A Quick Tour of Python

COMP2611: Data Structures
2019/2020 Semester I

Python

- Will use Python 3.7 for the course
  - Will use this for many courses going forward
- Simple but powerful (and useful) language
- Pseudocode-esque in syntax
  - Easy to learn
  - Easy to translate to and from Pseudocode
- Since, easy to learn, we expect that you will able learn much of it on your own
  - Today, we will cover some basics
  - Should finish agenda (whether on slides or not) on your own if we don’t in class
  - Use supplementary material on course site

Zen of Python

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.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Python

- Python is an interpreted language
  - Unlike C/C++ which are compiled
  - Python interpreter parses and executes programme as opposed to generating machine code

- Python is dynamically (gradually) typed
  - Variables don’t need explicit types
  - Gradual typing - optional type annotations

- Python is strongly typed

  - More precisely, duck typed (if it looks like a duck, and quacks…)

  - Will not auto-cast for you in **most** circumstances
    - Some string conversion done in print, auto-conversion to bool using __*bool*__ magic method

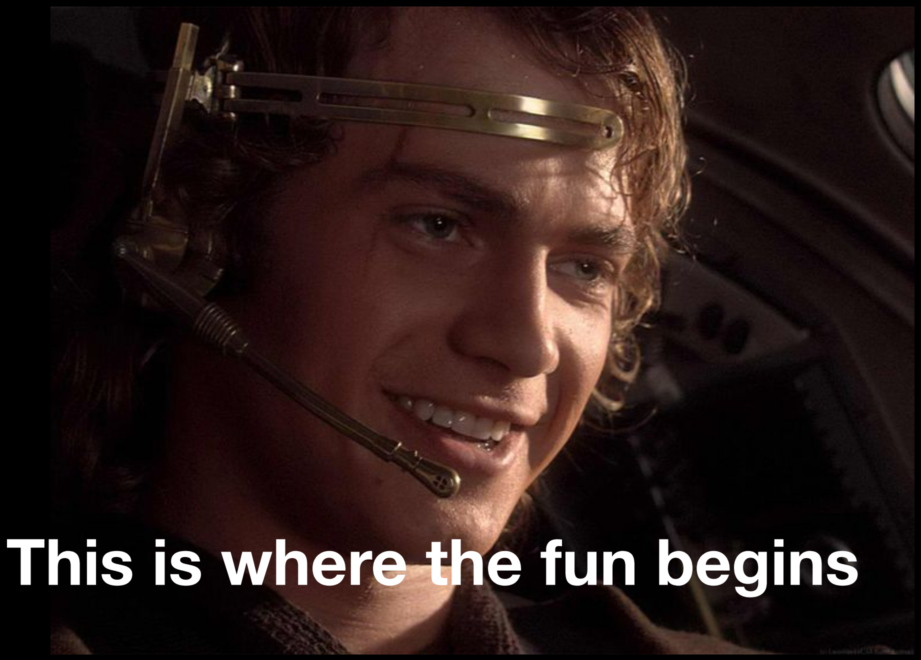
Python

- Two modes of operation:
  - REPL - Read Eval Print Loop
    - Interactive interpreter sessions
    - Useful for quick, temporary coding and sanity checks
    - Several flavours: vanilla, IPython, Jupyter
    - Won’t focus on this one, but good to learn
  - Source code
    - Code in source file
    - Call interpreter to process and execute source file
    - Uses “.py” file extension
    - Source code:
      - Modules (like header files in C/C++)
      - Main files (for execution)
- Several good IDEs - we suggest PyCharm

Agenda

- Fundamentals:
  - Variables, Types, If statements
  - Loops
  - Function calls
  - File I/O
  - Lists (Python equivalent of arrays)
  - Tuples
- Custom functions
- Classes and Objects (Python “equivalent” of structs)

Let’s starting look at some code :-)



Python

Python strings delimited by either double or single quotes

```python
print("Hello World")
```

print function
writes string to console
adds newline by default

No Semicolons!

