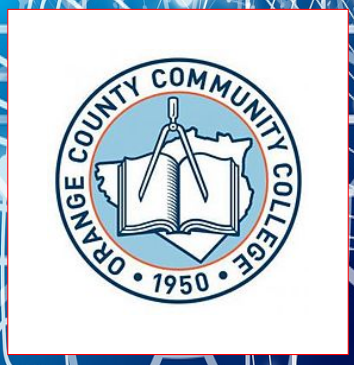


CSC 138

© 2022 All Rights Reserved

Miroslav Krajca

Contents

preface

Some of the material used in these notes comes from :

https://python-textbok.readthedocs.io/en/1.0/index.html
"Licence: ©Copyright 2013, 2014, University of Cape Town and individual contributors. This work is released under the CC BY-SA 4.0 licence. Revision 8e685e710775."

Full License text : https://creativecommons.org/licenses/by-sa/4.0/

Many images and pictures were obtained from: https://pixabay.com/
under the free for commercial use no attribution required.

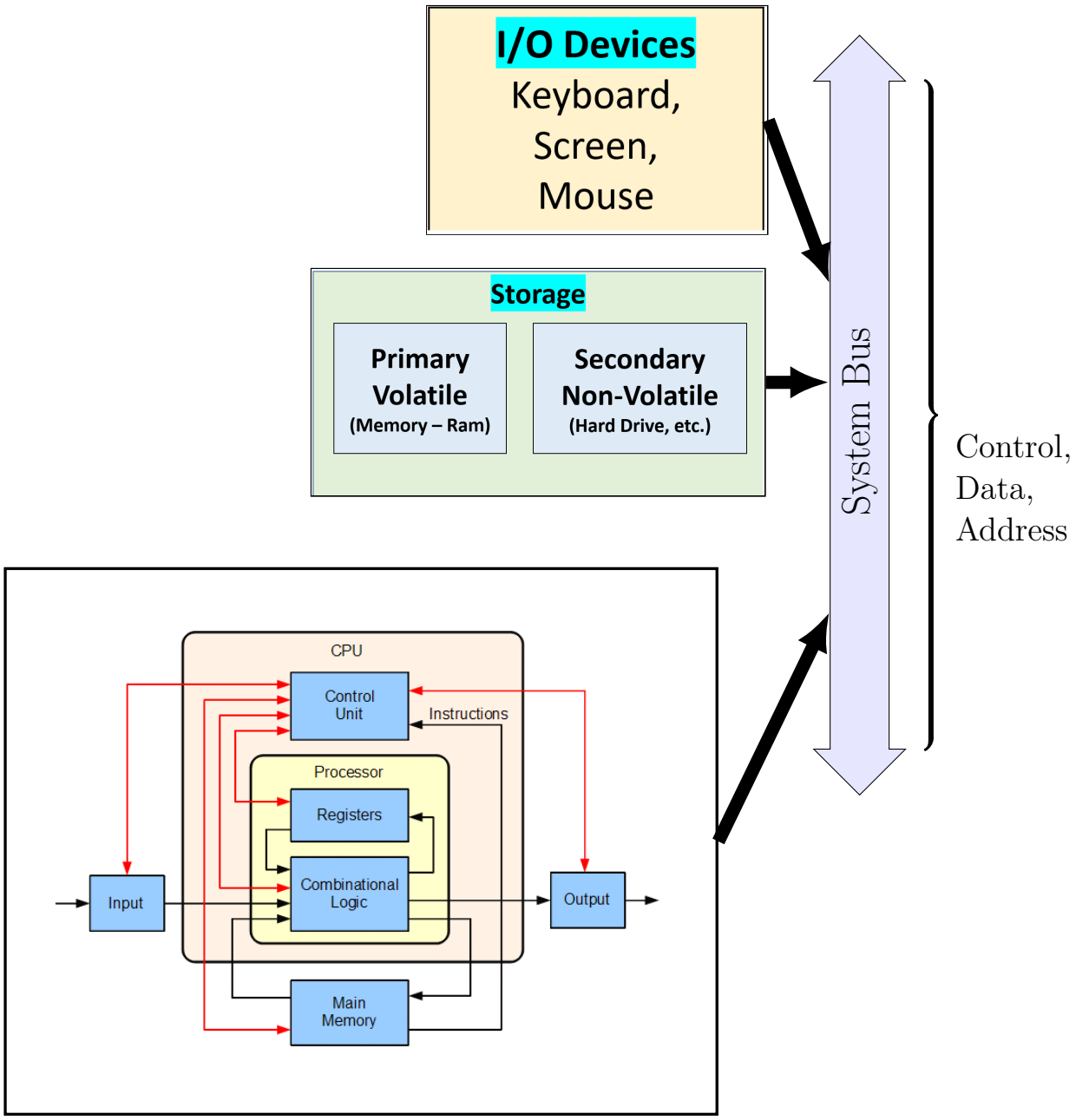
Introduction

1.1 Von Neumann Architecture

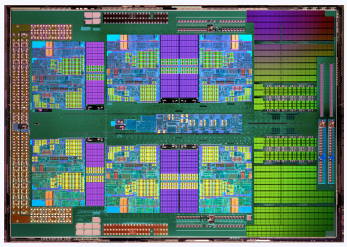
The Von Neumann architecture—also known as the Von Neumann model or Princeton architecture—is a computer architecture based on a 1945 description by John Von Neumann and others in the First Draft of a Report on the EDVAC. That document describes a design architecture for an electronic digital computer with these components:

- A processing unit that contains an arithmetic logic unit and processor registers
- A control unit that contains an instruction register and program counter
- Memory that stores data and instructions
- External mass storage
- Input and output mechanisms

https://en.wikipedia.org/wiki/Von_Neumann_architecture

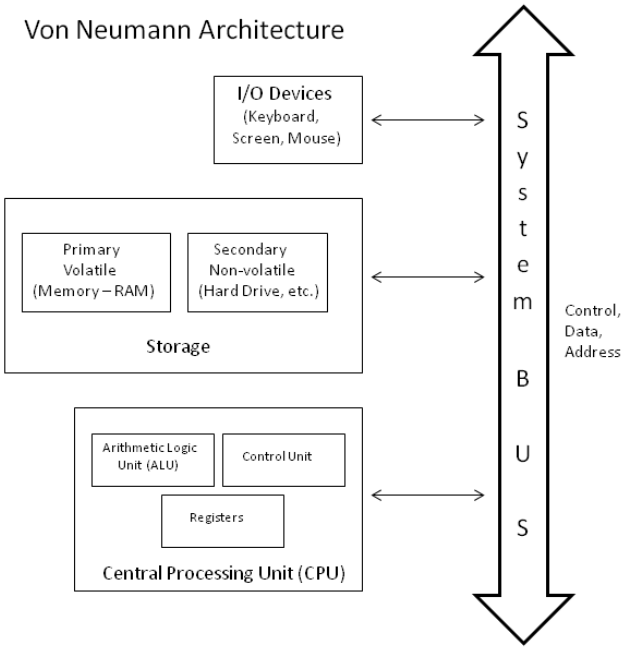

I/O Devices
Keyboard,
Screen,
Mouse
Storage
Primary
Volatile
(Memory – Ram)
Secondary
Non-Volatile
(Hard Drive, etc.)
System Bus
Control,
Data,
Address
CPU
Control
Unit
Instructions
Processor
Registers
Combinational
Logic
Input
Output
Main
Memory

Computer Components



- CPU
  - Control Unit
  - Arithmetic Logic Unit(ALU)
  - Registers
- Storage
  - Primary - memory/RAM

  - Secondary - Hard Disk



1.1.1 Fetch Decode Execute cycle.

instructions cycle

- Fetch
  - The CPU control unit retrieves(fetches) from main the next instruction in the sequence of the program
- Decode
  - The decoding process allows the CPU to determine what instruction is to be performed so that the CPU can tell how many operands it needs to fetch in order to perform the instruction
- Execute
  - Here, the function of the instruction is performed.

1.1.2 Memory architecture

The computer's memory or **RAM** (Random Access Memory) is divided into small units called **bytes**. Each byte is enough to store one text character. Each byte is assigned a unique number called an **Address**. If we want to get the information that is stored in that byte we go to the address and retrieve it. Each Byte is broken down into smaller units called **bits**. Each byte has 8 bits in it. A bit can only have two values, **0** or **1**.

1.1.3 Software

Types of software

- **System Software**
  - **Operating Systems.**
  - **Utility programs.**
  - **Device drivers.**
- **Application software**

1.2 Programming Languages

1.2.1 What is a program?

A program is a set of instructions that tells the computer how to perform a given task.

1.2.2 Difference between program and process.

program

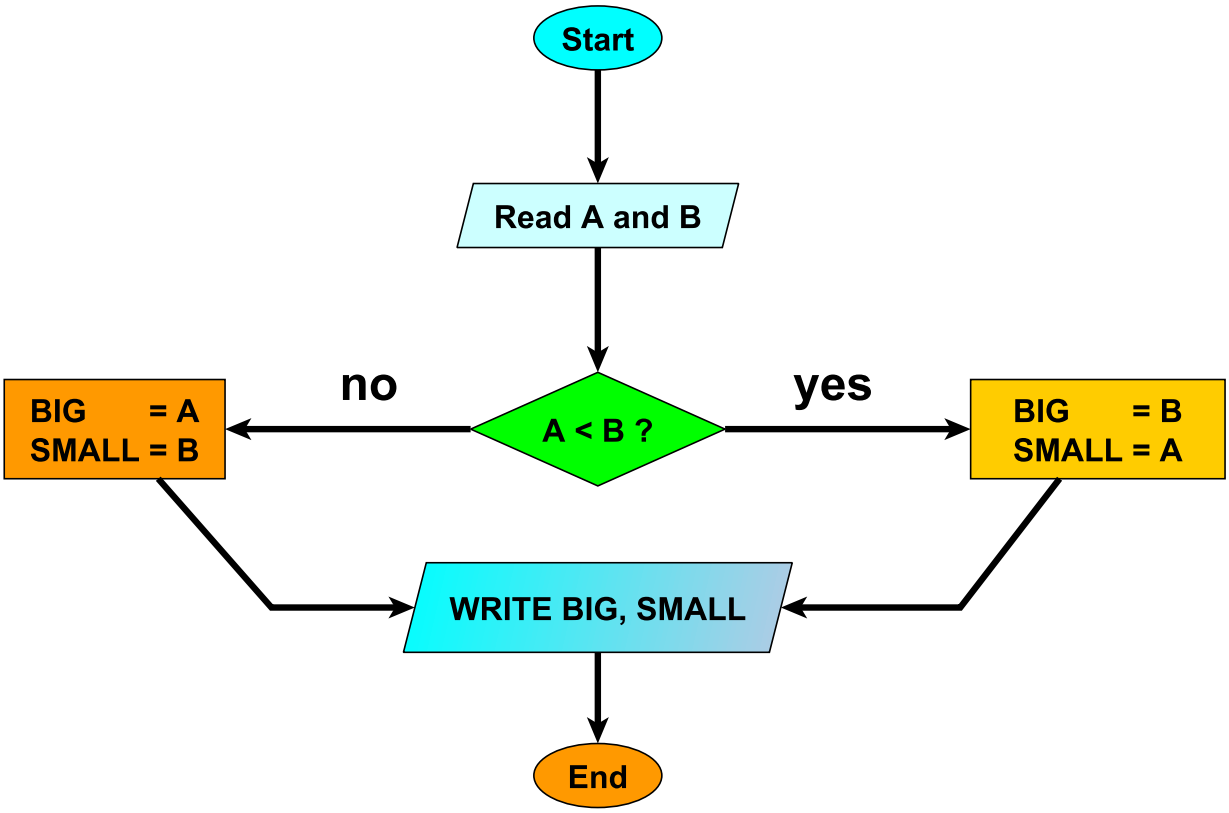
A program is a set of instructions and associated data that resides on the disk and is loaded by the operating system to perform some task. An executable file or a python script file are examples of programs. In order to run a program, the operating system's kernel is first asked to create a new process, which is an environment in which a program executes.

process

A process is a program in execution. A process is an execution environment that consists of instructions, user-data, and system-data segments, as well as lots of other resources such as CPU, memory, address-space, disk and network I/O acquired at runtime. A program can have several copies of it running at the same time but a process necessarily belongs to only one program.

