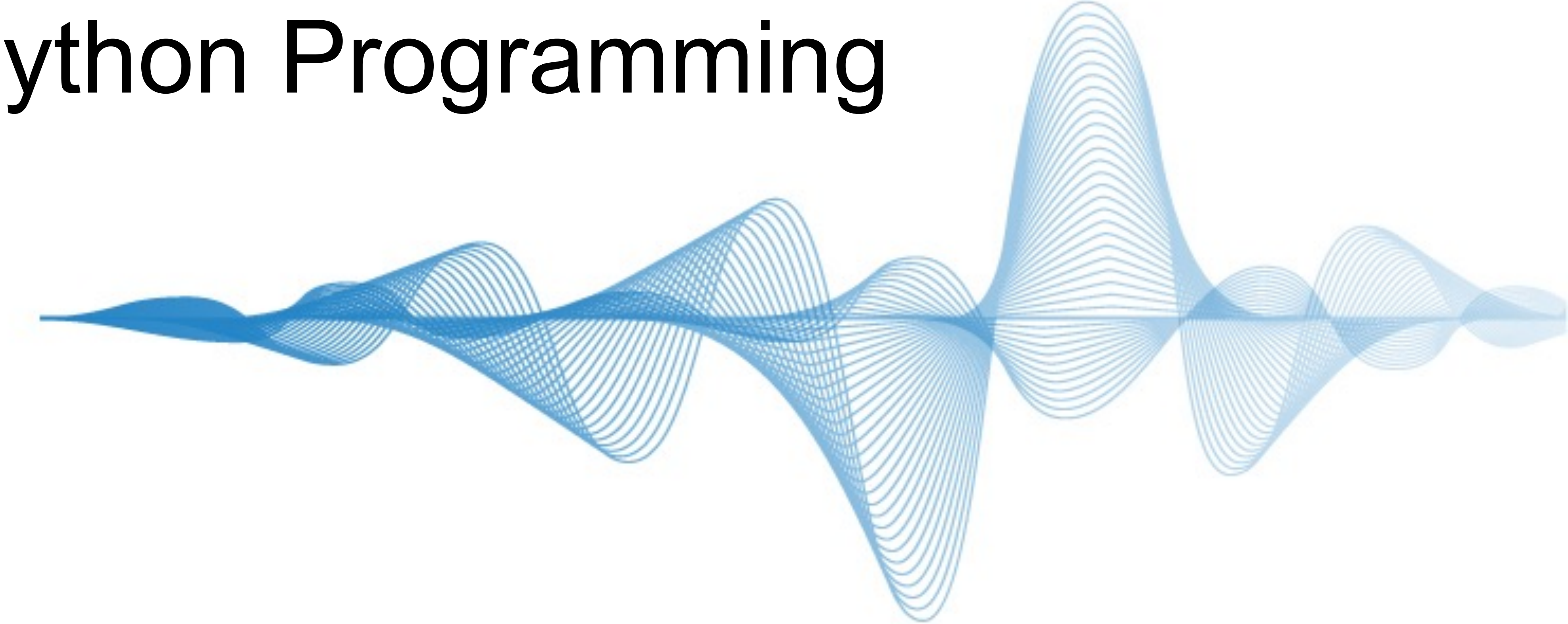
LMH
Lady Margaret Hall

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Python Programming



Getting Started: Python Programming

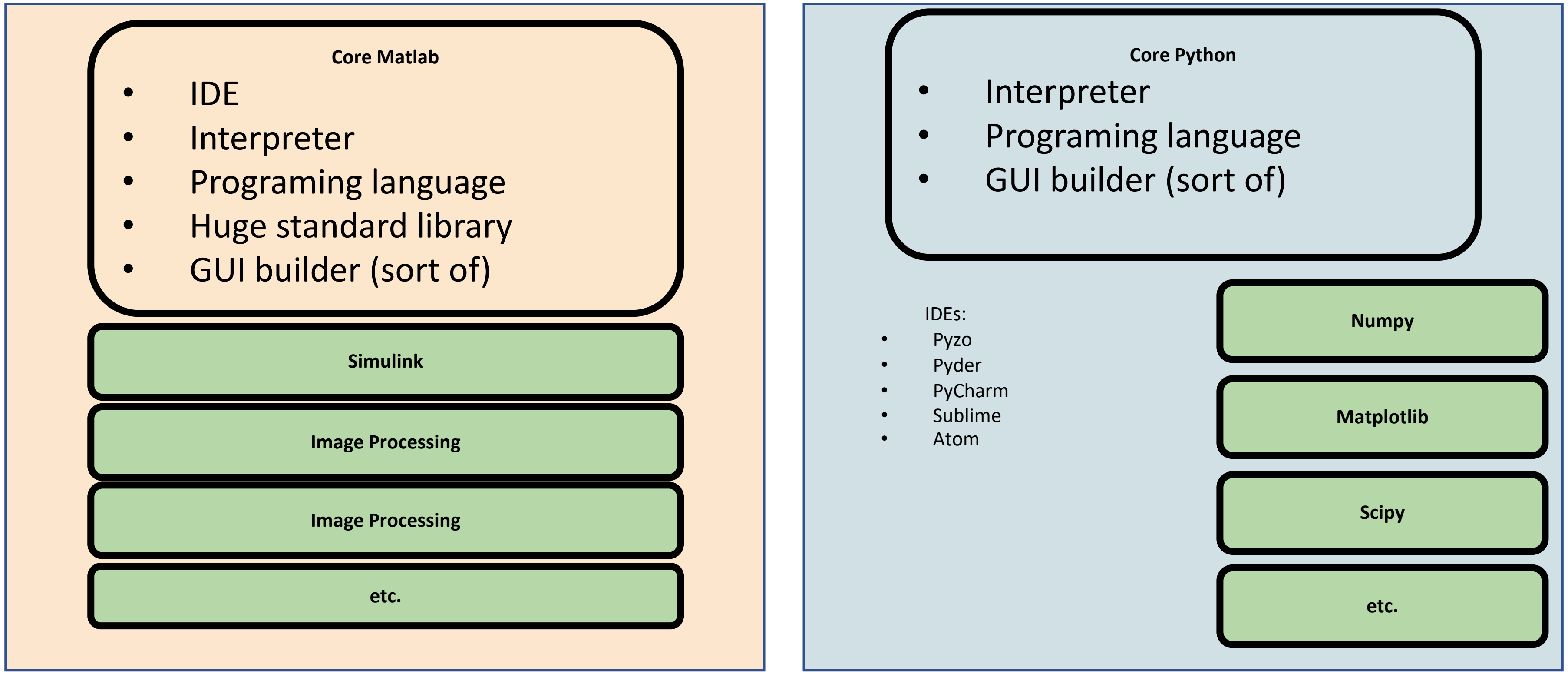
We have always loved, we still do, but it is time to move on

Time for more flexibility and scale

Which Programing Language?

