# Reading 04

## ☆ Article: The python language

**Python** is:

- an *interpreted* language
- under *open source* licence
- multi-platform

**Ex**

b = 'hello'

b+b is 'hellohello'

2*b is 'hellohello'

**Types in python!**

- int
- float
- complex (a = 1.5 + 0.5j) — a.real, a.imag
- bool
- lists (you can count from the end using negative indices)
  - ↳ colors = ['red', 'blue', 'white']
    colors[-1] is 'white'
  - ↳ you can also acquire sublists
    - ↳ Ex colors[2:4] → index 2 to 4
      colors[3:] → index 3 to end
      colors[:3] → index start to 3
      colors[::2] → every other index
  - ↳ elements of a list may have different data types
  - ↳ colors.append('pink') → adds 'pink' to end of list
  - ↳ colors.pop() → removes and returns last item
  - ↳ colors.extend(['pink', 'purple']) → extends list
  - ↳ colors[::-1] reverses the list
    - ↳ or colors.reverse()

## • Strings

- A string behaves just like a list
  - ↳ you can iter into it
- A string is an immutable object and it is not possible to modify its contents.
  - ↳ One may create new strings from old ones
- There is a .replace() method to modify a string though.
- String formatting:

  "integer %i, float %f, another string: %s" % (1, 0.1, "hello")

## • Dictionaries

- Ex tel = {'emannelle': 5752, 'sebastian': 5578}
  - ↳ key-value pairs
  - ↳ tel.keys()
  - ↳ tel.values()
  - ↳ 'sebastian' in dict → returns True

## • Tuples

- basically immutable lists
- written between parentheses or just seperated by commas
  - ↳ Ex t = 12345, 54321, 'hello'

## • Sets

- unordered, unique items
  - ↳ Ex s = set(('a', 'b', 'c', 'd'))

- A single object can have several names bound to it:

Ex:
```
a = [1,2,3]
b = a
a
```

- Remember to change a list in place, use indexing / slices.

## Control Flow

- the statement `if <OBJECT>`
  - Evaluates to False for:
    - any number equal to 0
    - an empty container (list, tuple, set, dictionary, ...)
    - False, None
  - Evaluates to True for:
    - everything else

- Statement `a is b`:
  - returns True if a and b are the same object
  - returns False otherwise

- Statement `a in b`
  - returns True for any collection b such that b contains a.
  - returns False otherwise
  - for dictionaries, it checks if a is a key of b.

- Examples of complex iteration:

```
vowels = 'aeiouy'
```

```
for i in 'powerful':
    if i in vowels:
        print(i)
```

```
o
e
u
```

```
message = "Hello how are you?"
message.split() # returns a list
```

```
['Hello', 'how', 'are', 'you?']
```

```
for word in message.split():
    print(word)
```

```
Hello
how
are
you?
```

- enumerate() allows you to keep track of index as well

```
for index, item in enumerate(words):
    print((index, item))
```

```
(0, 'cool')
(1, 'powerful')
(2, 'readable')
```

- Looping over dictionary:

```
d = {'a': 1, 'b':1.2, 'c':1j}
```

```
for key, val in d.items():
    print('Key: %s has value: %s' % (key, val))
```

```
Key: a has value: 1
Key: b has value: 1.2
Key: c has value: 1j
```

- List comprehension to create lists:

```
[i**2 for i in range(4)]
```

```
[0, 1, 4, 9]
```

## • Defining Functions

- If you do not specify a return value, then it is None by default.

- you can also have optional parameters, if you default their values.

Ex

```
def double_it(x=2):
    return x * 2
```

x is defaulted to 2 unless user passes something

- Ex of a potentially useful slicer function:

```
def slicer(seq, start=None, stop=None, step=None):
    """Implement basic python slicing."""
    return seq[start:stop:step]
```

- Be careful when passing mutable objects to functions (they can be modified in-place):

```
def try_to_modify(x, y, z):
    x = 23
    y.append(42)
    z = [99] # new reference
    print(x)
    print(y)
    print(z)
```

```
a = 77    # immutable variable
b = [99]  # mutable variable
c = [28]
try_to_modify(a, b, c)
```

```
23
[99, 42]
[99]
```

```
print(a)
```

```
77
```

```
print(b)
```

```
[99, 42]
```

```
print(c)
```

```
[28]
```

- global variables cannot be modified within functions, unless you declare them as global in the function:

```
def setx(y):
    global x
    x = y
    print('x is %d' % x)
```

```
setx(10)
```

```
x is 10
```

```
x
```

```
10
```

- Special Parameters:

Special forms of parameters:

- `*args`: any number of positional arguments packed into a tuple
- `**kwargs`: any number of keyword arguments packed into a dictionary

```
def variable_args(*args, **kwargs):
    print('args is', args)
    print('kwargs is', kwargs)
```

```
variable_args('one', 'two', x=1, y=2, z=3)
```

```
args is ('one', 'two')
kwargs is {'x': 1, 'y': 2, 'z': 3}
```

- Functions are objects (specifically first-class objects) which means they can be:
  - assigned to a variable
  - an item in a list (or any collection)
  - passed as an argument to another function

- Reusing Code: Scripts and modules

  - Standalone scripts can take in command-line-arguments
    - ↳ need to 'import sys'
    - ↳ command-line args are stored in `sys.argv`

  - importing objects from modules example:

    'from os import listdir'

- you can import your own modules!
  - ↳ maybe you create demo.py module!
  - ↳ Then you can 'import demo' and do 'demo.print_a()' in the new script
  - ↳ you can also do 'help(demo)' if needed

```
"A demo module."


def print_b():
    "Prints b."
    print("b")


def print_a():
    "Prints a."
    print("a")


c = 2
d = 2
```

- A package is a directory that contains many modules.

