**Introduction to Python Development**

# Environment Setup

## Installing Python3.7 on CentOS7

**Download and Install Python 3 from Source**

$ sudo -i

$ yum groupinstall -y "development tools"

$ yum install -y \

libffi-devel \

zlib-devel \

bzip2-devel \

openssl-devel \

ncurses-devel \

sqlite-devel \

readline-devel \

tk-devel \

gdbm-devel \

db4-devel \

libpcap-devel \

xz-devel \

expat-devel \

postgresql-devel

$ cd /usr/src

$ wget http://python.org/ftp/python/3.7.2/Python-3.7.2.tar.xz

$ tar xf Python-3.7.2.tar.xz

$ cd Python-3.7.2

$ ./configure --enable-optimizations

$ make altinstall

$ exit

*Important:* make altinstall causes it to not replace the built in python executable.

Using **sudo nano /etc/sudoers** (or your preferred text editor), ensure that secure\_path in /etc/sudoers file includes /usr/local/bin. The line should look something like this:

Defaults secure\_path = /sbin:/bin:/usr/sbin:/usr/bin:/usr/local/bin

**Upgrade Pip (might not be necesary)**

The version of pip that we have might be up-to-date, but it's a good practice to try to update it after the installation. We need to use the pip3.7 executable because we're working with Python 3, and we use sudo so that we can write files under the /usr/local directory.

$ sudo pip3.7 install --upgrade pip

## Installing Python3.7 on Debian/Ubuntu

**Download and Install Python 3 from Source**

sudo -i

$ apt update -y

$ apt install -y \

wget \

build-essential \

libffi-dev \

libgdbm-dev \

libc6-dev \

libssl-dev \

zlib1g-dev \

libbz2-dev \

libreadline-dev \

libsqlite3-dev \

libncurses5-dev \

libncursesw5-dev \

xz-utils \

tk-dev

$ cd /usr/src

$ wget http://python.org/ftp/python/3.7.2/Python-3.7.2.tar.xz

$ tar xf Python-3.7.2.tar.xz

$ cd Python-3.7.2

$ ./configure --enable-optimizations

$ make altinstall

*Note:* make altinstall causes it to not replace the built in python executable.

Ensure that secure\_path in /etc/sudoers file includes /usr/local/bin

**Upgrade Pip (might not be necesary)**

The version of pip that we have might be up-to-date, but it's a good practice to try to update it after the installation. We need to use the pip3.7 executable because we're working with Python 3, and we use sudo so that we can write files under the /usr/local directory.

$ pip3.7 install --upgrade pip

## Picking a Text Editor or IDE

**Terminal based editors**

There are two terminal based editors I would consider to be extremely popular:

* [Vim](https://www.vim.org/) - Modal editor, extremely customizable.
* [Emacs](https://www.gnu.org/software/emacs/) - Unbelievably customizable, not modal (at least not by default).

Both of these tools are either pre-installed or readily available on all major Linux distros.

**GUI Based Editors**

GUI based editors can be extremely powerful and more aestetically pleasing than terminal based editors. This list is comprised of classic "text editors", but most of them can be enhanced using plugins that add additional functionality. I'm going to divide them into two camps: native applications and Electron applications (built using JavaScript). This seems like a weird distinction, but plenty of people don't like the resource overhead that running Electron based applications requires.

Native:

* [SublimeText](https://www.sublimetext.com/) - Multi-platform. Very performant and extended using Python 3.
* [Notepad++](https://notepad-plus-plus.org/) - Windows only. Not as powerful as the others, but a good starter text editor that won't get in your way.

Electron Based:

* [Atom](https://atom.io/) - The original Electron based editor. Aestetically pleasing and very extendable through plugins.
* [VS Code](https://code.visualstudio.com/) - Arguably the most popular GUI based editor. Vast ecosystem of plugins and built-in debugger.

**IDEs**

The primary IDE that is used by people in the Python community would be [PyCharm](https://www.jetbrains.com/pycharm/). There is a free community edition and there is also a paid edition. To connect to a remote server to do your editing, you'll need to have the paid version.