MATLAB®

python™



Core Matlab

- IDE
- Interpreter
- Programing language
- Huge standard library
- GUI builder (sort of)

Simulink

Image Processing

Image Processing

etc.

Core Python

- Interpreter
- Programing language
- GUI builder (sort of)

IDEs:

- Pyzo
- Pyder
- PyCharm
- Sublime
- Atom

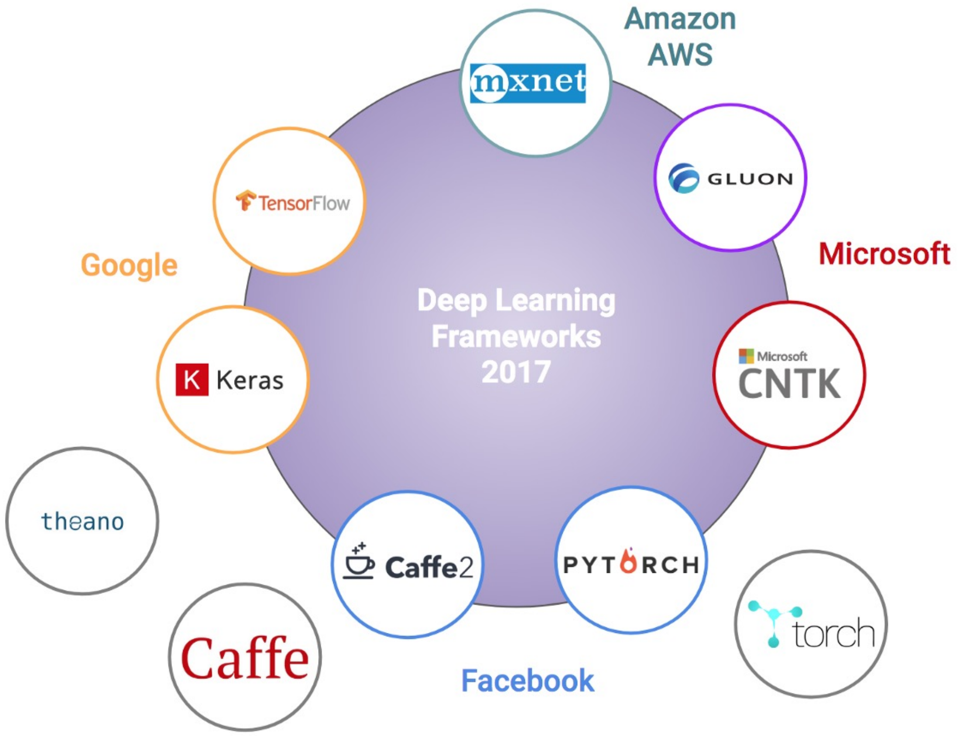
Numpy

Matplotlib

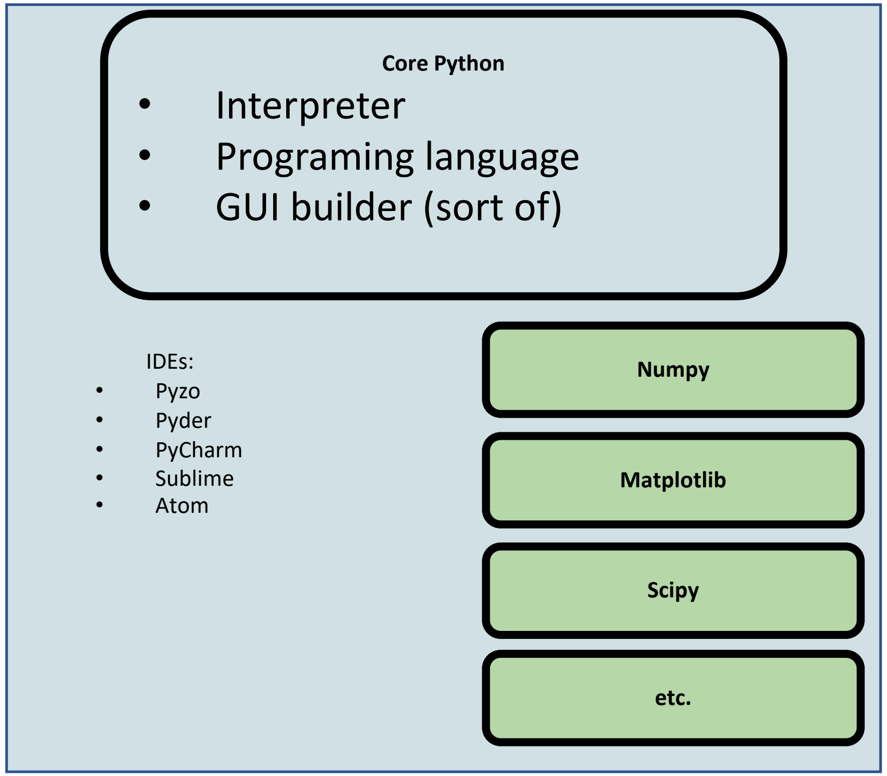
Scipy

etc.

More Importantly – Deep Learning Pack



python™


Core Python

- Interpreter
- Programing language
- GUI builder (sort of)

IDEs:

- Pyzo
- Pyder
- PyCharm
- Sublime
- Atom

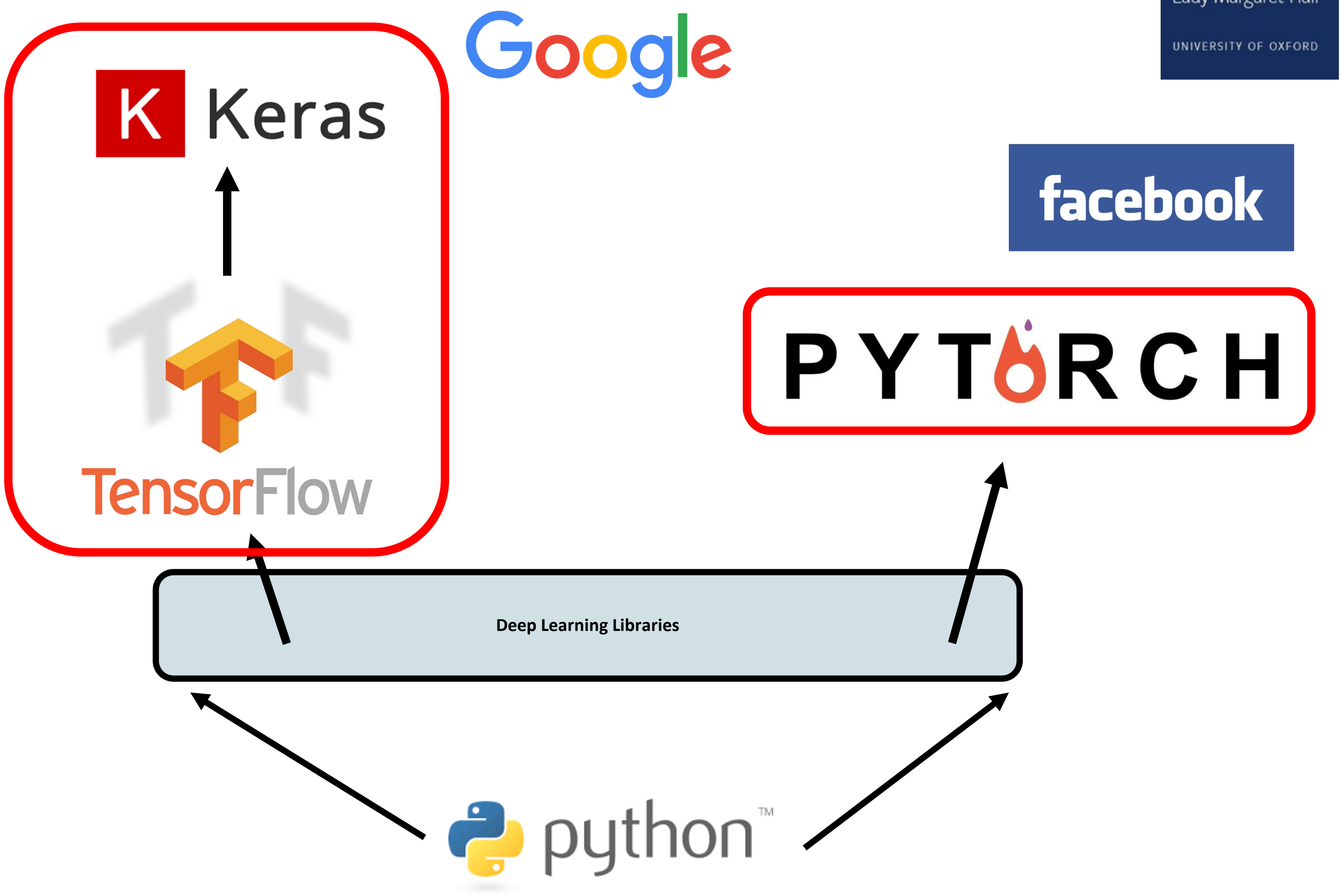
Numpy

Matplotlib

Scipy

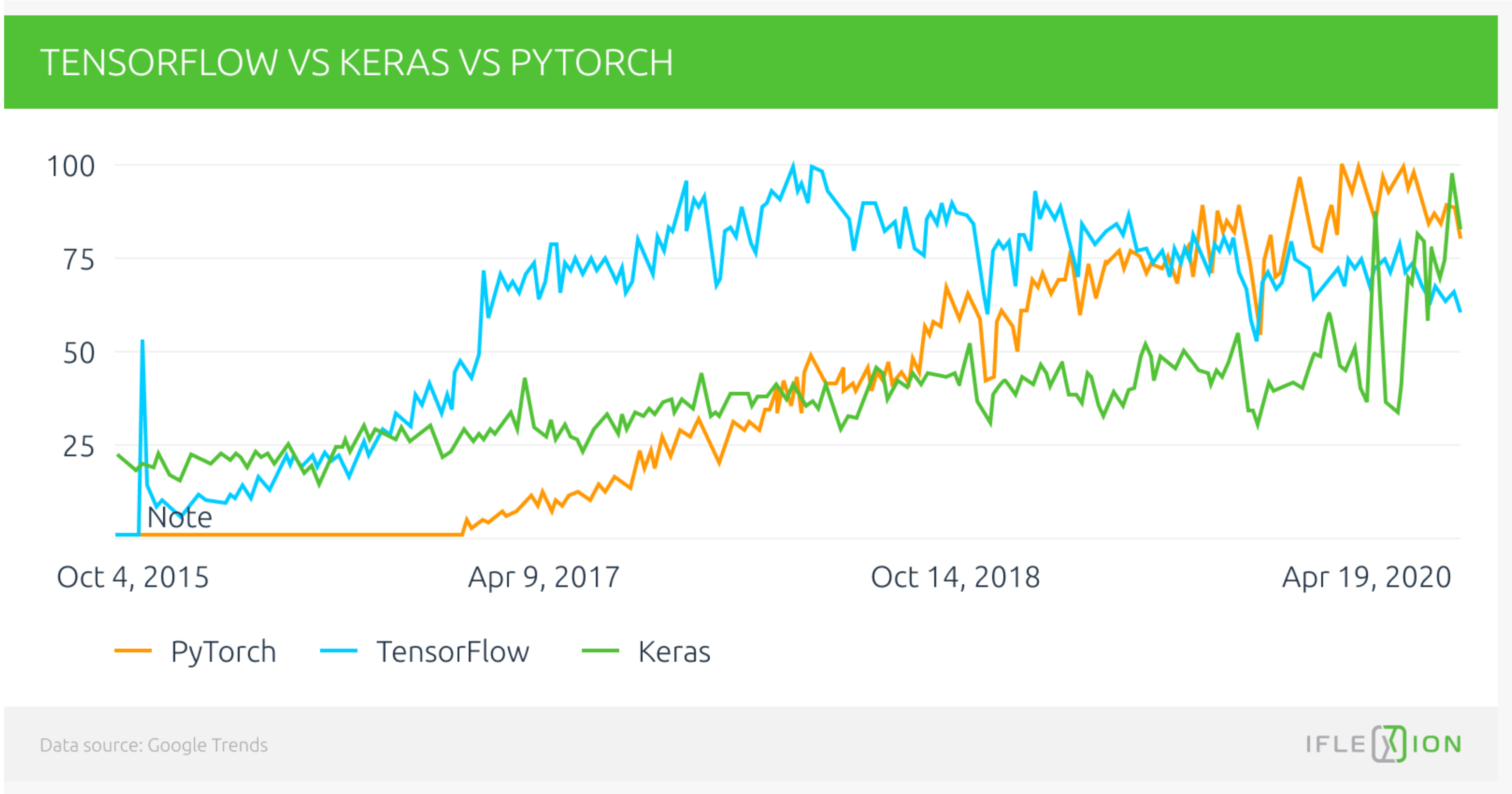
etc.