C++

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World" << endl;
    return 0;
}
```

Python

Reads from console and returns string

```python
name = input('Please enter your name')
print("Hello ", name)
```

Python strings delimited by either double or single quotes. No explicit access to char needed

C++

```cpp
#include <iostream>

#define MAX_NAME_LENGTH 30

using namespace std;

int main()
{
    char name[MAX_NAME_LENGTH];
    cout << "Please enter your name" << endl;
    cin >> name;
    cout << "Hello " << name << endl;
    return 0;
}
```

Python

usual operators

(+, -, *, /, <, >, ==, !=, etc…)

available

```python
x = 2
y = 3.0
sum_x_y = x + y
x_raised_to_y = x ** y

x_as_float = float(x)
y_as_int = int(y)

name0 = 'Alice'
name1 = 'Bob'
sentence = name0 + ' loves ' + name1
x_as_string = str(x)
y_as_string = str(y)
```

casting between numeric types

cast to string

String concat operator

C++

```cpp
int x = 2;
float y = 3;
float sum_x_y = x + y;
float x_raised_to_y = pow(x, y)// Need to use cmath header
float x_as_float = (float)x;
int y_as_int = (int)y;

char name0[MAX_NAME_LENGTH] = "Alice";
char name1[MAX_NAME_LENGTH] = "Bob";
char sentence[MAX_NAME_LENGTH * 3];
sentence[0] = '\0'
strcat(sentence, name0); // from cstring header
strcat(sentence, " loves ");
strcat(sentence, name1);

char x_as_string[10];
char y_as_string[10];

sprintf(x_as_string, "%d", x);
sprintf(y_as_string, "%f", y);
```

Python

```python
# can also write as a_string = “string"
a_string = 'string'
a_string_length = len(a_string)
another_string = "another"
another_less_than_a = a_string < another_string
are_they_equal = a_string == another_string
```

string comparison
as operators

C++

```cpp
char a_string[10] = "string";
int a_string_length = strlen(a_string);
char another_string = "another"
bool another_less_than_a =
    0 < strcmp(another_string, a_string);
bool are_they_equal =
    0 == strcmp(another_string, a_string);
```

Python

C++

```python
programme = 'programme'
p = programme[0]
r = programme[1]
prog = programme[0:4]
```

strings are 0-indexed

```cpp
char programme[20] = "programme";
char p = programme[0];
char r = programme[1];
char prog[20];
memcpy(prog, programme, 4);
prog[4] = '\0';
```

Python

```python
bx = True
by = True
bz = False
```

Boolean values in Python

```python
bx_and_bz = bx and bz
bx_or_by = bx or by
not_bx = not bx

# print can process arbitrary types
print(bx_and_bz)

# print can process arbitrary types
print(bx, " and ", bz, " = ", bx_and_bz)


one_as_bool = bool(1)
zero_as_bool = bool(0)
empty_string_as_bool = bool('')
non_empty_string_as_bool = bool("foobar")
```

cast to bool types conversions

are implicit calls to magic methods

bool(x) is really *x.__bool__()*.

Apples for inter alia,

str, *int*, and *float*

C++

```cpp
bool bx = true;
  bool by = true;
  bool bz = false;

  bool bx_and_bz = bx && bz;
  bool bx_or_by = bx || by;
  bool not_bx = !bx;
  cout << bx << " and " << bz << " = "
       << bx_and_bz << endl;
```

Rest messy to do in C++ :(

Python

Notice the colon

```python
x = 4
if (x % 2) == 0:
    print(x, ' is even')
else:
    print(x, ' is odd')
```

Indents denote scope. VERY IMPORTANT

Rule of thumb: Would I put this in curly braces in C++? If so, then ident

1 ident = 4 spaces. NOT TABS!

C++

```cpp
int x = 4;
if ((x % 2) == 0)
{
    cout << x << " is even" << endl;
}
else
{
    cout << x << " is odd" << endl;
}
```

Python