## Setting up a Vim Development Environment

A mettre plus tard si nécessaire

# Running Python

## Using the REPL (Read, Evaluate, Print, Loop)

Pour lancer le python installé il suffit de taper python3.7

Tu quittes l’interface en tapant exit()

## Creating and Running Python Files

**Creating Our First Python Script**

Tu crées un script python avec vim puis pour le lancer tu as différents choix :

Soit à l’endroit où il y a ton fichier tu tapes : python3.7 NomDuFichier.py

**Setting a Shebang**

You'll most likely want your scripts to be:

1. Executable from anywhere (in our $PATH)
2. Executable without explicitly using the python3.7 CLI

Thankfully, we can set the process to interpret our scripts by setting a shebang at the top of the file:

*hello.py*

#!/usr/bin/env python3.7

print("Hello, World")

Then we need to make the file executable using chmod:

$ chmod u+x hello.py

Then run the script now by using **./hello.py**

**Adding Scripts to Our $PATH**

Now we need to make sure that we can put this in our $PATH. For this course, we'll be using a bin directory in our $HOME folder to store our custom scripts, but scripts can go into any directory that is in your $PATH.

Let's create a bin directory and move our script:

$ mkdir ~/bin

$ mv hello.py ~/bin/hello

Here's how we add this directory to the $PATH in our .bashrc (the .bashrc for this course already contains this):

$ export PATH=$PATH:$HOME/bin

Finally, let's run the hello script from our $PATH:

$ hello

Hello, World!

Using Comments

**Single Line Comment**

We can comment out a whole line:

# This is a full line comment

**What About Block Comments?**

Python does not have the concept of block commenting that you may have encountered in other languages. Many people mistake a triple-quoted string as being a comment, but it is not — it's a multi-line string. That being said, multi-line strings can functionally work like comments, but they will still be allocated into memory.

"""

This is not a block comment,

but it will still work when you really need for some lines of code to not execute.

"""

# Built-in Data Types

## Strings

A string is a sequence of characters grouped together.

We can combine strings using the + operator and multiply a string by a number using the \* operator:

>>> "pass" + "word"

'password'

>>> "Ha" \* 4

'HaHaHaHa'

An "object" encapsulates two things 1) State & 2) behavior. For the built in types, the state makes sense because it's the entire contents of the object. The behavior aspect means that there are functions that we can call on the instances of the objects that we have. A function bound to an object is called a "method". Here are some example methods that we can call on strings:

**find** locates the first instance of a character (or string) in a string. This function returns the index of the character or string.

>>> "double".find('s')

-1 ( car il ne trouve pas)

>>> "double".find('u')

2

>>> "double".find('bl')

3

**lower** converts all of the characters in a string to their lowercase versions (if they have one). This function returns a new string without changing the original, and this becomes important later.

>>> "TeStInG".lower() # "testing"

'testing'

>>> "another".lower()

'another'

>>> "PassWord123".lower()

'password123'

**Sauter une ligne :**

>>> print("New\nLine")

New

Line

## Numbers

There are to main types of numbers we'll use in Python: int and float. For the most part, we won't be calling methods on number types — instead, we will use a variety of operators.

>>> 2 + 2 # Addition

4

>>> 10 - 4 # Subtraction

6

>>> 3 \* 9 # Multiplication

27

>>> 5 / 3 # Division

1.66666666666667

>>> 5 // 3 # Floor division, always returns a number without a remainder

1

>>> 8 % 3 # Modulo division, returns the remainder

2

>>> 2 \*\* 3 # Exponent

8

**Converting Strings and Numbers**

It's not uncommon for us to need to convert from one type to another when writing a script. Python provides built-in functions for doing that with the built-in types. For strings and numbers, we can use the str, int, and float functions to convert from one type to another (within reason).

>>> str(1.1)

'1.1'

>>> int("10")

10

>>> int(5.99999)

5

>>> float("5.6")

5.6

>>> float(5)

5.0

## Booleans and None

**Booleans**

Booleans represent "truthiness" and Python has two boolean constants: True and False.

Notice that these both start with capital letters. Later we will learn about comparisons operations, and those will often return either True or False.

**Representing Nothingness with None**

Most programming languages have a type that represents the lack of a value, and Python is no different. The constant used to represent nothingness in Python is None. None is a "falsy", and we'll often use it to represent when a variable has no value yet.

An interesting thing to note about None is that if you type None into your REPL there will be nothing printed to the screen. That's because None actually evaluates into nothing.

enter = True

## Working with Variables

We can assign a value to a variable by using a single = and we don't need to (nor can we) specify the type of the variable.

>>> my\_str = "This is a simple string"

## Lists

**Lists**

A list is created in Python by using the square brackets ([, and ]) and separating the values by commas. Here's an example list:

>>> my\_list = [1, 2, 3, 4, 5]

**Reading from Lists**

To access an individual element of a list you can use the index and Python uses a zero based index system.