More Importantly – Deep Learning Pack



Deep Learning Libraries

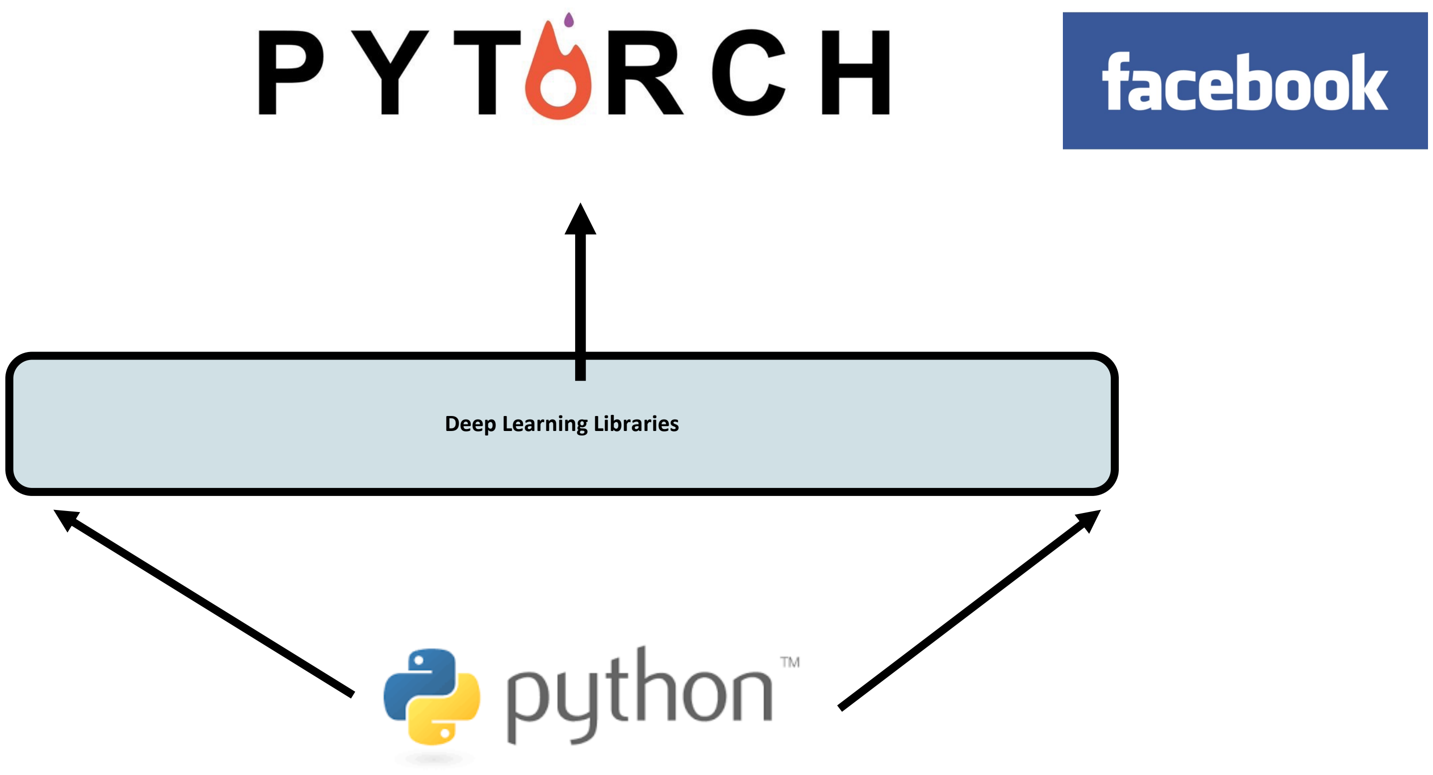
More Importantly – Deep Learning Pack


TENSORFLOW VS KERAS VS PYTORCH

Data source: Google Trends

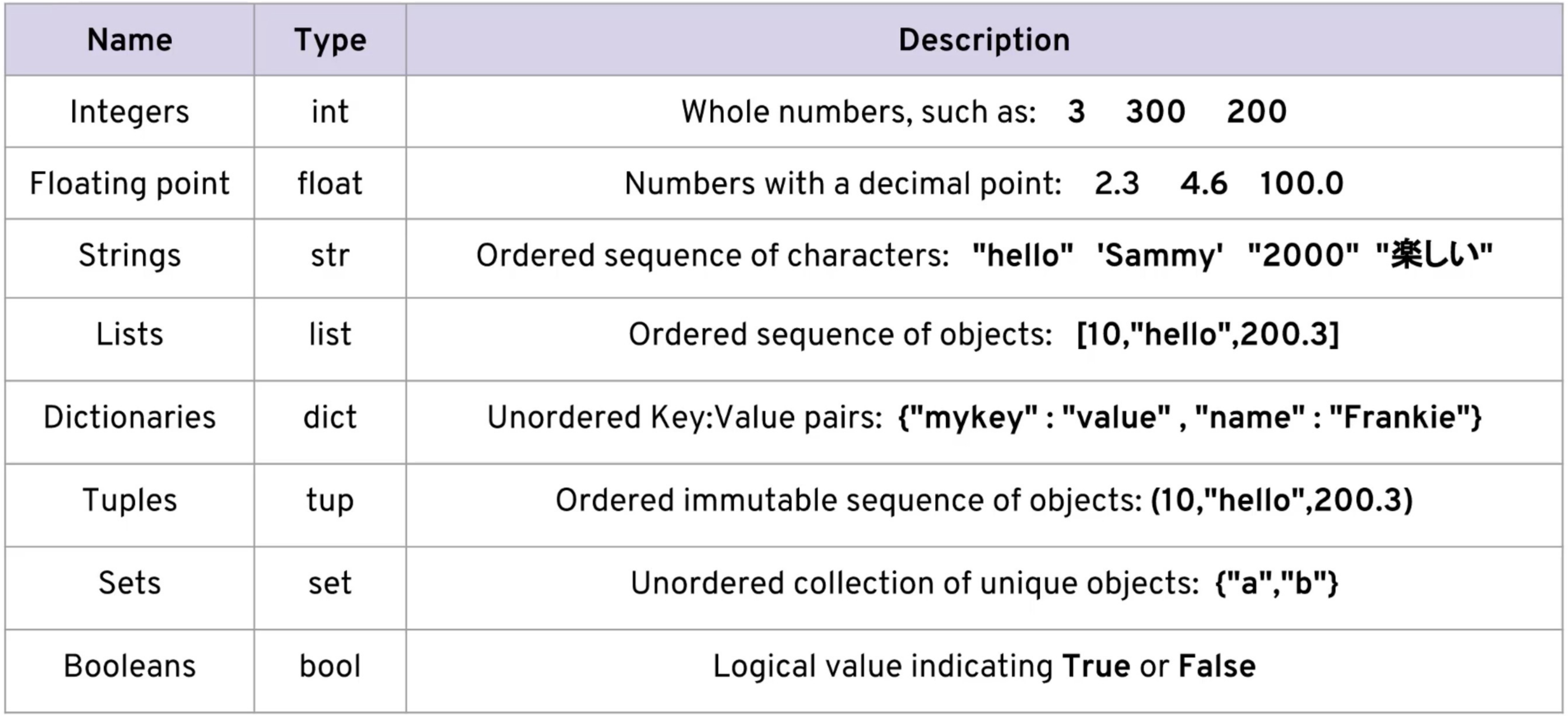
More Importantly – Deep Learning Pack

https://pytorch.org/



Deep Learning Libraries

Data Structures



Name	Type	Description
Integers	int	Whole numbers, such as: **3 300 200**
Floating point	float	Numbers with a decimal point: **2.3 4.6 100.0**
Strings	str	Ordered sequence of characters: **"hello" 'Sammy' "2000" "楽しい"**
Lists	list	Ordered sequence of objects: **[10,"hello",200.3]**
Dictionaries	dict	Unordered Key:Value pairs: **{"mykey" : "value" , "name" : "Frankie"}**
Tuples	tup	Ordered immutable sequence of objects: **(10,"hello",200.3)**