## Input and Output:

- writing to files:

```
f = open('workfile', 'w') # opens the workfile file
type(f)
```

```
_io.TextIOWrapper
```

```
f.write('This is a test \nand another test')
f.close()
```

To read from a file

```
f = open('workfile', 'r')
s = f.read()
print(s)
```

```
This is a test
and another test
```

```
f.close()
```

- Iterating over file in read mode!

```
f = open('workfile', 'r')

for line in f:
    print(line)
```

```
This is a test

and another test
```

```
f.close()
```

- File modes!
  - 'r' → read only
  - 'w' → write only
    - ↳ creating new or overwriting existing
  - 'a' → append to a file
  - 'r+' → "read and write"
  - 'b' → "binary mode"

# • Standard Library

- `os` module : operating system functionality
  - ↳ need to 'import os'

- `os.getcwd()` equivalent to pwd
- `os.listdir(os.curdir)` → ls
- `os.mkdir('junkdir')` → mkdir
- `os.rename('junkdir', 'foodir')`
- `fp = open('junk.txt', 'w')`
  `fp.close()`
  → making file
- `os.remove('junk.txt')` → deleting file

## • Different Path Commands:

```
fp = open('junk.txt', 'w')
fp.close()
a = os.path.abspath('junk.txt')
a
```

```
'/home/runner/work/scientific-python-lectures/scientific-python-lectures/intro/language/junk
```

```
os.path.split(a)
```

```
('/home/runner/work/scientific-python-lectures/scientific-python-lectures/intro/language
 'junk.txt')
```

```
os.path.dirname(a)
```

```
'/home/runner/work/scientific-python-lectures/scientific-python-lectures/intro/language'
```

```
os.path.basename(a)
```

```
'junk.txt'
```

```
os.path.splitext(os.path.basename(a))
```

```
('junk', '.txt')
```

```
os.path.exists('junk.txt')
```

```
True
```

```
os.path.isfile('junk.txt')
```

```
True
```

```
os.path.isdir('junk.txt')
```

```
False
```

```
os.path.expanduser('~/local')
```

```
'/home/runner/local'
```

```
os.path.join(os.path.expanduser('~'), 'local', 'bin')
```

```
'/home/runner/local/bin'
```

## • Running External Command Example:

`return_code = os.system('ls')`

- Checking environmental variables:
  - `os.environ.keys()`
  - `os.environ['SHELL']`

- `shutil` high level operations:
  - shutil.rmtree → recursively delete a directory tree
  - shutil.move → recursively move a file or directory to another tree
  - shutil.copy → copy files or directories

- `glob` Pattern matching on files:

```
import glob
glob.glob('*.txt')
```
→ matches all files ending in txt

- `pickle` easy persistence
  - useful to store arbitrary objects to a file. Not safe or fast!

- Exception Handling in Python

- try - except block example:

```
In [1]: while True:
   ....:     try:
   ....:         x = int(input('Please enter a number: '))
   ....:         break
   ....:     except ValueError:
   ....:         print('That was no valid number.  Try again...')
   ....:
Please enter a number: a
That was no valid number.  Try again...
Please enter a number: 1

In [2]: x
Out[9]: 1
```

- try / finally block example:

```
In [1]: try:
   ....:     x = int(input('Please enter a number: '))
   ....: finally:
   ....:     print('Thank you for your input')
   ....:
Please enter a number: a
Thank you for your input
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[10], line 2
      1 try:
----> 2     x = int(input('Please enter a number: '))
      3 finally:
      4     print('Thank you for your input')
ValueError: invalid literal for int() with base 10: 'a'
```

# • Object-oriented programming (OOP)

## Goals of OOP!

- to organize the code
- to reuse code in similar contexts

• Ex

```
class Student(object):
    def __init__(self, name):
        self.name = name
    def set_age(self, age):
        self.age = age
    def set_major(self, major):
        self.major = major
```

```
anna = Student('anna')
anna.set_age(21)
anna.set_major('physics')
```

- set_age and set_major are methods
- name, age, and major are attributes

• Ex Inheritence

```
class MasterStudent(Student):
    internship = 'mandatory, from March to June'
```

```
james = MasterStudent('james')
james.internship
```

```
'mandatory, from March to June'
```

```
james.set_age(23)
james.age
```

```
23
```

- MasterStudent class inherits everything from Student class (above)
- Adds the internship attribute

# Reading 4 Continued

## ☆ Article: Python Data Structures

- Data structures are used to store a collection of related data.
- There are 4 built-in data structures in Python!
    1. list
    2. tuple
    3. dictionary
    4. set

### Lists

↳ ordered collection of items

↳ square brackets

↳ mutable

### Classes and Objects

- An object is an instance of a class
- Classes have:
    - ↳ attributes
    - ↳ methods

### Tuples

↳ hold multiple objects

↳ optional pair of parentheses

↳ immutable

↳ index into tuple like this: my_tuple[2]

## Dictionary

↳ key-value pairs

↳ keys must be unique and immutable objects

↳ values can be mutable or immutable objects.

## Sequences

- Lists, tuples, and strings are examples of sequences.
- Major features are membership tests (i.e. the `in` and `not in` expressions) and indexing operations.
- my_sequence[:] returns a copy of the whole sequence.
- my_sequence[::{step}] returns the sequence elements at each corresponding {step}

## Set

- unordered collection of items

↳ if you have 2 sets "bri" and "bric" and you want to find the intersection, you can:

bri & bric or bri.intersection(bric)

- When you create an object and assign it to a variable, the variable only refers to the object and does not represent the object itself!

↳ The variable name points to where the object is stored.