>>> my\_list[0]

1

>>> my\_list[2]

3

If we try to access an index that is too high (or too low) then we'll receive an error:

>>> my\_list[5]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

IndexError: list index out of range

To make sure that we're not trying to get an index that is out of range, we can test the length using the **len** function (and then subtract)

>>> my\_list = [1, 2, 3, 4, 5]

>>> len(my\_list)

5

Additionally, we can access subsections of a list by "slicing" it. We provide the starting index and the ending index (the object at that index won't be included).

>>> my\_list[0:2]

[1, 2]

>>> my\_list[1:]

[2, 3, 4, 5]

>>> my\_list[:3]

[1, 2, 3]

[1, 2, 3, 4, 5]

>>> my\_list[0::2]

[1, 3, 5]

**Afficher une liste à l’envers**

>>> my\_list[::-1]

[5, 4, 3, 2, 1]

**Modifying a List**

Unlike strings which can't be modified (you can't change a character in a string), you can change a value in a list using the subscript equals operation:

>>> my\_list[0] = "a"

>>> my\_list

['a', 2, 3, 4, 5]

If we want to add to a list we can use the .append method. This is an example of a method that modifies the object that is calling the method:

>>> my\_list.append(6)

>>> my\_list.append(7)

>>> my\_list

['a', 2, 3, 4, 5, 6, 7]

Lists can be added together (concatenated):

>>> my\_list + [8, 9, 10]

['a', 2, 3, 4, 5, 6, 7, 8, 9, 10]

Ou autre méthode :

>>> my\_list += [8, 9, 10]

>>> my\_list

['a', 2, 3, 4, 5, 6, 7, 8, 9, 10]

We can remove a section of a list by assigning an empty list to the slice:

>>> my\_list = ['a', 'b', 'c', 'd', 5, 6, 7]

>>> my\_list[4:] = []

>>> my\_list

['a', 'b', 'c', 'd']

Ici on supprime à partir de l’index 4 et tout ce qu’il y a après lui.

Removing items from a list based on value can be done using the .remove method:

>>> my\_list.remove('b')

>>> my\_list

['a', 'c', 'd']

Items can also be removed from the end of a list using the pop method:

>>> my\_list = ['a', 'c', 'd']

>>> my\_list.pop()

'd'

>>> my\_list

['a', 'c']

>>> my\_list

['a', 'c']

We can also use the pop method to remove items at a specific index:

>>> my\_list

['a', 'c']

>>> my\_list.pop(0)

'a'

>>> my\_list

['c']

## Tuples and Ranges

Tuples are a fixed-width, immutable sequence type. We create tuples using parenthesis ((, )) and at least one comma (,):

>>> point = (2.0, 3.0)

Since tuples are immutable, we don't have access to the same methods that we do on a list. We can use tuples in some operations like concatenation, but we can't change the original tuple that we created.

>>> point\_3d = point + (4.0,)

>>> point\_3d

(2.0, 3.0, 4.0)

One interesting characteristic of tuples is that we can unpack them into multiple variables at the same time:

>>> x, y, z = point\_3d

>>> x

2.0

>>> y

3.0

>>> z

4.0

**Ranges**

Ranges are an immutable sequence type that defines a start, a stop, and a step value, and then the values within are calculated as it is interacted with. This allows for ranges to be used in place of sequential lists and while taking less memory and including more items.

>>> my\_range = range(10)

>>> my\_range

range(0, 10)

>>> list(my\_range)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(range(1, 14, 2))

[1, 3, 5, 7, 9, 11, 13]

Notice that the "stop" value is not included in the range, the values are all up-until the stop.

## Dictionaries

Dictionaries are the main mapping type that we'll use in Python. This object is comparable to a Hash or "associative array" in other languages.

Things to note about dictionaries:

1. Unlike Python 2 dictionaries, as of Python 3.6 keys are ordered in dictionaries. Will need OrderedDict if you want this to work on another version of Python.
2. You can set the key to any IMMUTABLE type (no lists)
3. Avoid using things other than simple objects as keys.
4. Each key can only have one value (so don't have duplicates when creating a dict)

We create dictionary literals by using curly braces ({ and }), separating keys from values using colons (:), and separating key/value pairs using commas (,). Here's an example dictionary:

>>> ages = { 'kevin': 59, 'alex': 29, 'bob': 40 }

>>> ages

{'kevin': 59, 'alex': 29, 'bob': 40}

We can read a value from a dictionary by subscripting using the key:

>>> ages['kevin']

59

Keys can be added or changed using subscripting and assignment:

>>> ages['kayla'] = 21

>>> ages

{'kevin': 59, 'alex': 29, 'bob': 40, 'kayla': 21}

Items can be removed from a dictionary using the del statement or by using the pop method:

>>> del ages['kevin']

>>> ages

{'alex': 29, 'bob': 40, 'kayla': 21}

>>> ages = { 'kevin': 59, 'alex': 29, 'bob': 40 }

>>> ages.pop('alex')

29

>>> ages

{'kevin': 59, 'bob': 40}

We can use the get method:

>>> ages.get('kevin')

59

It's not uncommon to want to know what keys or values we have without caring about the pairings. For that situation we have the values and keys methods:

>>> ages = {'kevin': 59, 'bob': 40}

>>> ages.keys()

dict\_keys(['kevin', 'bob'])

>>> list(ages.keys())

['kevin', 'bob']

>>> ages.values()

dict\_values([59, 40])

>>> list(ages.values())

[59, 40]

**Alternative Ways to Create a dict Using Keyword Arguments**

There are a few other ways to create dictionaries that we might see, those being using the dict constructor with key/value arguments and a list of tuples:

>>> weights = dict(kevin=160, bob=240, kayla=135)

>>> weights

{'kevin': 160, 'bob': 240, 'kayla': 135}

>>> colors = dict([('kevin', 'blue'), ('bob', 'green'), ('kayla', 'red')])

>>> colors

{'kevin': 'blue', 'bob': 'green', 'kayla': 'red'}

## Python string formatting

The following example summarizes string formatting options in Python.

name = 'Peter'

age = 23

print('%s is %d years old' % (name, age))

print('{} is {} years old'.format(name, age))

print(f'{name} is {age} years old')

Les trois exemples vont afficher: Peter is 23 years old

print('%s is %d years old' % (name, age))

This is the oldest option. It uses the % operator and classic string format specifies such as %s and %d.

print('{} is {} years old'.format(name, age))

Since Python 3.0, the format() function was introduced to provide advance formatting options.

print(f'{name} is {age} years old')

Python f-strings are available since Python 3.6. The string has the f prefix and uses {} to evaluate variables.

**Python f-string expressions**

We can put expressions between the {} brackets.

bags = 3

apples\_in\_bag = 12

print(f'There are total of {bags \* apples\_in\_bag} apples')

The output is: There are total of 36 apples

# Control Flow

## Comparisons and Conditionals

Comparisons

>>> 1 < 2

True

>>> 0 > 2

False

>>> 2 == 1

False

>>> 2 != 1

True

>>> 3.0 >= 3.0

True

>>> 3.1 <= 3.0

False

>>> "this" == "this"

True

>>> "this" == "This"

False

>>> "b" > "a"

True

>>> "abc" < "b"

True

Notice that the string 'b' is considered greater than the strings 'a' and 'abc'. The characters are compared one at a time alphabetically to determine which is greater. This concept is used to sort strings alphabetically.

**The in Check**

We often get lists of information that we need to ensure contains (or doesn't contain) a specific item. To make this check in Python, we'll use the in and not in operations.

>>> 2 in [1, 2, 3]

True

>>> 4 in [1, 2, 3]

False

>>> 2 not in [1, 2, 3]

False

>>> 4 not in [1, 2, 3]

True

**Conditionals**

**if/elif/else**

>>> name = "Kevin"

>>> if len(name) >= 6:

... print("name is long")

... elif len(name) == 5:

... print("name is 5 characters")

... elif len(name) >= 4:

... print("name is 4 or more")

... else:

... print("name is short")

...

name is 5 characters

## Logic Operations

* [Boolean Operators](https://docs.python.org/3/library/stdtypes.html#boolean-operations-and-or-not)

**The not Operation**

Sometimes we want to know the opposite boolean value for something. This might not sound intuitive, but sometimes we want to execute an if statement when a value is False, but that's not how the if statement works. Here's an example of how we can use not to make this work:

>>> name = ""

>>> not name

True

>>> if not name:

... print("No name given")

...

>>>

We know that an empty string is a "falsy" value, so not "" will always return True

**The or Operation**

Occasionally, we want to carry out a branch in our logic if one condition OR the other condition is True. Here is where we'll use the or operation. Let's see or in action with an if statement:

>>> first = ""

>>> last = "Thompson"

>>> if first or last:

... print("The user has a first or last name")

...