Sets	set	Unordered collection of unique objects: **{"a","b"}**
Booleans	bool	Logical value indicating **True** or **False**

Numbers: Integers and Floats + Logical Operations

Numbers: Integers and Floats + Logica Operations

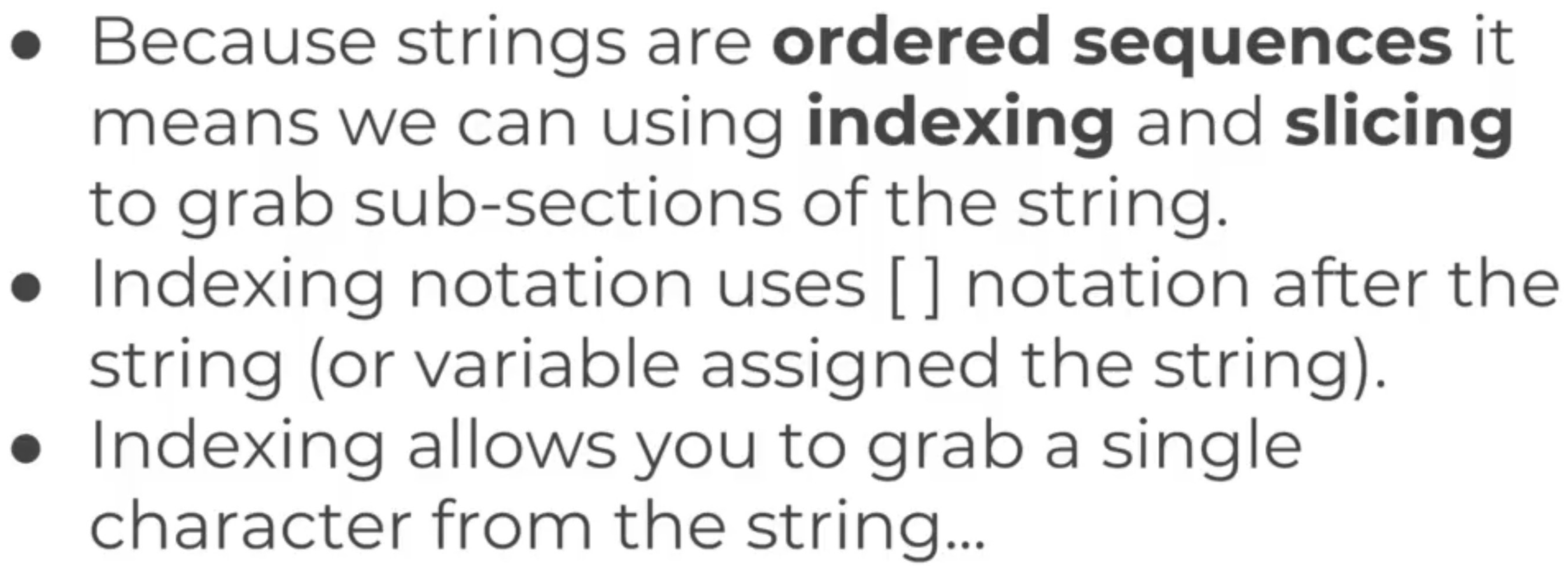
Open: 01-numbers.ipynb

Strings

Strings

- Strings are sequences of characters, using the syntax of either single quotes or double quotes:
  - **'hello'**
  - **"Hello"**
  - **" I don't do that "**

Strings



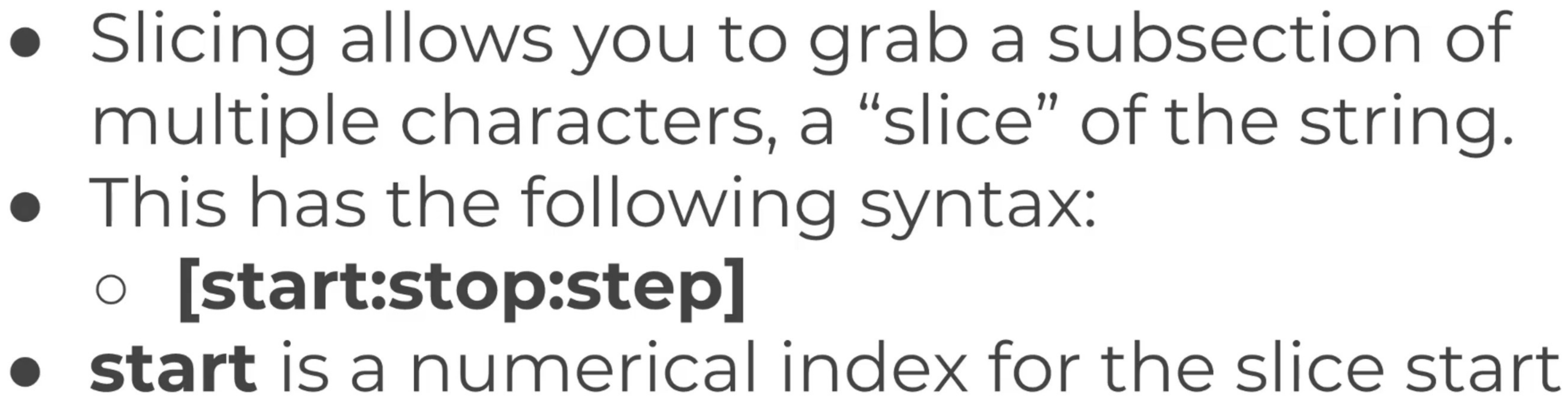
- Because strings are **ordered sequences** it means we can using **indexing** and **slicing** to grab sub-sections of the string.
- Indexing notation uses [ ] notation after the string (or variable assigned the string).
- Indexing allows you to grab a single character from the string...