```python
x = 9
if (x % 2) == 0:
    print(x, ' is even')
elif (x % 3) == 0:
    print(x, 'is divisible by 3')
else:
    print(x, ' is odd')
```

elif short for “else if”

C++

```cpp
int x = 9;
 if ((x % 2) == 0)
 {
     cout << x << " is even" << endl;
 }
 else if ((x % 3) == 0)
 {
     cout << x << " is divisible by 3" << endl;
 }
 else
 {
     cout << x << " is odd" << endl;
 }
```

Python

```python
x = 9
if (x % 2) == 0:
    print(x, ' is even')
elif (x % 3) == 0:
    print(x, 'is divisible by 3')
else:
    print(x, ' is odd')
```

elif short for “else if”

C++

```cpp
int x = 9;
 if ((x % 2) == 0)
 {
     cout << x << " is even" << endl;
 }
 else if ((x % 3) == 0)
 {
     cout << x << " is divisible by 3" << endl;
 }
 else
 {
     cout << x << " is odd" << endl;
 }
```

Python

```python
x = 0
while x < 4:
    print(x)
    x += 1   # Shorthand for x = x + 1
```

No ++ operator

C++

```cpp
int x = 0;
while (x < 4)
{
    cout << x << endl;
    x++;
}
```

Python

```python
def gcd(a, b):
    while b != 0:
        t = b
        b = a % b
        a = t
    return a

def print_hello():
    print("HELLO!")
```

Python functions without explicit *return* statement *return None* (Python equiv to NULL)

C++

```cpp
int gcd(int a, int b)
{
    int t;
    while (b != 0)
    {
        t = b;
        b = a % b;
        a = t;
    }
    return a;
}

void print_hello()
{
    cout << "HELLO!" << endl;
}
```

Python

Default arguments

```
def greet(name, greeting='Hello!'):
    print(greeting, ' My name is ', name)

greet('Alice') # prints "Hello! My name is Alice"
greet('Mario', 'Ciao!') # prints "Ciao! My name is Mario"
greet(greeting='Konichiwa.', name='Mitsuha') # prints "Konichiwa. My name is Mitsuha"
```

Supply arguments in different order

once you remember their name

Python

```python
def simple_function(x):
    return x * x

# functions can take other functions are arguments
# VERY useful. Much more than you might necessarily think
def differentiate(x, f, h=0.00001):
    numer = f(x + h) - f(x)
    denom = h
    frac = numer / denom
    return frac

deriv_at_3 = differentiate(3, simple_function) # returns 6
deriv_at_3_b = differentiate(3, lambda x: x * x) # returns 6
```

Anonymous function. Alternative way to write short functions when you need them as arguments to other functions

NB: Can also return functions from other functions.

Python

- Python uses dynamic arrays called lists
  - Dynamic arrays resize as needed on their own
  - 0-indexed like C/C++ arrays
  - Can hold data of more than one type

Python

```python
# Lists store sequences
li = []
# You can start with a prefilled list
other_li = [4, 5, 6]


# Add stuff to the end of a list with append
li.append(1)    # li is now [1]
li.append(2)    # li is now [1, 2]
li.append(4)    # li is now [1, 2, 4]
li.append(3)    # li is now [1, 2, 4, 3]
# Remove from the end with pop
li.pop()        # => 3 and li is now [1, 2, 4]
# Let's put it back
li.append(3)    # li is now [1, 2, 4, 3] again.