1.2.3 Algorithm

An algorithm is a set of instructions designed to perform a specific task. This can be a simple process, such as multiplying two numbers, or a complex operation, such as playing a compressed video file.



Knuth's definition (1968): " An algorithm is a finite, definite, effective procedure, with some output."

Kowalski (1979): Algorithm = Logic + Control

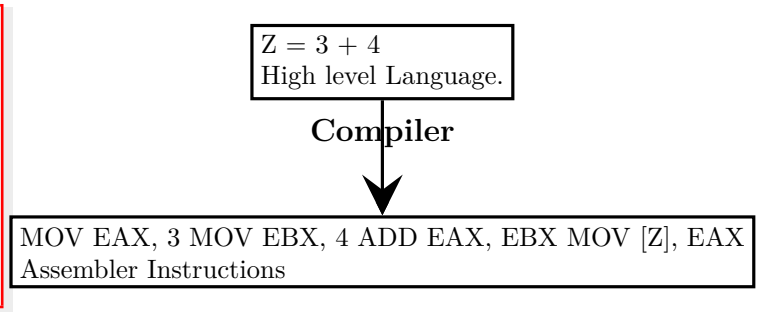
- **Instruction Set / Machine Language**
  - Arithmetic Operations.
  - Memory/Data Operations
  - Control Flow
  - Machine Code

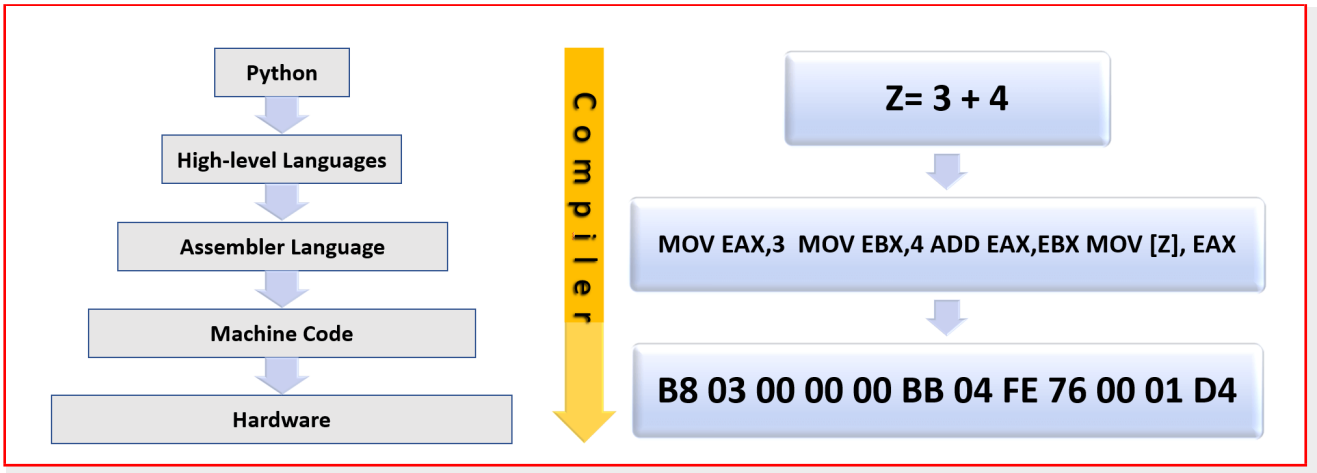
Instructions

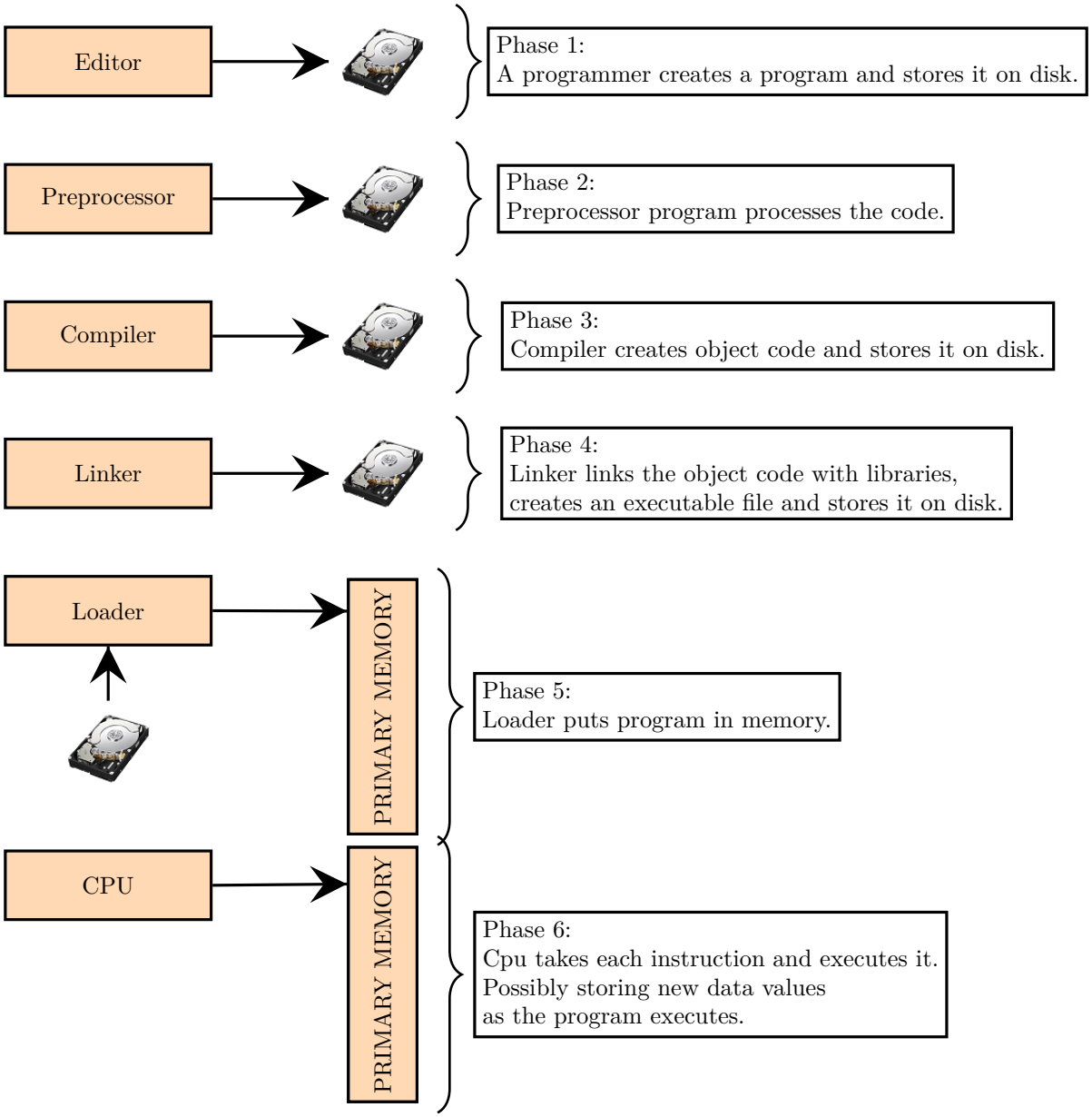
Assembly Language Instruction	Hex Instruction
MOV EAX, 3	B8 03 00 00 00
MOV EBX , 4	BB 04 00 00 00
ADD EAX, EBX	01 D8

Machine Code Instruction	Hex Instruction	Operation
1011 1000 0000 0011 0000 0000 0000 0000 0000 0000	B8 03 00 00 00	Move 3 into Register EAX
1011 1011 0000 0100 0000 0000 0000 0000 0000 0000	BB 04 00 00 00	Move 4 into Register EBX
0000 0001 1101 1000	01 D8	Add Register EBX to EAX

- **High-level Languages**.
  - Source Code
  - Preprocessor
  - Compiler
  - Interpreter
  - Object Code
  - Linker
  - Executable Code





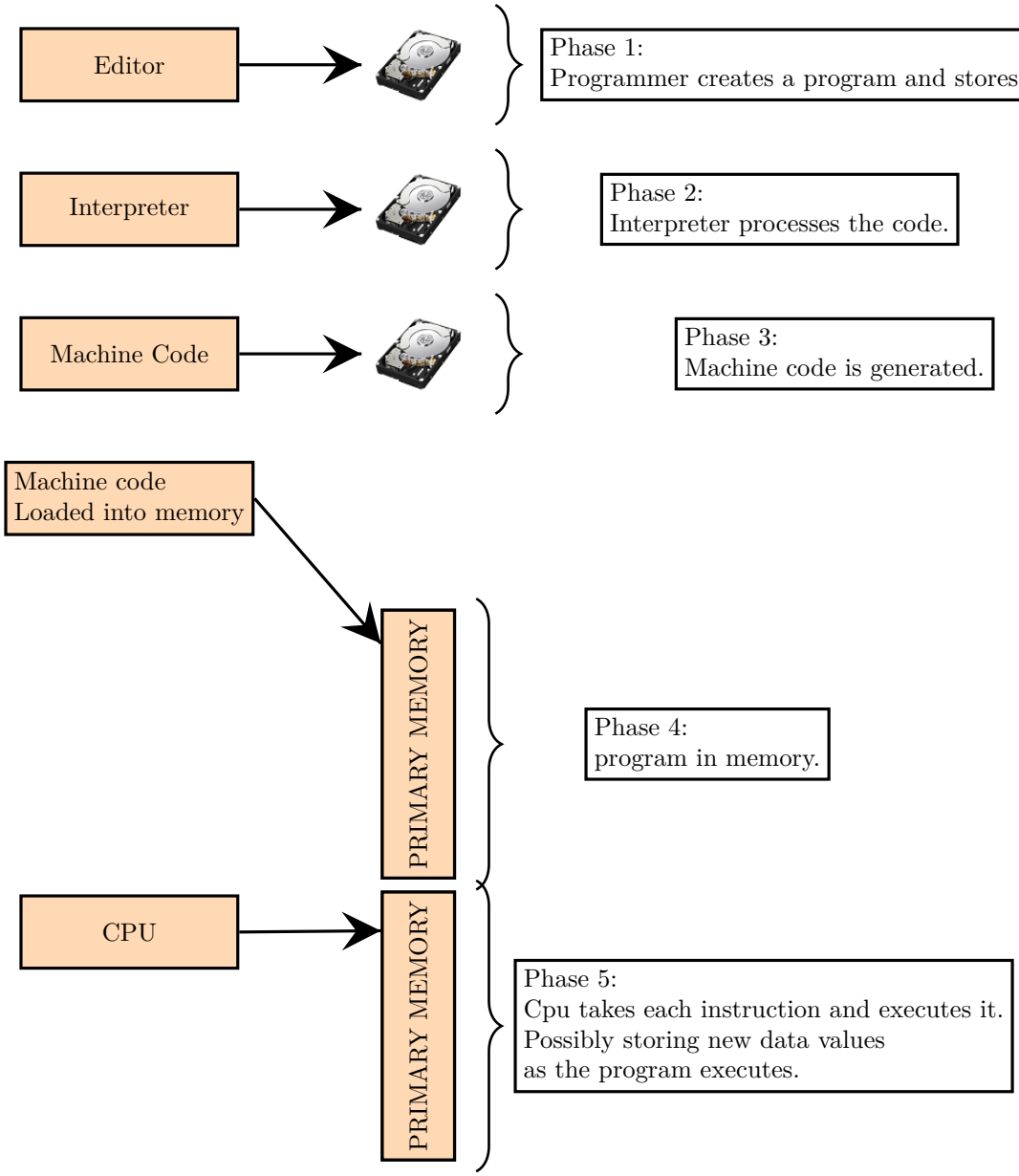


Python is a popular general-purpose, interpreted, extensible programming language. As of this writing, there are over 313,308 different packages that can be used to extend end enhance the abilities of the programmer to create new programs.