Strings

- These actions use [ ] square brackets and a number index to indicate positions of what you wish to grab.

Character :	**h**	**e**	**l**	**l**	**o**
Index :	**0**	**1**	**2**	**3**	**4**

Strings



- Slicing allows you to grab a subsection of multiple characters, a “slice” of the string.
- This has the following syntax:
    - **[start:stop:step]**
- **start** is a numerical index for the slice start

Strings

Open: 02-strings.ipynb

Lists

Lists

- Lists are ordered sequences that can hold a variety of object types.
- They use [] brackets and commas to separate objects in the list.
    - **[1,2,3,4,5]**
- Lists support indexing and slicing. Lists can be nested and also have a variety of useful methods that can be called off of them.

Open: 03-Lists.ipynb

Dictionaries

Dictionaries

- Dictionaries are unordered mappings for storing objects. Previously we saw how lists store objects in an ordered sequence, dictionaries use a key-value pairing instead.
- This key-value pair allows users to quickly grab objects without needing to know an index location.

Dictionaries

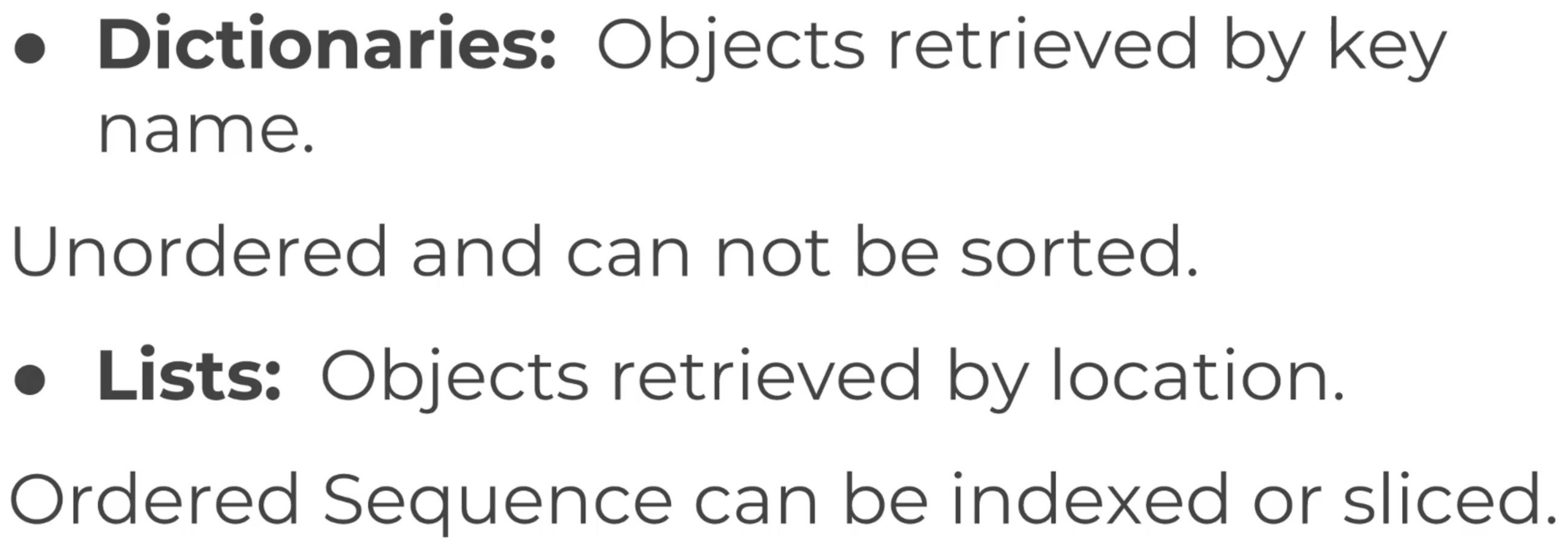
- Dictionaries use curly braces and colons to signify the keys and their associated values.

```
{'key1':'value1','key2':'value2'}
```

- So when to choose a list and when to choose a dictionary?

Open: 04-dictionaries.ipynb

Dictionaries



- **Dictionaries:** Objects retrieved by key name.

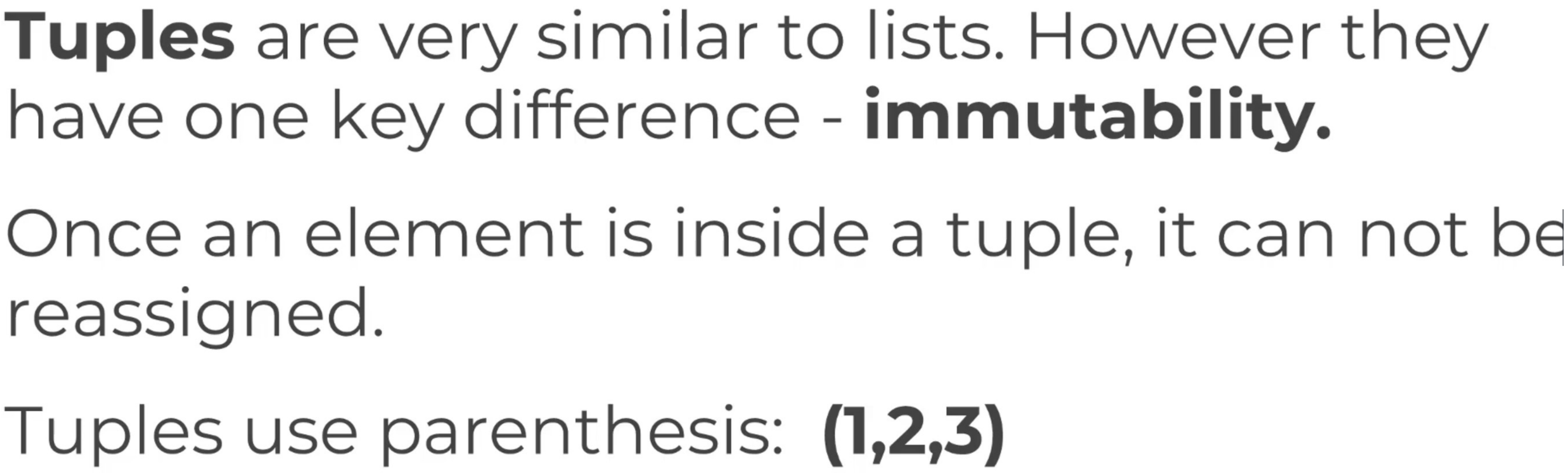
Unordered and can not be sorted.