# Access a list like you would any array
li[0]   # => 1
# Look at the last element
li[-1]  # => 3

# concat lists using plus operator
another_li = li + other_li  # contains [1, 2, 4, 3, 4, 5, 6]
```

Python

```python
# Looking out of bounds is an IndexError
li[4]  # Raises an IndexError

# You can look at ranges with slice syntax.
# The start index is included, the end index is not
# (It's a closed/open range for you mathy types.)
li[1:3]   # => [2, 4]
# Omit the beginning and return the list
li[2:]    # => [4, 3]
# Omit the end and return the list
li[:3]    # => [1, 2, 4]
# Select every second entry
li[::2]   # =>[1, 4]
# Return a reversed copy of the list
li[::-1]  # => [3, 4, 2, 1]
# Use any combination of these to make advanced slices
# li[start:end:step]

# Make a one layer deep copy using slices
li2 = li[:]  # => li2 = [1, 2, 4, 3] but (li2 is li) will result in false.
```

Python

```python
# Check for existence in a list with "in" (sequential search)
1 in li  # => True

# Examine the length with "len()"
len(li)  # => 6

# Fill multiply lists as well
li3 = [1, 'a'] * 3 # contains [1, "a",1, "a",1, "a"]
```

Python

```
a_string = 'Alice Bob Catherine'
names0 = a_string.split() # ['Alice', 'Bob', 'Catherine']

b_string = 'Alice;Bob;Catherine'
names1 = b_string.split(';')
```

Separator. The default is whitespace.

Knowing how to split strings is VERY important

for reading data for assignments

Python

```
# Tuples are like lists but are immutable.
tup = (1, 2, 3)
tup[0]      # => 1
tup[0] = 3  # Raises a TypeError

# Note that a tuple of length one has to have a comma after the last element
but
# tuples of other lengths, even zero, do not.
type((1))   # => <class 'int'>
type((1,))  # => <class 'tuple'>
type(())    # => <class 'tuple'>

# You can do most of the list operations on tuples too
len(tup)         # => 3
tup + (4, 5, 6)  # => (1, 2, 3, 4, 5, 6)
tup[:2]          # => (1, 2)
2 in tup         # => True
```

Python

```
# You can unpack tuples (or lists) into variables
a, b, c = (1, 2, 3)  # a is now 1, b is now 2 and c is now 3
# You can also do extended unpacking
a, *b, c = (1, 2, 3, 4)  # a is now 1, b is now [2, 3] and c is now 4
# Tuples are created by default if you leave out the parentheses
d, e, f = 4, 5, 6  # tuple 4, 5, 6 is unpacked into variables d, e
and f
# respectively such that d = 4, e = 5 and f = 6
# Now look how easy it is to swap two values
e, d = d, e  # d is now 5 and e is now 4
```

Python

```python
for i in range(4):
    print(i)

for i in range(4, 8):
    print(i)

for i in range(4, 8, 2):
    print(i)
```

C++

```cpp
for(int i = 0; i < 4; i++)
{
    cout << i << endl;
}

for(int i = 4; i < 8; i++)
{
    cout << i << endl;
}

for(int i = 4; i < 8; i += 2)
{
    cout << i << endl;
}
```

Python

```python
animals = ["dog", "cat", "mouse"]
for i in range(0, len(animals)):
    # You can use format() to interpolate formatted strings
    print("{} is a mammal".format(animals[i]))
```

Python

Python's *for* is really a foreach loop!

Can loop without needing index

```
animals = ["dog", "cat", "mouse"]
for animal in animals:
    # You can use format() to interpolate formatted strings
    print("{} is a mammal".format(animal))
```

Python

Use *enumerate* when you need access to both index and element

```
for i, animal in enumerate(animals):
    print('{} is the animal in place {}'.format(animal, i))
```

Python

- Objects take the place of structs in Python
  - **Every** value in Python is an object
  - Classes provide blueprint for objects
  - Memory references (pointers) are stored in variables
  - Methods defined on objects to operate on their data
  - *is* operator checks equality based on memory reference
    - When comparing against *None*, **<u>ALWAYS</u>** use *is*

Python

reference to object itself

see note

```python
class Human:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def say_hello(self):
        print('Hello. I am ', self.name,
              ' and I am ', age, ' years old')

    def have_birthday(self):
        self.age += 1
```

All methods with that underscore naming conventions are magic methods. *__init__* is the constructor magic method that handles object creation

C++