These extensions are called **libraries** and the most popular place to get these libraries is **PyPI**

1.2.4 Interpreted language.

Interpreted languages try to solve the issue of portability. The program(source code) is always present and turned into machine language as needed. When the same source code is run on different platforms (Intel, Spark, Arm, MIPS) the machine code is generated as needed for that platform target.



1.2.5 Compiled languages.

Compiled languages **"compile"** or turn the source code into a binary program all at once. The problem is that the program will run only on the machine/operating system for which it was compiled.

1.3 Obtaining Python

https://www.python.org/downloads/
There are two versions of Python that are in use widely.
https://wiki.python.org/moin/Python2orPython3

Since Python3 is newer and will be most widely used in the future these notes will concentrate on Python V3.

Most Python code is used on Linux or other Unix like variants we can use our Linux software package manager to install Python. If you are using Python on windows you can download binaries from the above link.

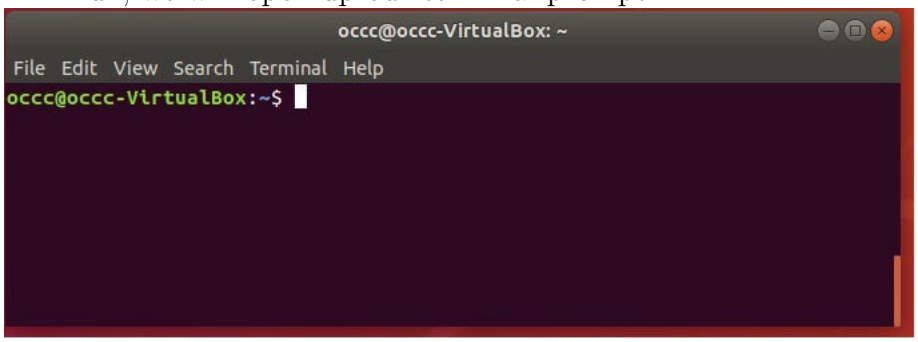
1.3.1 Installing Python on Ubuntu

Listing 1.1: Install Python

```
1 occc@occc-VirtualBox:~$ sudo apt install python3.6
```

1.4 Interacting with Python

One of the first ways we will use python will be to interact directly with the interpreter. In Linux, we will open up our terminal prompt.



We type python3 on the command prompt.

python interact

```
1 root@occc-VirtualBox:~# python3
2 Python 3.8.5 (default, Jul 28 2020, 12:59:40)
3 [GCC 9.3.0] on linux
4 Type "help", "copyright", "credits" or "license" for more ↙
    ↳ information.
5 >>>
```

To see which versions of python we have installed type:

python version

```
occc@occc-VirtualBox:~$ python --version
Python 2.7.15+
root@occc-VirtualBox:~# python3 --version
Python 3.8.5
root@occc-VirtualBox:~#
```

1.5 Writing our first Python code

At the " >>> " prompt type **print("Hello World")** then hit the enter key.

first python code

```
>>> print("Hello World")
Hello World
>>>
```

Since python is interpreted it can look at this one line(statement) evaluate it and immediately give us a result.
In this case, we asked python to print the words **"Hello World"** and it immediately printed them.

1.5.1 single and double quotes

In most programming languages the double-quote `"` indicates that what is included is a literal string or a set of characters that are to be used exactly as typed.
In languages such as Java, C, C++ there is a distinction between double quote and a single quote. `'` single-quote in Java and C++ mean a single character while a double quote means a series of characters (String). As far as language syntax is concerned, there is
no difference in a single or double-quoted string. Both representations can be used interchangeably. However, if either single or double quote is a part of the string itself, then the string must be placed in double or single quotes respectively.

quotes example 1

```
>>> print("Hello World")
Hello World
>>> print('Hello World')
Hello World
>>>
```

quotes example 2

```
>>> print("You're Smart")
You're Smart
>>>
```

quotes example 3

```
>>> print('You're Smart')
File "<stdin>", line 1
print('You're Smart')
^
SyntaxError: invalid syntax
>>>
```

1.6 Arithmetic Operations

add numbers

```
>>> 4 + 3
7
>>>
```

1.6.1 Order of operations.

order of operations

```
>>> 4 + 3 * 2
10
>>>
```

1.6.2 Division

Division. There are two types of division operations.

- **Real or floating-point numbers.**
- **Integer based operations.**

Real numbers operation.

real number division

```
1 >>> 5 / 2
2 2.5
3 >>>
```

Integer(whole number only) based operation.

integer number division

```
1 >>> 5 // 2
2 2
3 >>>
```

The difference is "/" and "//" division operator.

division examples

```
1 >>> 9 / 5
2 1.8
3 >>> 9 // 5
4 1
5 >>> 1 / 3
6 0.3333333333333333
7 >>> 1 // 3
8 0
9 >>>
```

Integer operation always rounds down to the nearest whole number.

division rounding

```
>>> ( 3 + 2) * 5
25
>>> ( 3 + 2) * 5 / 4
6.25
>>>
```

1.6.3 Exponents

**

exponents

```
x = 5
y = 2
z = x**y    # 5 to the power of 2  5^2
print(z)
print(2**3) #used in print statement with literal numbers
```

Output

```
25
8
```

1.7 Quitting the python interpreter

To leave the interpreter shell type

quit interpreter

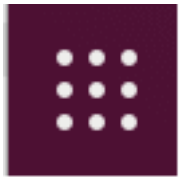
```
>>> quit()
```

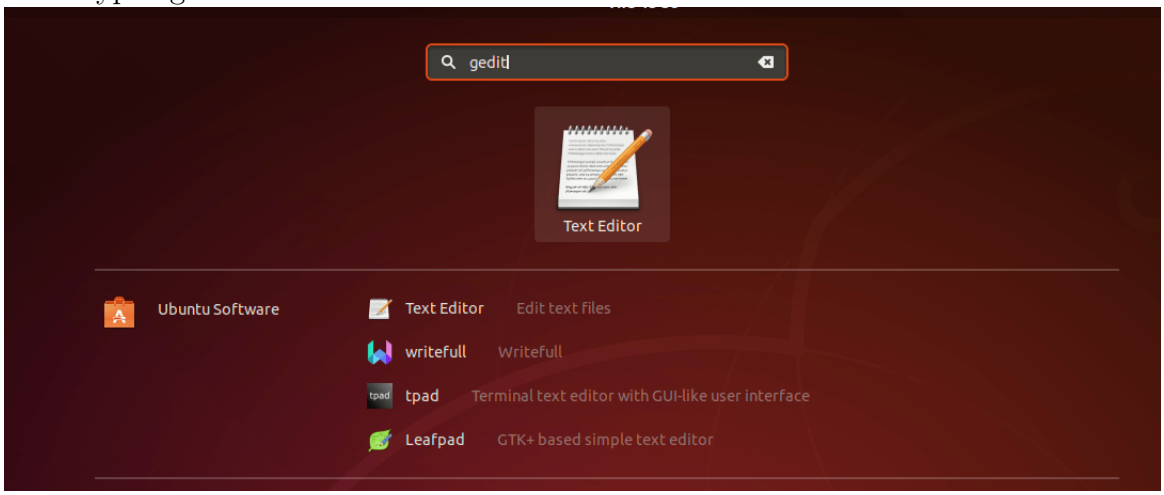
Creating Python scripts

On your command prompt in the bash shell. Type **"mkdir labs"** hit enter. Then type **"cd labs"** and hit enter.

```
occc@occc-VirtualBox:~$ mkdir labs
occc@occc-VirtualBox:~$ cd labs
occc@occc-VirtualBox:~/labs$
```

In your Ubuntu desktop click the gedit icon. 

If you do not see the icon click the bottom icon "applications:" 
Then type "gedit" in the search field.

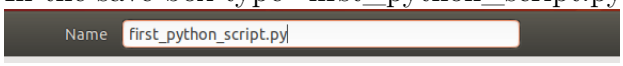


Click the Text editor icon.
In your editor type:

```
1   print("Hello World")
```

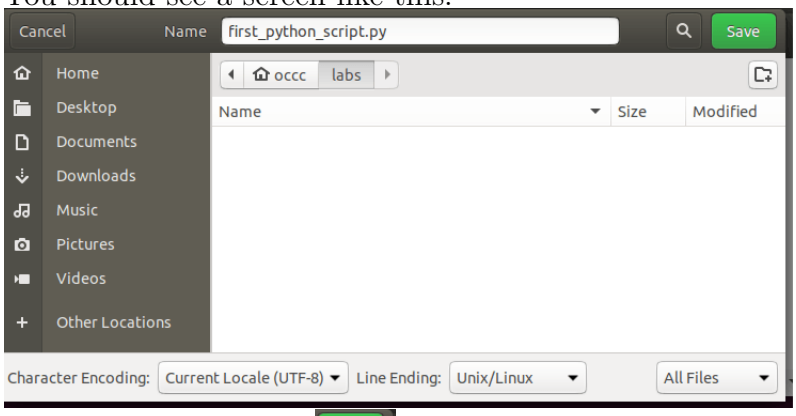
Click the save icon. 

In the save box type ”first_python_script.py”



Double click the ”labs directory.

You should see a screen like this.



Click the save button. 

Go back to our command prompt.

```
occc@occc-VirtualBox:~$ mkdir labs
occc@occc-VirtualBox:~$ cd labs
occc@occc-VirtualBox:~/labs$
```

Type **”ls”** (directory listing) to make sure our script is there.

```
1  occc@occc-VirtualBox:~/labs$ ls
2 first_python_script.py
3 occc@occc-VirtualBox:~/labs$
```

Run the script by typing ”python3 first_python_script.py”

```
1  occc@occc-VirtualBox:~/labs$ python3 first_python_script.py
2  Hello World
3  occc@occc-VirtualBox:~/labs$
```

With the above command, we invoked the python interpreter and told it to use the file as input and process it.
Just like before the ”Hello World” was printed out and shown on the screen.