- **Lists:** Objects retrieved by location.

Ordered Sequence can be indexed or sliced.

Tuples and Sets

Tuples and Sets

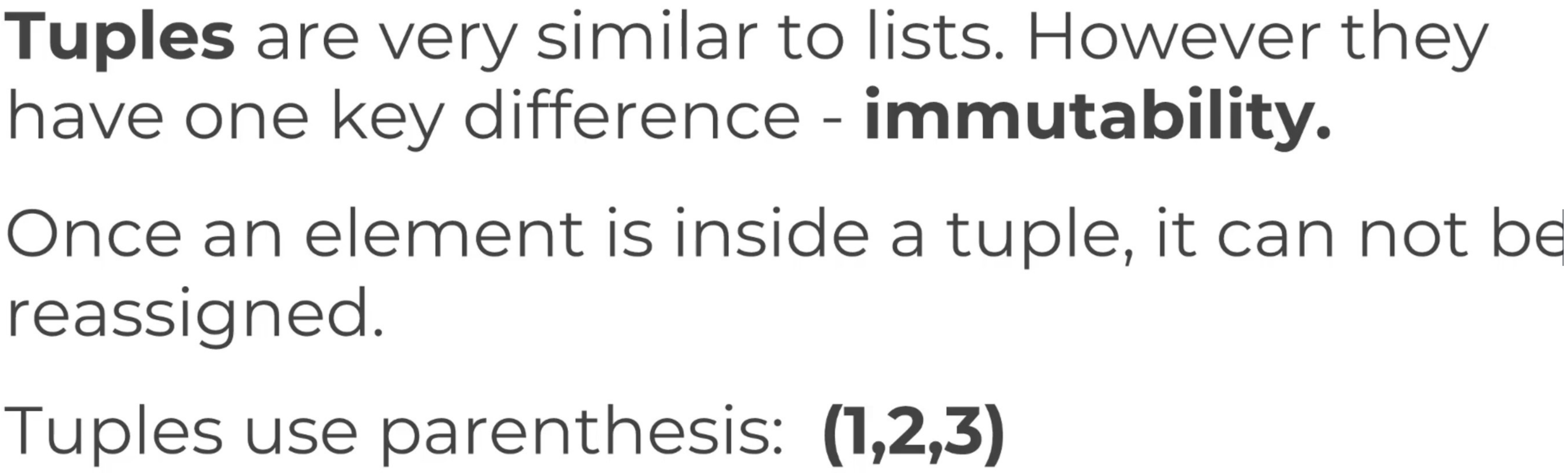


Tuples are very similar to lists. However they have one key difference - **immutability.**

Once an element is inside a tuple, it can not be reassigned.

Tuples use parenthesis: **(1,2,3)**

Tuples and Sets

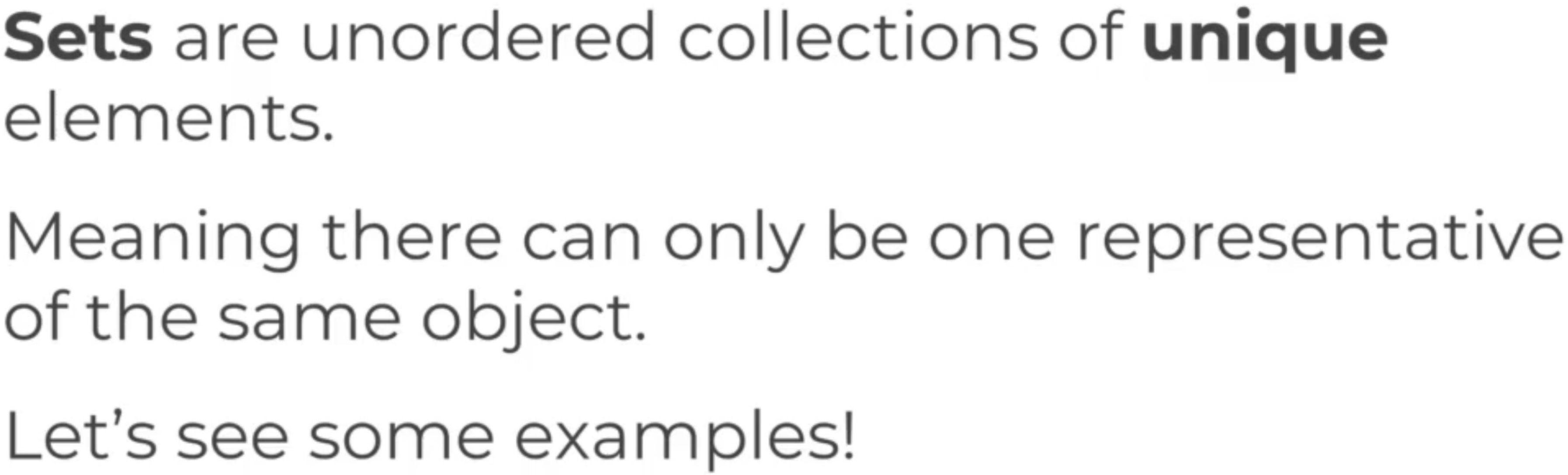


Tuples are very similar to lists. However they have one key difference - **immutability.**

Once an element is inside a tuple, it can not be reassigned.

Tuples use parenthesis: **(1,2,3)**

Tuples and Sets



Sets are unordered collections of **unique** elements.

Meaning there can only be one representative of the same object.

Let’s see some examples!