```cpp
typedef struct
{
    char name[30];
    int age;
} Human;

Human create_human(char name[], int age)
{
    Human h;
    strcpy(h.name, name);
    h.age = age;
    return h;
}

void have_birthday(Human* h)
{
    h.age += 1;
}
```

Python

calls constructor

```python
alice = Human('Alice', 22)
alice.say_hello()
print(alice.name)
print(alice.age)
alice.have_birthday()
alice.say_hello()
```

C++

```cpp
typedef struct
{
    char name[30];
    int age;
} Human;

Human create_human(char name[], int age)
{
    Human h;
    strcpy(h.name, name);
    h.age = age;
    return h;
}

void have_birthday(Human* h)
{
    h.age += 1;
}
```

Python

- Many more magic methods
    - __str__ defines how to cast object to string
    - __repr__ defines how to generate string for printing
    - __bool__ defines how to cast to bool
    - __hash__ defines how to compete hash code
    - __add__, __mul__, __sub__, etc.. define arithmetic operations
    - __lt__, __eq__, __gte__, etc .. define comparisons
    - __len__ defines how to compute length
    - There are **a lot** more, but these are the most important ones to consider for now. Only use what you need!

Python

```python
class Human:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def say_hello(self):
        print('Hello. I am ', self.name, ' and I am ', age, ' years old')

    def have_birthday(self):
        self.age += 1

    def __str__(self):
        s = 'Name: ' + self.name
        s = s +  ' Age: ' + str(self.age)
        return s

    def __repr__(self):
        s = 'Name: ' + self.name
        s = s +  ' Age: ' + str(self.age)
        return s


alice = Human('Alice', 22)
alice_string = str(alice) # calls alice.__str__()
print(alice_string)
print(alice)
```

Python

```python
class Rational:
    def __init__(self, numer, denom):
        gcd_val = gcd(numer, denom)
        self.numer = numer / gcd_val
        self.denom = denom / gcd_val

    def __str__(self):
        return str(self.numer) + '/' + str(self.denom)

    def __repr__(self):
        return str(self.numer) + '/' + str(self.denom)

    # defines +
    def __add__(self, other_rational):
        new_denom = self.denom * other_rational.denom
        new_numer = self.numer * other_rational.denom
        new_numer += other_rational.numer * self.denom
        return Rational(new_numer, new_denom)

    # defines unary -, e.g, -x
    def __negate__(self):
        return Rational(-self.numer, self.denom)

    # defines binary -, e.g x - y
    def __sub__(self, other_rational):
        # Because we already defined __add__ and __negate__ above
        # we can use the operations that call down to them
        return self + -other_rational

    # defines equality e.g. x == y
    def __eq__(self, other_rational):
        return self.numer = other_rational.numer and self.denom == other_rational.denom
```

Python

```
r1 = Rataional(2, 4)
r2 = Rational(1, 2)
print(r1 == r2) # prints True
r3 = Rational(1, 3)
print(r2 - r3) # prints "1/6"
```

Should try implementing multiplication, division, and the other comparison operators at home

Python

```python
class Stack:
    def __init__(self):
        self.contents = []

    def push(self, item):
        self.contents.append(item)

    def pop(self):
        return.contents.pop()

    def __len__(self):
        return len(self.contents)


s = Stack()
s.push(2)
s.push(3)
print(s.pop()) # should print 3
print(len(s))
```

Python

```
fp = open('myfile.txt', 'r')
for line in fp:
    do_things(line)
fp.close()
```

```
with open('myfile.txt', 'r') as fp:
    for line in fp:
        do_things(line)
```

For each line in the file…

Do the same thing, but the second closes
file handle after iterating over lines in the file

Python

```
li = # list of lines to write as strings
fp = open('myfile.txt', 'w')
for line in li:
    fp.write(line + '\n')
fp.close()
```