On the command prompt type **”which python3”**

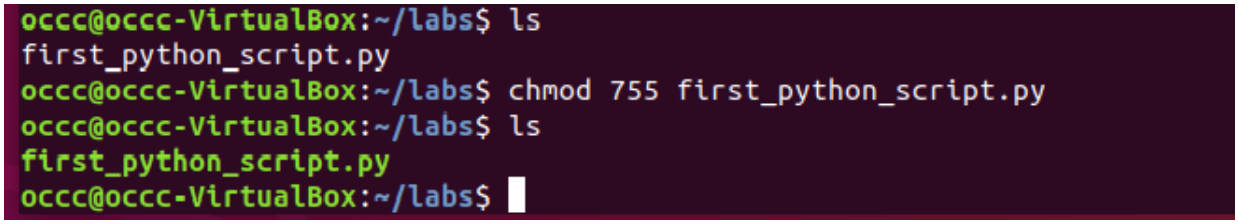
```
1   occc@occc-VirtualBox:~/labs$ which python3
2   /usr/bin/python3
3   occc@occc-VirtualBox:~/labs$
```

This will tell us the location of the python3 interpreter. Go back to your gedit text editor and change the script to look like the following:

```
1   #!/usr/bin/python3
2   print("Hello World")
```

Save the file and go back to the bash shell.
The first line consists of what is known as a ”haspling” or ”shebang” #!
That is then followed by the path to the python interpreter. /usr/bin/python3
Type:

```
occc@occc-VirtualBox:~/labs$ ls
first_python_script.py
occc@occc-VirtualBox:~/labs$ chmod 755 first_python_script.py
occc@occc-VirtualBox:~/labs$ ls
first_python_script.py
occc@occc-VirtualBox:~/labs$
```



Notice that the color of the script changed from white to green. This made the script executable and can be run without the python3 program in front of it.

type **”./first_python_script.py”**

```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
Hello World
occc@occc-VirtualBox:~/labs$
```

The **”#!”** is called a **Hashpling** or **shebang**. These characters, when put on the first line of our script, tells the command prompt to invoke the python3 interpreter and pass on the rest of the file as input to it

2.1 Comments

2.1.1 Python Comments

Comments are very important while writing a program. It describes what’s going on inside a program so that a person looking at the source code does not have a hard time figuring it out. You might forget the key details of the program you just wrote in a month time. So taking the time to explain these concepts in the form of comments is always fruitful.

In Python, we use the hash `#` symbol to start writing a comment.

It extends up to the newline character. Comments are for programmers for a better understanding of a program. Python Interpreter ignores the comment.

Listing 2.1: comments single line

```
#This is a comment
#print out Hello
print('Hello')
```

2.1.2 Multi-line comments

If we have comments that extend multiple lines, one way of doing it is to use hash (#) at the beginning of each line. For example:

```
#This is a long comment
#and it extends
#to multiple lines
```

Another way of doing this is to use triple quotes, either `'''` or `"""`.

These triple quotes are generally used for multi-line strings. But they can be used as a multi-line comment as well. Unless they are not docstrings, they do not generate any extra code.

```
"""This is also
a perfect example of
multi-line comments"""
```

2.1.3 Some basic Unix commands which will be helpful.

ls -l — lists your files in 'long format', which contains lots of useful information, e.g. the exact size of the file, who owns the file and who has the right to look at it, and when it was last modified.
cp filename1 filename2 — copies a file
mv filename1 filename2 — moves a file (i.e. gives it a different name, or moves it into a different directory
chmod options filename — lets you change the read, write, and execute permissions on your files. The default is that only you can look at them and change them, but you may sometimes want to change these permissions.
mkdir dirname — make a new directory
cd dirname — change directory.
You basically 'go' to another directory, and you will see the files in that directory when you do 'ls'. You always start out in your 'home directory', and you can get back there by typing 'cd' without arguments. 'cd ..' will get you one level up from your current position. You don't have to walk along step by step - you can make big leaps or avoid walking around by specifying pathnames.

2.2 Further reading

We will see many examples of Python's built-in functions and types and modules in the standard library – but this document is only a summary and not an exhaustive list of all the features of the language. As you work on the exercises in this module, you should use the **official Python documentation** as a reference.

For example, each module in the standard library has a section in the documentation which describes its application programming interface, or API – the functionality which is available to you when you use the module in your code. By looking up the API you will be able to see what functions the module provides, what input they require, what output they return, and so on. The documentation often includes helpful examples that show you how the module is meant to be used.

The documentation is available on the web, but you can also install it on your computer – you can either download a copy of the documentation files in HTML format so that you can browse them locally, or use a tool like pydoc, which prints out the documentation on the command-line:

Listing 2.2: pydoc

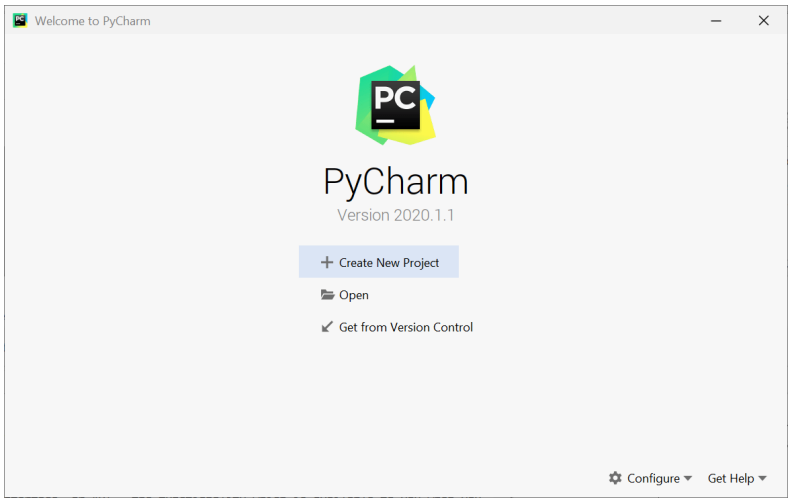
```
pydoc print
```

```
The "print" statement
*********************

print_stmt ::= "print" ([expression ("," expression)* [","]]
| ">>" expression [("," expression)+ [","]])

"print" evaluates each expression in turn and writes the resulting
object to standard output (see below).  If an object is not a string,
it is first converted to a string using the rules for string
conversions.  The (resulting or original) string is then written.  A
space is written before each object is (converted and) written unless
the output system believes it is positioned at the beginning of a
line.  This is the case (1) when no characters have yet been written
to standard output, (2) when the last character is written to standard
output is a whitespace character except "' '", or (3) when the last
write operation on standard output was not a "print" statement. (In
some cases it may be functional to write an empty string to standard
output for this reason.)

Note: Objects which act like file objects but which are not the
built-in file objects often do not properly emulate this aspect of
the file object's behavior, so it is best not to rely on this.
```

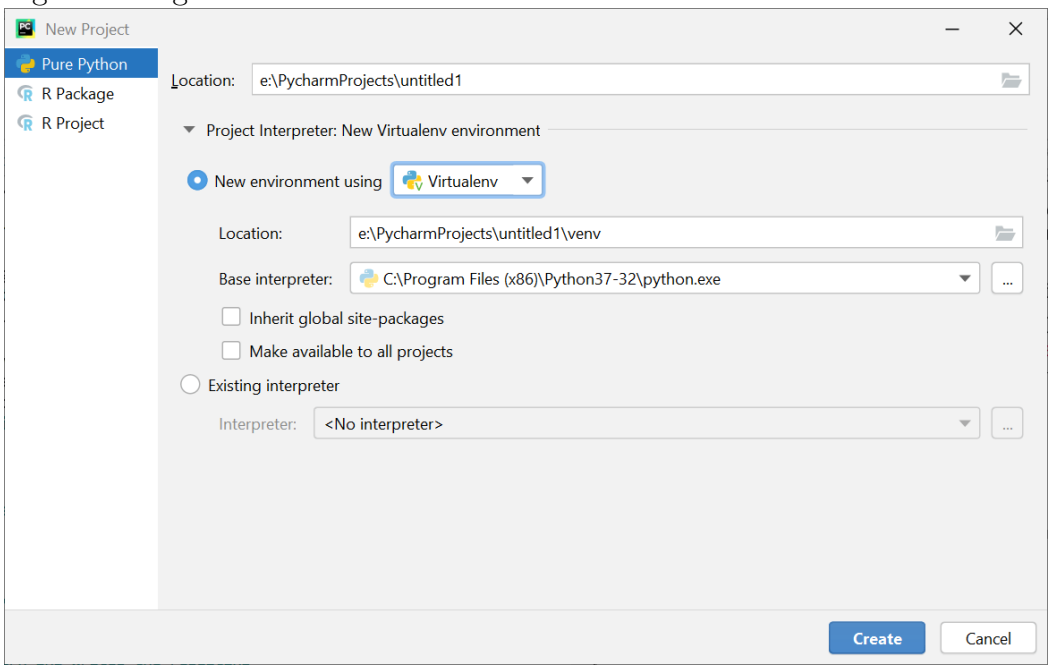
2.3 Using IDE

An integrated development environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of at least a source code editor, build automation tools and a debugger.

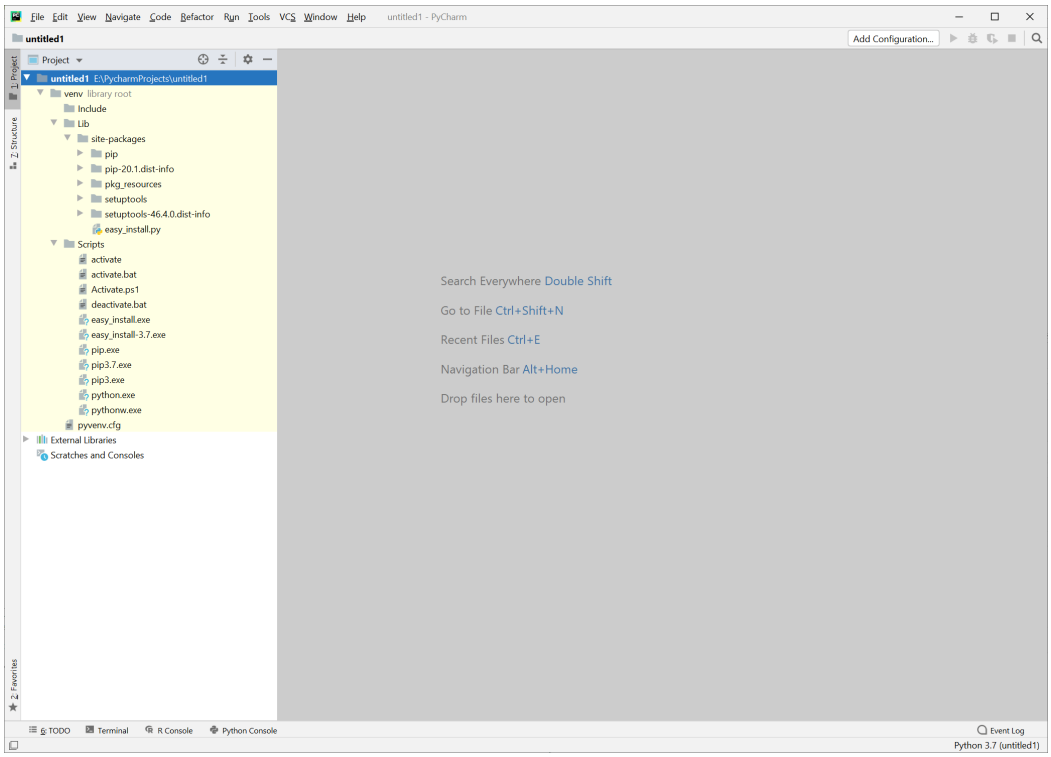
https://www.jetbrains.com/pycharm/download/#section=windows
Using PyCharm.



