
In : list_2d[0][1:3]
Out: [2, 1]
```

for your own sanity, try to keep your lists down to the smallest number of dimensions possible

TUPLES

Denoted by **circle (...)** brackets

```
num_tuple = (0, 3, "Bye", False)
```

and tend to store **heterogeneous** sequences of values

They share *almost* the same funtionality with lists

They can be sliced in the same way

```
my_tuple = (1, 4, 8, 1, 1)
```

```
In : my_tuple[3]
Out: 1
```

```
In : my_tuple[2:]
Out: (8, 1, 1)
```

and expanded

```
In :my_tuple + my_tuple[2:]
Out: (1, 4, 8, 1, 1, 8, 1, 1)
```

as well as being similarly nested,

```
# 2 dimesions
tuple_2d = ((0, 2, 1, 0), (1, 9, 4, 0))

# 3 dimesions
tuple_3d = (((0, 2), (1, 0)), ((1, 9), (4, 0)))
```

where access is the same

```
list_2d = ((0, 2, 1, 0), (1, 9, 4, 0))
```

```
In : list_2d[0][1:3]
Out: (2, 1)
```

HOWEVER

THEIR BEHAVIOUR IS DIFFERENT

Have the unique attribute of **packing**, whereby comma separated values are stored into a variable

```
pack_tuple = 1, 3, True, "hello"
```

```
In : pack_tuple
Out: (1, 3, True, "hello")
```

and can be retrieved by **unpacking**

```
a, b, c, d = pack_tuple
```

```
In : a, b, c, d
Out: 1
Out: 3
Out: True
Out: "hello"
```

tuples of single values **require** a trailing **comma**

```
single_tup = (1, )
```

otherwise they are considered as the value between the (...)

```
bad_tup = (1)
```

```
In : bad_tup == 1
Out: True
```

...rather...

```
In : single_tup == 1
Out: False
```

most importantly...

TUPLES ARE IMMUTABLE

this means they cannot be changed

for example: this is not allowed

```
my_tuple = (1, 4, 8, 1, 1)
```

```
In : my_tuple[1] = -9999
TypeError: 'tuple' object does not support item assignment
```

nor this

```
In : del my_tuple[1]
TypeError: 'tuple' object does not support item deletion
```

so the tuple and its values are actually protected

WHAT IS THE POINT OF IMMUTABILITY?

Can be *hashed*; i.e. fixed in memory, which is more efficent/optimised

Protects the programmer from themselves; can't accidently change values

---

We will come back to this when we look at functions in the next lesson

ONE MORE THING

while this might be a list, ...

```
name = ["A", "l", "e", "j", "a", "n", "d", "r", "o"]
```

does it also remind you of something else?

STRINGS ARE A SEQUENCE OF CHARACTERS

AND BEHAVE VERY MUCH LIKE LISTS

so you can perform similar operations on them

```
name = "Alejandro"
```

```
In : name[1:3]
Out: "le"
```

...or...

```
In : name + "300"
Out: "Alejandro300"
```

HOWEVER, THEY ARE THEIR OWN OBJECT

with their own methods

```
In : name.upper()
Out: "ALEJANDRO"
```

and behaviour; e.g immutable

```
In : del name[-1]
Out: TypeError: 'str' object doesn't support item deletion
```

DICTS

dictionaries

are a powerful way of storing and retrieving record data, based on the concept of

{KEY : VALUE}

pairing

key-values pairs always separated by a **colon**, between **curly braces {...}**

```
my_dict = {"key1": value1, "key2": value2, "key3": value3, ..
```

KEYS CAN REFERENCE PRACTICALLY ANYTHING

could be single value entries

```
my_rec1 = {"Jake": 35, "Finn": 16, "Gunter": 7}
```

or lists, tuples, or more complex data types

```
my_rec2 = {"name": ["Jake", "Finn", "Gunter"], \
           "ages": [35, 16, 7]}
```

To access values in the dictionary, you can reference them by their associated key

```
In : my_rec1["Gunter"]
Out: 7
```

```
In : my_rec2["name"]
Out: ["Jake", "Finn", "Gunter"]
```

then any subsequent slicing if the value is a sequence

```
In : my_rec2["name"][-1]
Out: "Gunter"
```

Adding new key:values to dictionaries is done like so

```
my_rec1["Marcy"] = 21
```

```
In : my_rec1
Out: {"Marcy": 21, "Gunter": 7, "Jake": 35, "Finn": 16}
```

and entries similarly deleted as with lists

```
In : del my_rec1["Marcy"]
Out: {"Gunter": 7, "Jake": 35, "Finn": 16}
```

`dict` types also have some nice methods to grab keys and values separately

```
In : my_rec1.keys()
Out: ["Jake", "Finn", "Gunter"]

In : my_rec1.values()
Out: [35, 16, 7]
```

keys **should be unique** as you want them to give specific reference.

```
bad_dict = {"key1": 14239, "key1": 599, "key2": 3430}
```

otherwise, replicate keys will just refer to the last occurence

```
In : bad_dict["key1"]
Out: 599
```

Finally, tuple **pairs**...

```
tup2 = ("Jake", 35)
```

are seen as key-values **pairs**

```
tup_list = [("Jake", 35), ("Finn", 16), ("Gunter", 7)]
```

such that a list of tuples can be converted to a dict object

```
In : dict(tup_list)
Out: {"Gunter": 7, "Jake": 35, "Finn": 16}
```

Notice how the order of the dict is different to how it's stored in the list

THIS IS IMPORTANT TO REMEMBER!

A dictionary **is not ordered** in the obvious sense.

The order of key:values entered != the order of key:values stored

OTHER SEQUENCES

PYTHON HAS OTHER SEQUENCE OBJECTS THAT ARE COMMONLY USED

`numpy arrays`

`sets`

---

we will cover these in future lessons

SUMMARY

we ♥integers & floats

we ♥strings

we ♥lists & tuples

we ♥dictionaries

---

we ♥ Python

REMEMBER

CODING IS HARD!

..BUT EASY AT THE SAME TIME

FINALLY

practice

practice

practice

NEXT TIME