Tuples and Sets

Open: 05-Tuples_Sets_Unpacking.ipynb

Conditional Flow and Functions

Conditional Flow

Often in programming we want to perform cetain actions based on certain conditions, this is called condtional flow or conditional expressions

These conditonal expressions mimic human reasoning or actions, e.g. the python if statement is exactly equivalent to the “if” in the expression, if it rains get an umbrella

If, For, While, and Functions

Open: 06-If_For_While_Functions.ipynb

Classes and Objects in Python

Slides from: https://courses.cs.washington.edu/courses/cse142/11au/python/08-classes-objects.ppt

OOP, Defining a Class

- Python was built as a procedural language
  - OOP exists and works fine, but feels a bit more "tacked on"

- Declaring a class:

```
class name:
    statements
```

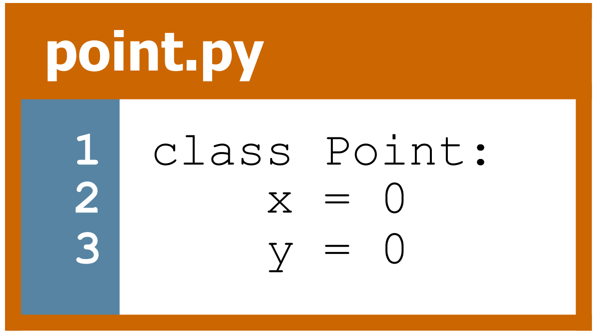
Fields

name = value

- Example:

```
class Point:
    x = 0
    y = 0

# main
p1 = Point()
p1.x = 2
p1.y = -5
```



point.py

```
class Point:
    x = 0
    y = 0
```

- can be declared directly inside class (as shown here) or in constructors (more common)
- Python does not really have encapsulation or private fields
    - relies on caller to "be nice" and not mess with objects' contents

Using a Class

`import` **class**

- client programs must import the classes they use

point_main.py

```python
from Point import *

# main
p1 = Point()
p1.x = 7
p1.y = -3
...

# Python objects are dynamic (can add fields any time!)
p1.name = "Tyler Durden"
```

Object Methods

```
def name(self, parameter, ..., parameter):
      statements
```

- `self` *must* be the first parameter to any object method
  - represents the "implicit parameter" (`this` in Java)
- *must* access the object's fields through the `self` reference

```
class Point:
    def translate(self, dx, dy):
        self.x += dx
        self.y += dy
    ...
```

"Implicit" Parameter (`self`)

- Python: `self`, explicit

```
def translate(self, dx, dy):
    self.x += dx
    self.y += dy
```

  - Exercise: Write `distance`, `set_location`, and `distance_from_origin` methods.

Exercise Answer

point.py

```python
from math import *

class Point:
    x = 0
    y = 0

    def set_location(self, x, y):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return sqrt(self.x * self.x + self.y * self.y)

    def distance(self, other):
        dx = self.x - other.x
        dy = self.y - other.y
        return sqrt(dx * dx + dy * dy)
```

Calling Methods

- A client can call the methods of an object in two ways:
  - (the value of `self` can be an implicit or explicit parameter)

1) **object**.**method**(**parameters**)

or

2) **Class**.**method**(**object**, **parameters**)

- Example:

```
p = Point(3, -4)
p.translate(1, 5)
Point.translate(p, 1, 5)
```

Constructors

```
def __init__(self, parameter, ..., parameter):
        statements
```

- a constructor is a special method with the name `__init__`
- Example:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    ...
```

  - How would we make it possible to construct a `Point()` with no parameters to get (0, 0)?

`toString` and `__str__`

```
def __str__(self):
        return string
```

- equivalent to Java's `toString` (converts object to a string)
- invoked automatically when `str` or `print` is called

Exercise: Write a `__str__` method for `Point` objects that returns strings like `"(3, -14)"`

```
def __str__(self):
    return "(" + str(self.x) + ", " + str(self.y) + ")"
```

Complete Point Class

point.py

```python
from math import *

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return sqrt(self.x * self.x + self.y * self.y)

    def distance(self, other):
        dx = self.x - other.x
        dy = self.y - other.y
        return sqrt(dx * dx + dy * dy)

    def translate(self, dx, dy):
        self.x += dx
        self.y += dy

    def __str__(self):
        return "(" + str(self.x) + ", " + str(self.y) + ")"
```

Operator Overloading

- **operator overloading**: You can define functions so that Python's built-in operators can be used with your class.
  - See also: http://docs.python.org/ref/customization.html

Operator	Class Method
-	`__neg__(self, other)`
+	`__pos__(self, other)`
*	`__mul__(self, other)`
/	`__truediv__(self, other)`

Unary Operators

-	`__neg__(self)`
+	`__pos__(self)`

Operator	Class Method
==	`__eq__(self, other)`
!=	`__ne__(self, other)`
<	`__lt__(self, other)`
>	`__gt__(self, other)`
<=	`__le__(self, other)`
>=	`__ge__(self, other)`

Exercise

- Exercise: **Write a `Fraction` class** to represent rational numbers like 1/2 and -3/8.

- Fractions should always be stored in reduced form; for example, store 4/12 as 1/3 and 6/-9 as -2/3.
  - Hint: A GCD (greatest common divisor) function may help.

- Define `add` and `multiply` methods that accept another `Fraction` as a parameter and modify the existing `Fraction` by adding/multiplying it by that parameter.

- Define `+`, `*`, `==`, and `<` operators.

Generating Exceptions

```
raise ExceptionType("message")
```

- useful when the client uses your object improperly
- types: `ArithmeticError`, `AssertionError`, `IndexError`, `NameError`, `SyntaxError`, `TypeError`, `ValueError`

- Example:

```
class BankAccount:
    ...
    def deposit(self, amount):
        if amount < 0:
            raise ValueError("negative amount")
        ...
```

Inheritance

```
class name(superclass):
        statements
```

- Example:

```
class Point3D(Point):    # Point3D extends Point
    z = 0
    ...
```

- Python also supports *multiple inheritance*

```
class name(superclass, ..., superclass):
        statements
```

(if > 1 superclass has the same field/method, conflicts are resolved in left-to-right order)

Calling Superclass Methods

- methods: **class.method(object, parameters)**
- constructors: **class.**__init__(**parameters**)

```
class Point3D(Point):
    z = 0
    def __init__(self, x, y, z):
        Point.__init__(self, x, y)
        self.z = z

    def translate(self, dx, dy, dz):
        Point.translate(self, dx, dy)
        self.z += dz
```

Q&A