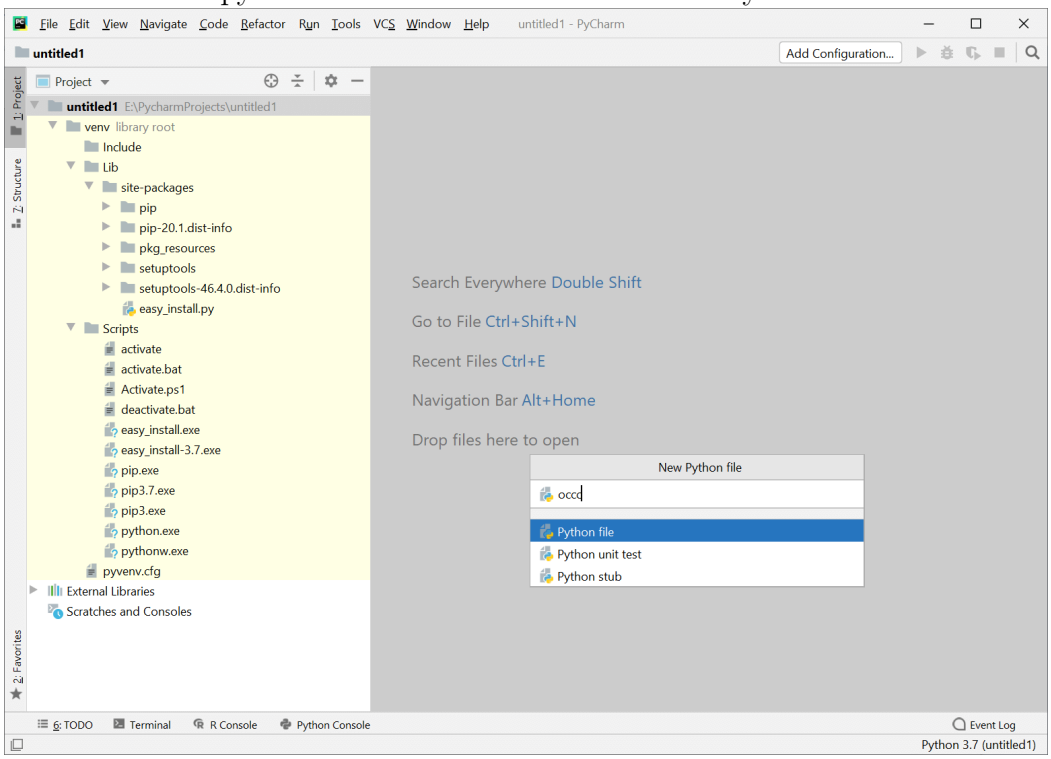
Most IDE's are based on projects. Each application is a project that has all the necessary files organized together.



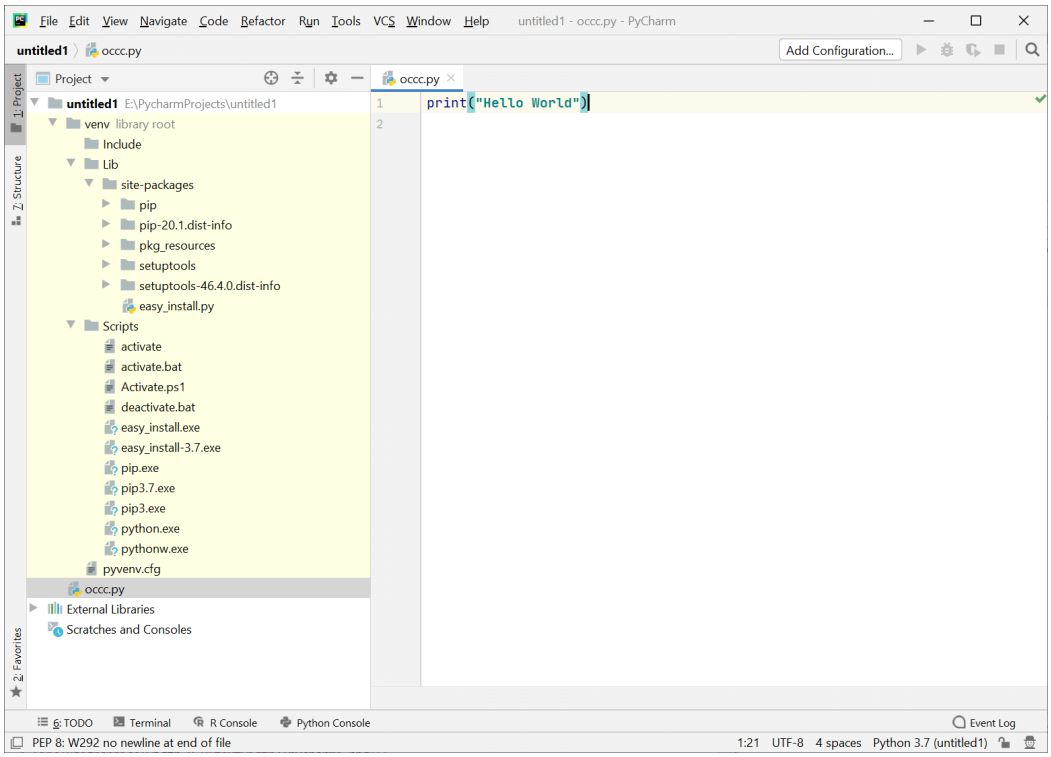
• Location. This is where we will store our files.
• Environment. since each project could use different versions of libraries and even a different version of the python interpreter we create a virtualized environment for our project.



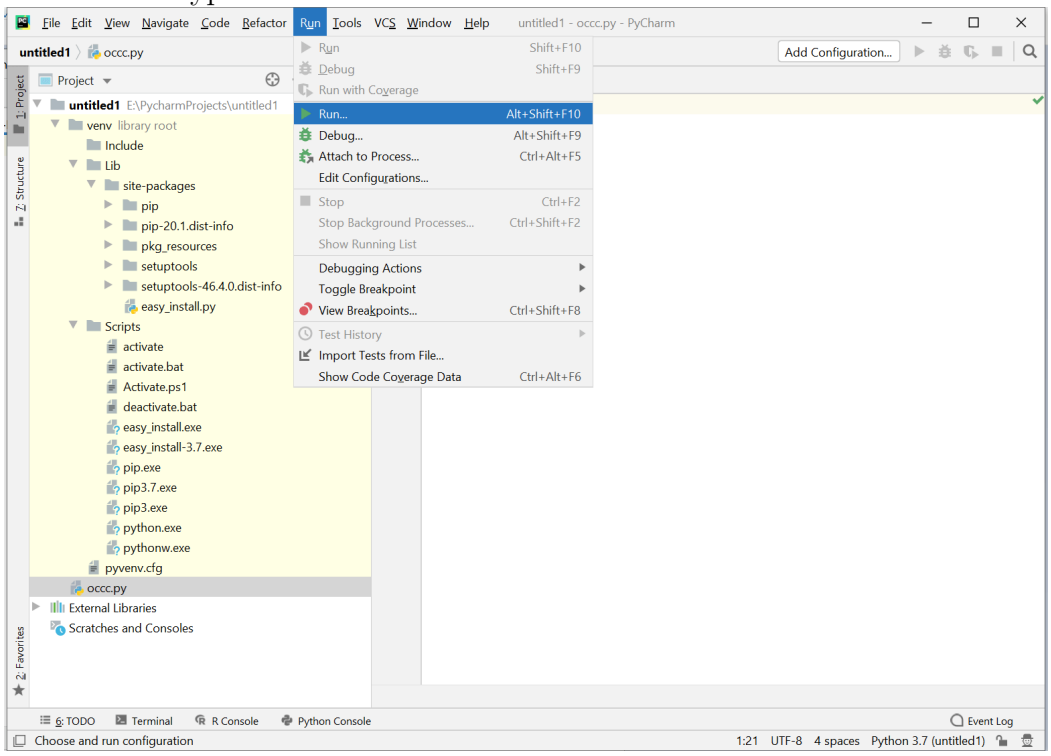
• Create a new python file for our code. File-> new Python File



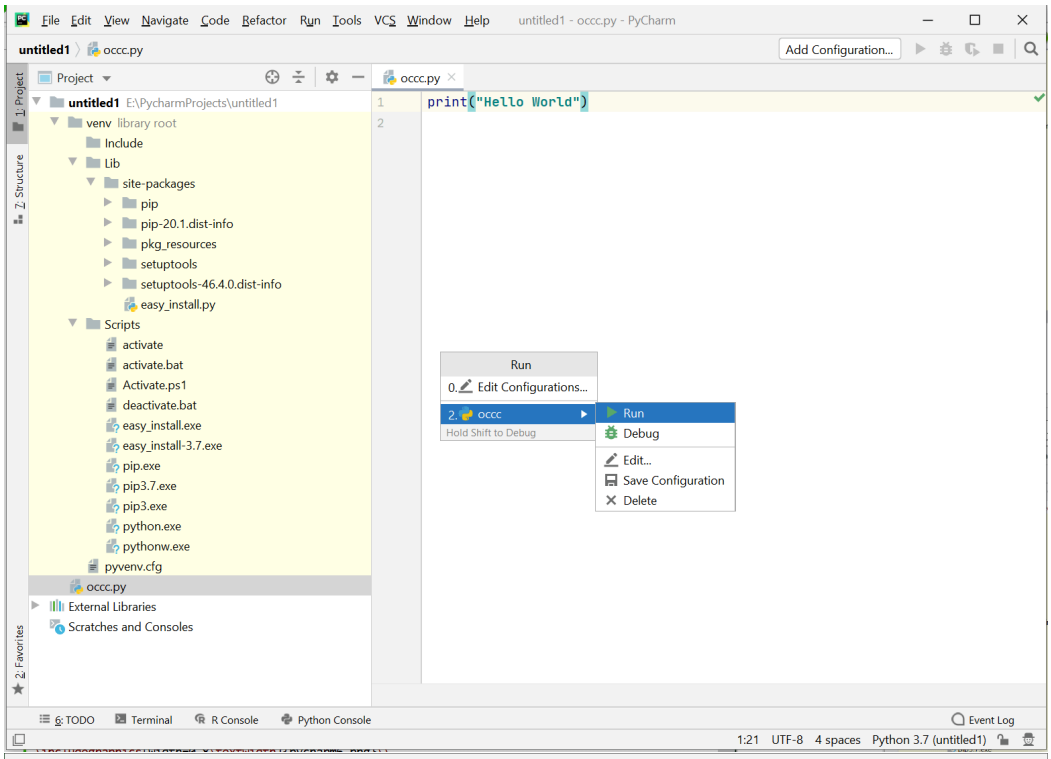
Type in a new file name. and hit enter.