The user has a first or last name

>>>

If both first and last were "falsy" then the print would never happen:

>>> first = ""

>>> last = ""

>>> if first or last:

... print("The user has a first or last name")

...

>>>

**The and Operation**

The opposite of or is the and operation, which requires both conditions to be True.

## The While Loop

>>> while True:

... print("looping")

...

looping

looping

looping

looping

That loop will continue forever, we've created an infinite loop. To stop the loop, press Ctrl-C. Infinite loops are one of the potential problems with while loops if we don't use a condition that we can change from within the loop then it will continue forever if initially true.

>>> count = 1

>>> while count <= 4:

... print("looping")

... count += 1

...

looping

looping

looping

looping

>>>

## The For Loop

**The for Loop**

The most common use we have for looping is when we want to execute some code for each item in a sequence. For this type of looping or iteration, we'll use the for loop.

>>> colors = ['blue', 'green', 'red', 'purple']

>>> for color in colors:

... print(color)

...

blue

green

red

purple

If we wanted not to print out certain colors we could utilize the continue or break statements again. Let's say we want to skip the string 'blue' and terminate the loop if we see the string 'red':

>>> colors = ['blue', 'green', 'red', 'purple']

>>> for color in colors:

... if color == 'blue':

... continue

... elif color == 'red':

... break

... print(color)

...

green

A dictionary example:

>>> ages = {'kevin': 59, 'bob': 40, 'kayla': 21}

>>> for key in ages:

... print(key)

...

kevin

bob

kayla

A string example:

>>> for letter in "my\_string":

... print(letter)

...

m

y

\_

s

t

r

i

n

g

>>>

# Encapsulating Code

## Writing Functions

**Function Basics**

We can create functions in Python using the following:

* The def keyword
* The function name - lowercase starting with a letter or underscore (\_)
* Left parenthesis (()
* 0 or more argument names
* Right parenthesis ())
* A colon :
* An indented function body

Here's an example without an argument:

>>> def hello\_world():

... print("Hello, World!")

...

>>> hello\_world()

Hello, World!

>>>

If we want to define an argument, we will put the variable name we want it to have within the parentheses:

>>> def print\_name(name):

... print(f"Name is {name}")

...

>>> print\_name("Keith")

Name is Keit

Neither of these examples has a return value, but we will usually want to have a return value unless the function is our "main" function, or carries out a "side-effect" like printing. If we don't explicitly declare a return value, then the result will be None.

We can declare what we're returning from a function using the return keyword:

>>> def add\_two(num):

... return num + 2

...

>>> result = add\_two(2)

>>> result

4

**Working with Multiple Arguments**

When we have a function that takes multiple arguments, we need to separate them using commas and give them unique names:

>>> def add(num1, num2):

... return num1 + num2

...

>>> result = add(1, 5)

>>> result

6

**Using Keyword Arguments**

Every function call we've made up to this point has used what are known as positional arguments, but if we know the name of the arguments and not necessarily the positions we can call them *all* using keyword arguments like so:

>>> def contact\_card(name, age, car\_model):

... return f"{name} is {age} and drives a {car\_model}"

...

>>> contact\_card("Keith", 29, "Honda Civic")

'Keith is 29 and drives a Honda Civic'

>>> contact\_card(age=29, car\_model="Civic", name="Keith")

'Keith is 29 and drives a Civic'

>>> contact\_card("Keith", car\_model="Civic", age="29")

'Keith is 29 and drives a Civic'

>>> contact\_card(age="29", "Keith", car\_model="Civic")

File "<stdin>", line 1

SyntaxError: positional argument follows keyword argument

Tu peux mélanger les positions tant que tu précises bien l’argument et sa valeur, si tu ne précises pas et que tu inverses l’ordre alors il y aura un problème comme dans le dernier cas ou le nom vient en deuxième position sans précision de l’argument alors que dans le format c’est l’âge qui est attendu à cet endroit.

**Defining Default Arguments**

Along with being able to use keyword arguments when we're calling a function, we're able to define default values for arguments to make them optional when the information is commonly known and the same. To do this, we use the assignment operator (=) when we're defining the argument:

>>> def can\_drive(age, driving\_age=16):

... return age >= driving\_age

...

>>> can\_drive(16)

True

>>> can\_drive(16, driving\_age=18)

False

Default arguments need to go at the end of the arguments list when defining the function, so that positional arguments can still be used to call the function.

# Object-Oriented Programming Basics

Creating Classes

1