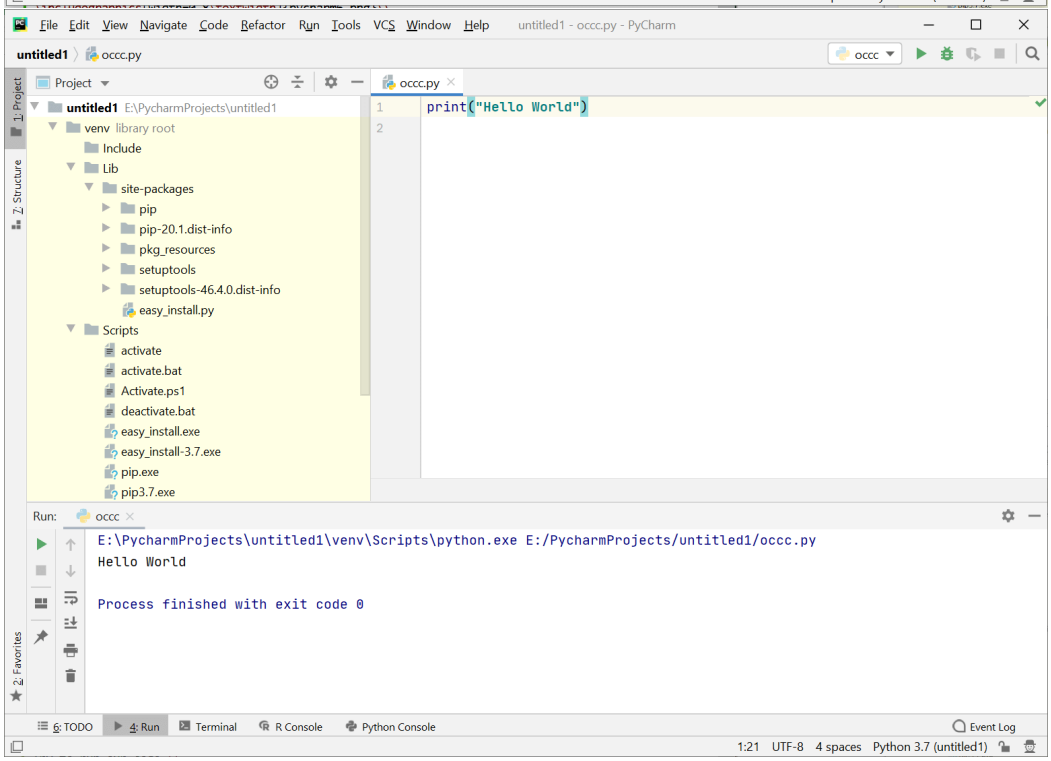


We can now type in our code.



Try to run our code.

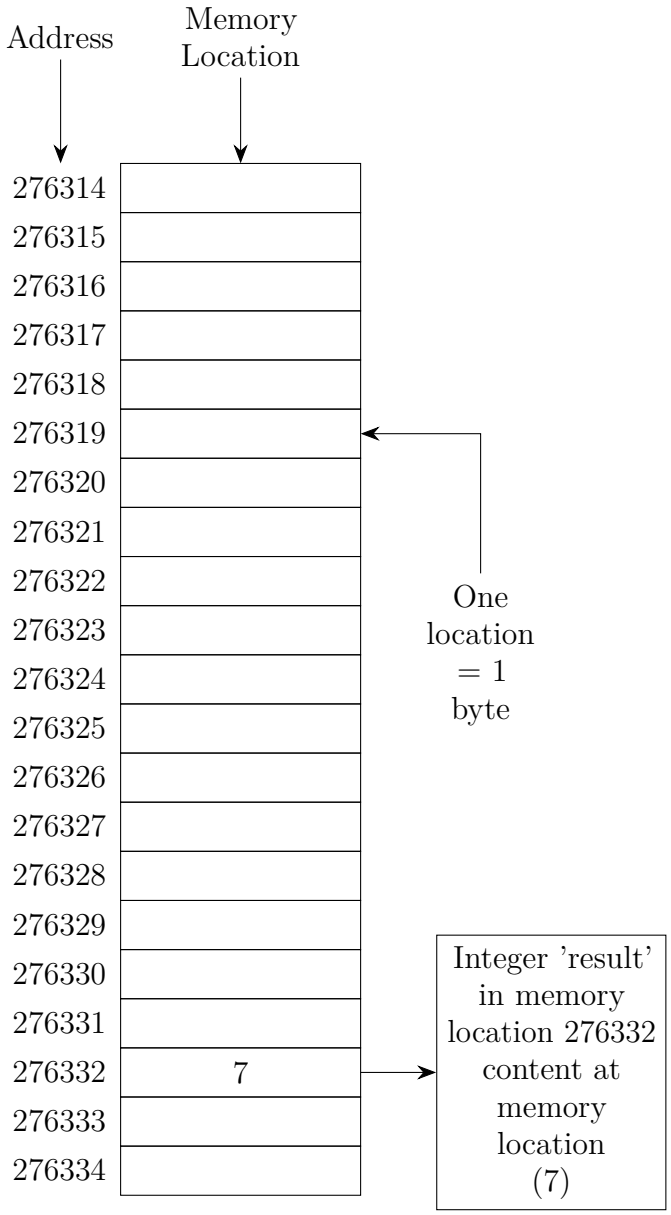

File Edit View Navigate Code Refactor Run Tools VCS Window Help
untitled1 - occc.py - PyCharm
untitled1 〉 occc.py
Add Configuration...
Project
occc.py
untitled1 E:\PycharmProjects\untitled1
venv library root
Include
Lib
site-packages
pip
pip-20.1.dist-info
pkg_resources
setuptools
setuptools-46.4.0.dist-info
easy_install.py
Scripts
activate
activate.bat
Activate.ps1
deactivate.bat
easy_install.exe
easy_install-3.7.exe
pip.exe
pip3.7.exe
pip3.exe
python.exe
pythonw.exe
pyvenv.cfg
occc.py
External Libraries
Scratches and Consoles
print("Hello World")
Run
0. Edit Configurations...
2. occc
Hold Shift to Debug
Run
Debug
Edit...
Save Configuration
Delete
6: TODO
Terminal
R Console
Python Console
Event Log
1:21 UTF-8 4 spaces Python 3.7 (untitled1)


File Edit View Navigate Code Refactor Run Tools VCS Window Help
untitled1 - occc.py - PyCharm
untitled1 〉 occc.py
occc
Project
occc.py
untitled1 E:\PycharmProjects\untitled1
venv library root
Include
Lib
site-packages
pip
pip-20.1.dist-info
pkg_resources
setuptools
setuptools-46.4.0.dist-info
easy_install.py
Scripts
activate
activate.bat
Activate.ps1
deactivate.bat
easy_install.exe
easy_install-3.7.exe
pip.exe
pip3.7.exe
print("Hello World")
Run:
occc
E:\PycharmProjects\untitled1\venv\Scripts\python.exe E:/PycharmProjects/untitled1/occc.py
Hello World
Process finished with exit code 0
6: TODO
4: Run
Terminal
R Console
Python Console
Event Log
1:21 UTF-8 4 spaces Python 3.7 (untitled1)

In programming, a variable is a value that can change, depending on conditions or on information passed to the program. Variables are also called **named memory location.**

```
occc@occc-VirtualBox:~/labs$ python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 3 + 4
7
>>>
```

In the above example, we got an answer of 7. However, this answer is not stored anywhere. If we want to use this result, later on, we have to store the answer. To do that we have to tell the computer to allocate space in our memory where we can place the result in.



If we want to store the result we have to find a memory address that is not used. The interpreter will ask the operating system for a memory address that we can use. This will give us a storage location. If we want to use this memory location we have to tell the computer the memory address.

Since memory addresses are large it would be impossible for us to remember what goes at which memory address. For convenience, we give this memory location a name.

If we wanted to store the result we could store it in a variable called result. **result = 3 + 4.** Every time we want to access this memory location we will reference it by its name.

Hence the definition of **named memory location**.

Modifying our first script we get the following:

python interact

```
root@occc-VirtualBox:~# python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more ↙
   ↳  information.
>>>
```

variable example 1

```
#!/usr/bin/python3
if __name__=='__main__':
    print("Hello World")
    result = 3 + 4
    print(result)
```

Output

```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
Hello World
7
occc@occc-VirtualBox:~/labs$
```

Notice that we did not include the quotes around the result variable. This is how Python knows that you want to print the content of the variable instead of the words "result"

The **if __name__=='__main__':** allows the Python interpreter to determine if this code is executed as the main script or if its part of some other script. This will be covered later on in the course under the importation of modules.It is not necessary to have it, but it is a good practice to include it. Note all of our code has to be indented on the same column after the above line

Including the quotes we get a different outcome.

variable example 2

```
#!/usr/bin/python3
if __name__=='__main__':
    print("Hello World")
    result = 3 + 4
    print(result)
    print("result")
```

Output

```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
Hello World
7
result
occc@occc-VirtualBox:~/labs$
```

A variable is similar to the memory functionality found in most calculators, in that it holds one value which can be retrieved many times, and that storing a new value erases the old. A variable differs from a calculator's memory in that one can have many variables storing different values, and that each variable is referred to by name.

3.1 Variable assignments

Variables get their content(data) from the assignment operator. =
Any data(information) on the right hand side of the = is stored in the left hand side of the = operator.
If the right hand side is an expression (such as an arithmetic expression), it will be evaluated before the assignment occurs.

Assigning an initial value to variable is called **initializing** the variable. In some languages defining a variable can be done in a separate step before the first value assignment. It is thus possible in those languages for a variable to be defined but not have a value – which could lead to errors or unexpected behavior if we try to use the value before it has been assigned. In Python a variable is defined and assigned a value in a single step, so we will almost never encounter situations like this.

variable assigment

```
1
2       #!/usr/bin/python3
3       x = 123
4       print('Assigned 123 to x')
5       print(x)
6       x = x + 5000
7       print('Added 5000 to x')
8       print(x)
9       pi = 3.14
10      print('pi is equal to 3.14')
11      print(pi)
12      x = pi
13      print('assigned pi to x')
14      print(x)
15      x = x / 2.0
16      print('assigned x = x / 2.0')
17      print(x)
18      occc = "Orange"
19      print('assigned occc = Orange')
20      print(occc)
21      print('try to combine float type and str type and assign ↙
           ↳  to x')
22      x = pi + occc
```

Output

```
Assigned 123 to x
123
Added 5000 to x
5123
pi is equal to 3.14
3.14
assigned pi to x
3.14
assigned x = x / 2.0
1.57
assigned occc = Orange
Orange
try to combine float type and str type and assign to x
Traceback (most recent call last):
File "./first_python_script.py", line 22, in <module>
x = pi + occc
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

We will get an error if we try to add or combine unsupported data types.

3.2 Variable naming

You can use any letter, the special characters "_" and every number provided you do not start with it.
White spaces and signs with special meanings in Python, as "+" and "-" are not allowed.
Variable names are case sensitive.
We should always name variables so that we can understand what they do and what they are for.
Example:

variable naming example 1

```
1
2  #!/usr/bin/python3
3  if __name__=='__main__':
4      rectangle_width = 20
5      rectangle_height = 15
6      college_name = "occc"
7      # bad naming
8      rw = 20
9      rh = 15
10     cn = "occc"
```

From the above code, we can infer what the first three variables do. The last three can become confusing when we have dozens if not hundreds of variables in our code.

We cannot use special symbols like `!, @, #, $, % ,` etc. in our identifier.

No identifier can have the same name as one of the Python keywords:
and, as, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield

Reserved keywords

Python3 Reserved Keywords					
False	None	and	True	as	assert
break	class	continue	def	del	elif
else	except	finally	for	from	global
if	import	in	is	lambda	nonlocal
not	or	pass	raise	return	try
while	with	yield			

If we use any of the reserved keywords or not allowed special characters in our variable naming we will get an error.

```
1 >>> from = 20
```

```
  File "<stdin>", line 1
    from = 20
         ^
SyntaxError: invalid syntax
>>> occc@ = 3.14
  File "<stdin>", line 1
    occc@ = 3.14
         ^
SyntaxError: invalid syntax
>>>
```

3.3 Different types of variables (data types)

3.3.1 Integers

Integers are positive or negative whole numbers with no decimal point.

```
>>> print(123123123123123123123123123123123123123123123123)
123123123123123123123123123123123123123123123123
>>>
```

In Python 3, there is effectively no limit to how long an integer value can be. It is only constrained by the amount of memory your system has.
The following strings can be pre-pended to an integer value to indicate a base other than 10:

Number base

Prefix	Interpretation	Base
0b (zero + lowercase letter 'b') 0B (zero + uppercase letter 'B')	Binary	2
0o (zero + lowercase letter 'o') 0O (zero + uppercase letter 'O')	Octal	8
0x (zero + lowercase letter 'x') 0X (zero + uppercase letter 'X')	Hexadecimal	16

```
>>> print(15)
15
>>> print(0x15)
21
>>> print(0b1111)
15
>>>
```

```
>>> type(15)
<class 'int'>
>>> type(0x15)
<class 'int'>
>>> type(0b1111)
<class 'int'>
>>>
```

When you type a large integer, you might be tempted to use commas between groups of three

digits, as in 1,000,000. This is not a legal integer in Python, but it is legal:

no comma use in integers

```
1
2 #!/usr/bin/python3
3 if __name__ == '__main__':
4     print(1,000,000)
5     large_int = 1,000,000
6     print(large_int)
```

Output

```
1 0 0
(1, 0, 0)
```

Well, that's not what we expected at all! Python interprets 1,000,000 as a comma-separated sequence of integers, which it prints with spaces between.

This is an example of a semantic error: the code runs without producing an error message, but it doesn't do the "right" thing.

3.3.2 Floating point numbers (real)

The float type in Python designates a floating-point number. Float values are specified with a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation:

```
1     >>> 3.14
2     3.14
3     >>> 314e-2
4     3.14
5     >>> 314e2
6     31400.0
7     >>>
```

```
1     >>> type(3.14)
2     <class 'float'>
3     >>> type(314e-2)
4     <class 'float'>
5     >>> type(314e2)
6     <class 'float'>
7     >>>
```

Almost all platforms represent Python float values as 64-bit "double-precision" values, according to the IEEE 754 standard. In that case, the maximum value a floating-point number can have is approximately $1.8 * 10^{308}$. Python will indicate a number greater than that by the string inf:

```
1 >>> 1.79e308
2 1.79e+308
3 >>> 1.8e308
4 inf
5 >>> 5e-324
6 5e-324
7 >>> 1e-325
8 0.0
9 >>>
```

The closest a nonzero number can be to zero is approximately $5 * 10^{-324}$ Anything closer to zero than that is effectively zero.

float.as_integer_ratio() : Returns a pair of integers whose ratio is exactly equal to the actual float having a positive denominator.In the case of infinite, it raises overflow error and value errors on Not a number (NaNs).

as integer ratio

```
# working of float.as_integer_ratio()
if __name__=='__main__':
    a = 3.5
    b = a.as_integer_ratio()
    print(b[0], "/", b[1])
    c = 3.14
    d = c.as_integer_ratio()
    print("----------------------------")
    print(d[0], "/", d[1])
```

Output

```
7 / 2
----------------------------
7070651414971679 / 2251799813685248
```

3.3.3 Complex Numbers

Complex numbers are specified as <real part>+<imaginary part>j. For example:

```
>>> 2 + 3j
(2+3j)
>>> type(2+3j)
<class 'complex'>
>>>
```

complex real imaginary

```

# importing "cmath" for complex number operations
import cmath

# Initializing real numbers
x = 9
y = 45

# converting x and y into complex number
z = complex(x,y);

# printing real and imaginary part of complex number
print ("The real part of complex number is : ",end="")
print (z.real)

print ("The imaginary part of complex number is : ",end="")
print (z.imag)
```

Output

```
The real part of complex number is : 9.0
The imaginary part of complex number is : 45
```

3.3.4 Strings

Strings are sequences of character data. The string type in Python is called str. String literals may be delimited using either single or double-quotes. All the characters between the opening delimiter and matching closing delimiter are part of the string:

```
>>> print("I am a string.")
I am a string.
>>> type("I am a string.")
<class 'str'>
>>> print('Me too')
Me too
>>> type('Me too')
<class 'str'>
>>>
```

A string in Python can contain as many characters as you wish. The only limit is your machine's memory resources. A string can also be empty:

```
>>> ''
''
>>> ""
''
```

```
>>>
```

Including quotes in a string.

```
>>> print('This string contains a single quote (') character.')
  File "<stdin>", line 1
    print('This string contains a single quote (') character.')
                                                   ^
SyntaxError: invalid syntax
>>>
```

If you want to include either type of quote character within the string, the simplest way is to delimit the string with the other type. If a string is to contain a single-quote, delimit it with double quotes and vice versa:

```
>>> print("This string contains a single quote (') character.")
This string contains a single quote (') character.
>>> print('This string contains a double quote (") character.')
This string contains a double quote (") character.
>>>
```

Specifying a **backslash** in front of the quote character in a string **"escapes"** it and causes Python to suppress its usual special meaning. It is then interpreted simply as a literal single-quote character:

```
>>> print('This string contains a single quote (') character.')
  File "<stdin>", line 1
    print('This string contains a single quote (') character.')
                                                   ^
SyntaxError: invalid syntax
>>> print('This string contains a single quote (\') character.')
This string contains a single quote (') character.
>>> print("This string contains a single quote (\") character.")
This string contains a single quote (") character.
>>>
```

Escape sequence

Escape sequence	Usual interpretation of Character(s) After backslash	"Escaped" Interpretation
\'	Terminates string with single quote opening delimiter	Literal single quote (') character
\"	Terminates string with double quote opening delimiter	Literal double quote (") character
\\	Introduces escape sequence	Literal backslash (\) character

Ordinarily, a newline character terminates line input. So pressing 

in the middle of a string will cause Python to think it is incomplete:

```
>>> print("orange
  File "<stdin>", line 1
    print("orange
                 ^
SyntaxError: EOL while scanning string literal
>>> print("orange \
```

```
... county \
... community \
... college")
orange county community college
>>>
```

```
>>> print('four\\two')
four\two
>>>
```

List of escape sequences

Escape Sequence	"Escaped" Interpretation	Example
\a	ASCII Bell (BEL) character	»>print("bell \a") bell »>
\b	ASCII Backspace (BS) character	»>print("Hello World") Hello World »>print("Hello \b World") Hello World »>
\f	ASCII Formfeed (FF) character	»>print("Hello \f World") Hello World »>
\n	ASCII Linefeed (LF) character	»>print("Hello \nWorld") Hello World »>print("Hello \n World") Hello World »>
\N{<name>}	Character from Unicode database with given <name>	»>print(' \N{rightwards arrow}') → »>
\r	ASCII Carriage Return (CR) character	»>print("Hello \r World") World »>
\t	ASCII Horizontal Tab (TAB) character	»>print("Hello\t World") Hello World »>
\uxxxx	Unicode character with 16-bit hex value xxxx	
\Uxxxxxxxx	Unicode character with 32-bit hex value xxxxxxxx	»>print('\u2192') → »>
\v	ASCII Vertical Tab (VT) character	»>print("Hello \v World") Hello World »>

3.3.5 String operations

Note: All string methods return new values. They do not change the original string.

3.3.5.0 String split

The **split()** method returns a list(will be covered later on) of all the words in the string, using str as the separator (splits on all white-space if left unspecified), optionally limiting the number of splits to maxsplit.

Syntax:

string.split(separator, maxsplit)

Parameter Values:

Parameter	Description
separator	Optional. Specifies the separator to use when splitting the string. By default any whitespace is a separator
maxsplit	Optional. Specifies how many splits to do. The default value is -1, which is "all occurrences"

string split

```
1  #!/usr/bin/python3
2  if __name__ == '__main__':
3     str = "Orange county community college"
4     #split on spaces
5     str_split = str.split(' ')
6     print(str_split)
7     #split on charcter 'c'
8     str_split = str.split('c')
9     print(str_split)
10    #split on space but limit to one
11    str_split = str.split(' ',1)
12    print(str_split)
```

Output

```
occc@occc-VirtualBox:~/labs$ ./string.py
['Orange', 'county', 'community', 'college']
['Orange ', 'ounty ', 'ommunity ', 'ollege']
['Orange', 'county community college']
occc@occc-VirtualBox:~/labs$
```

Notice that the split character is not included in the result and is discarded

3.3.5.0 String concatenation

Concatenating With the + Operator

```
>>> 'orange' + 'county'
'orangecounty'
>>>
>>> x = "Orange "
>>> y = "County"
>>> z = x + y
>>> z
'Orange County'
>>>
```

Using the `*` math operator

```
>>> x = "Orange "
>>> y = x * 3
>>> y
'Orange Orange Orange '
>>>
```

Strings are immutable! If you concatenate or repeat a string stored in a variable, you will have to assign the new string to another variable in order to keep it.

```
>>> x = "Orange "
>>> x + "County"
'Orange County'
>>> x
'Orange '
>>>
```

```
>>> x = "Orange "
>>> x = x + "County"
>>> x
'Orange County'
>>>
```

Python does not do implicit string conversion. If you try to concatenate a string with a non-string type, Python will raise a **TypeError**:

```
>>> x = "Orange "
>>> y = x + 3.14
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: must be str, not float
>>>
```

Concatenating With. **.join()**

here is another, more powerful, way to join strings together: the join() method.

The common use case here is when you have an iterable—like a list—made up of strings, and you want to combine those strings into a single string. Like .split(), .join() is a string instance method.

```
#!/usr/bin/python3
if __name__ == '__main__':
   str_list = ["CIT","CSS","CSC"]
   sep = ":"
   new_string = sep.join(str_list)

   print(new_string)
```

Output

```
occc@occc-VirtualBox:~/labs$ ./string.py
CIT:CSS:CSC
occc@occc-VirtualBox:~/labs$
```

3.3.6 String finding

```
>>> string1 = "Orange County Community College"
>>> len(string1)  #len of string1
31
>>> string1.count('e') # How many times does the letter e ↙
   ↳  appear in the string
3
>>> string1.find('e') # Find the letter e in the string ↙
   ↳  returning index of
5
>>> string1[5]
'e'
>>> string1.index("County")
7
>>>
```

3.3.7 String trim(strip)

Python provides three methods that can be used to trim whitespaces from the string object.
Python Trim String

strip() : returns a new string after removing any leading and trailing whitespaces including tabs ().

rstrip() : returns a new string with trailing whitespace removed. It's easier to remember as removing white spaces from "right" side of the string.

lstrip() : returns a new string with leading whitespace removed, or removing whitespaces from the "left" side of the string.

string strip

```
#!/usr/bin/python3
if __name__ == '__main__':
   string1 = "  Orange  "
   print("Original string  :{0}:".format(string1))
   string_lstrip = string1.lstrip()
   string_rstrip = string1.rstrip()
   string_strip = string1.strip()
   print("lstrip string    :{0}:".format(string_lstrip))
   print("rstrip string    :{0}:".format(string_rstrip))
   print("strip string     :{0}:".format(string_strip))
```

Output

```
occc@occc-VirtualBox:~/labs$ ./string.py
Original string  :  Orange  :
lstrip string    :Orange  :
rstrip string    :  Orange:
strip string     :Orange:
occc@occc-VirtualBox:~/labs$
```

Additional string functions can be found at:

https://docs.python.org/3/library/stdtypes.html?highlight=center#str

3.3.8 Triple-Quoted Strings

There is yet another way of delimiting strings in Python. Triple-quoted strings are delimited by matching groups of three single quotes or three double quotes. Escape sequences still work in triple-quoted strings, but single quotes, double quotes, and newlines can be included without escaping them. This provides a convenient way to create a string with both single and double quotes in it:

```
>>> print('''This string has a single (') and a double (") quote.
    ''')
This string has a single (') and a double (") quote.
>>> print("""This is a
... string that spans
... across several lines""")
This is a
string that spans
across several lines
>>>
```

tripple quote string ex 1

```
!/usr/bin/python3
if __name__ == '__main__':
    str1 = """Orange """
    str2 = """County """
    str3 = """Community """
    str4 = """College"""

    # check data type of str1, str2 & str3 & str4
    print(type(str1))
    print(type(str2))
    print(type(str3))
    print(type(str4))
    print("-"*20)
    print(str1 + str2 + str3 + str4)
```

Output

```
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
--------------------
Orange County Community College
```

tripple quote string multi line

```
#!/usr/bin/python3
if __name__ == '__main__':
    my_str = """Orange
County
community
college"""
    # note in tripple quoted string we do not have to preserve
      ↪ indentation
    # check data type of my_str
    print(type(my_str))

    print(my_str)
```

Output

```
<class 'str'>
Orange
County
community
college
```

tripple quote string multi line ignore end of lines

```
#!/usr/bin/python3
if __name__ == '__main__':
    my_str = """Orange \
    County \
    community \
    college"""
    # note in tripple quoted string we do not have to preserve ↙
       ↳  indentation
    # check data type of my_str
    print(type(my_str))

    print(my_str)
```

Output

```
<class 'str'>
Orange County community college
```

3.3.9 Boolean

Boolean Type, Boolean Context, and "Truthiness"
Python 3 provides a Boolean data type. Objects of Boolean type may have one of two values, **True** or **False**:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
>>>
```

3.4 Changing variable types

```
>>> x = 123                    # integer
>>> x = 3.14                   # double float
>>> x = "hello"                # string
```

The same variable can hold different data types. However when we change the data type the original value is lost.

determine data types

```
1
2 #!/usr/bin/python3
3 x = 123
4 print(x)
5 print(type(x))   #type will not print out in script files ↙
    ↳  alone it has to be in a print function
6 x = 3.14
7 print(x)
8 print(type(x))
9 x = 'occc'
10 print(x)
11 print(type(x))
```

Output

```
123
<class 'int'>
3.14
<class 'float'>
occc
<class 'str'>
```

3.4.1 Compound assignment operators

We have already seen that we can assign the result of an arithmetic expression to a variable:

```
1 total = a + b + c + 50
```

Counting is something that is done often in a program. For example, we might want to keep a count of how many times a certain event occurs by using a variable called count. We would initialize this variable to zero and add one to it every time the event occurs. We would perform the addition with this statement:

```
1 count = count + 1
```

This is ,in fact, a very common operation. Python has a shorthand operator, `+=`, which lets us express it more cleanly, without having to write the name of the variable twice:

```
1 # These statements mean exactly the same thing:
2 count = count + 1
3 count += 1
4
5 # We can increment a variable by any number we like.
6 count += 2
7 count += 7
8 count += a + b
```

There is a similar operator, `-=`, which lets us decrement numbers:

```
# These statements mean exactly the same thing:
count = count - 3
count -= 3
```

Other common compound assignment operators are given in the table below:

assignment operators

Operator	Example	Equivalent to
+=	a += 5	a = a + 5
-=	a -= 5	a = a - 5
*=	a *= 5	a = a * 5
/=	a /= 5	a = a / 5
%=	a %= 5	a = a % 5

3.5 % modulus

The % (modulo) operator yields the remainder from the division of the first argument by the second.

mod operartor

```
#!usr/bin/python
#Demonstrate Mod operator
x = 7
y = 3
print("Division = {0} \nmod = {1}".format(x/y,x%y))
```

Output

```
Division = 2.3333333333333335
mod = 1
```

The modulus operator works on integers and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign `%`.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another: if x % y is zero, then x is divisible by y.

You can also extract the right-most digit or digits from a number. For example, x % 10 yields the right-most digit of x (in base 10). Similarly, x % 100 yields the last two digits.

3.6 Variable type conversion

As we write more programs, we will often find that we need to convert data from one type to another, for example from a string to an integer or from an integer to a floating-point number. There are two kinds of type conversions in Python: implicit and explicit conversions.

3.6.1 Implicit conversion

Recall from the section about floating-point operators that we can arbitrarily combine integers and floating-point numbers in an arithmetic expression – and that the result of any such expression will always be a floating-point number. This is because Python will convert the integers to floating-point numbers before evaluating the expression. This is an implicit conversion – we don't have to convert anything ourselves. There is usually no loss of precision when an integer is converted to a floating-point number.
For example, the integer 2 will automatically be converted to a floating-point number in the following example:

```
1 result = 8.5 * 2
```

3.6.2 Explicit conversion

Converting numbers from float to int will result in a loss of precision. For example, try to convert 5.834 to an int it is not possible to do this without losing precision. In order for this to happen, we must explicitly tell Python that we are aware that precision will be lost. For example, we need to tell the compiler to convert a float to an int like this:

```
1 i = int(5.834)
```

The int function converts a float to an int by discarding the fractional part – it will always round down! If we want more control over the way in which the number is rounded, we will need to use a different function:

data type rounding

```
# the floor and ceil functions are in the math module
import math

# ceil returns the closest integer greater than or equal to the number
# (so it always rounds up)
i = math.ceil(5.834)

# floor returns the closest integer less than or equal to the number
# (so it always rounds down)
i = math.floor(5.834)

# round returns the closest integer to the number
# (so it rounds up or down)
# Note that this is a built-in function -- we don't need to import math to use it.
i = round(5.834)
```

Explicit conversion is sometimes also called **casting**

3.6.3 Converting to and from strings

As we saw in the earlier sections, Python seldom performs implicit conversions to and from str – we usually have to convert values explicitly. If we pass a single number (or any other value) to the print function, it will be converted to a string automatically – but if we try to add a number and a string, we will get an error:

```
>>> print("3" + 4)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: must be str, not int
>>>
```

To convert numbers to strings, we can use string formatting – this is usually the cleanest and most readable way to insert multiple values into a message. If we want to convert a single number to a string, we can also use the str function explicitly:

```
print("3" + str(4))
```

3.6.4 Type Conversion

Python has several built-in conversion utilities that allow it to convert one data type another if possible. Some data types cannot be easily converted from one to another, but most can.

Function	Description	Example
ascii()	Returns a string containing a printable representation of an object	»>pi = 3.14 »>ascii(pi) '3.14' »>
bin()	Converts an integer to a binary string	»>x = 20 »>bin(x) '0b10100' »>
bool()	Converts an argument to a Boolean value	»>x = 1 »>bool(x) True »>x = 0 »>bool(x) False »>
chr()	Returns string representation of character given by integer argument	»>x = 65 »>chr(x) 'A' »>
complex()	Returns a complex number constructed from arguments	»>z = complex('5-9j') »>print(z) (5-9j) »>
float()	Returns a floating-point object constructed from a number or string	»>real_num = "2.7654" »>float(real_num) 2.7654 »>
hex()	Converts an integer to a hexadecimal string	»>x = 255 »>hex(x) '0xff' »>
int()	Returns an integer object constructed from a number or string	»>float_number = 3.14 »>string_number = "314" »>string_float = "3.14" »>int(float_number) 3 »>int(string_number) 314 »>int(string_float) Traceback (most recent call last): File "<stdin>", line 1, in <module> ValueError: invalid literal for int() with base 10: '3.14' »>
oct()	Converts an integer to an octal string	»>x = 567 »>oct(x) '0o1067' »>
ord()	Returns integer representation of a character	»>X = "A" »>ord(X) 65 »>
repr()	Returns a string containing a printable representation of an object	»>x = "foo" »>repr(x) "'foo'" »>
str()	Returns a string version of an object	»>x = 3.14 »>str(x) '3.14' »>
type()	Returns the type of an object or creates a new type object	»>x = "foo" »>type(x) <class 'str'> »>

3.7 literals

A literal is a notation for representing a fixed (const) value.
A variable is storage location associated with a symbolic name (pointed to, if you'd like).

```
occc = "Orange"
^       ^
|       |--- literal, Orange is *literally* Orange
|
|------------- variable, named "occc", and the content may vary (is variable)
```

There a re many different types of literals. Some of these we will cover later on in the course.

Literals

String literals	**"OCCC" , '12345'**
Int literals	**0,1,2,-1,-2**
Long literals	**89675L**
Float literals	**3.14**
Boolean literals	**True or False**
Complex literals	**12j**
Special literals	**None**
Unicode literals	**u"hello"**
List literals	**[], [5,6,7]**
Tuple literals	**(), (9,),(8,9,0)**
Dict literals	**{}, {'x':1}**
Set literals	**{8,9,10}**

Printing variables

```
>>> q = 459
>>> p = 0.098
>>> print(q, p, p * q)
459 0.098 44.982
>>>
```

In the print function we can print out multiple variables at a time. We can also do operations and print out the result.

4.0.1 sep=”

```
>>> print(q, p, p * q, sep=",")
459,0.098,44.982
>>> print(q, p, p * q, sep=" ||*|| ")
459 ||*|| 0.098 ||*|| 44.982
>>>
```

The **sep=””** will put a character or a series of characters between our variables.

```
>>> print(str(q) + " " + str(p) + " " + str(p * q))
459 0.098 44.982
>>>
>>> print("variable q: " + str(q) + " variable p: " + str(p) + "
    ↪   p times q: " + str(p * q))
variable q: 459 variable p: 0.098 p times q: 44.982
>>>
```

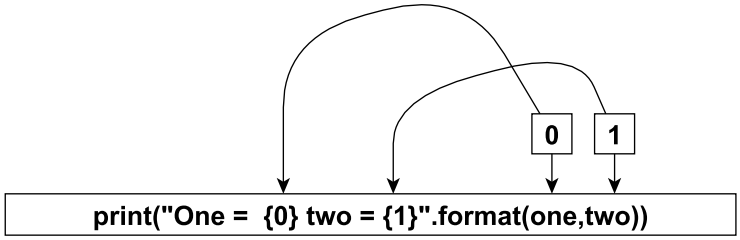
Alternatively, we can insert strings between the variables as we need them.

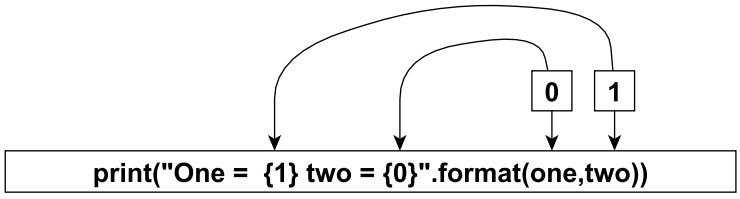
4.0.2 .format()

Formating output:

".format()"

```
>>> one = "one"
>>> two = "two"
>>> print("One =  {0} two = {1}".format(one,two))
One =  one two = two
>>> print("One =  {1} two = {0}".format(one,two))
One =  two two = one
>>>
```





Another way of doing this is:

```
>>> one = "one"
>>> two = "two"
>>> print("One = %s,  Two = %s" % (one,two))
One = one,  Two = two
>>>
```

The above way is the older way and maybe deprecated later in future releases of python.

4.0.3 Padding and aligning output

Padding and aligning strings By default values are formatted to take up only as many characters as needed to represent the content. It is however also possible to define that a value should be padded to a specific length.

```
>>> three_letters = "one"
>>> print("::{0:>10}::".format(three_letters))
::       one::
```

```
>>> print("::{0:<10}::".format(three_letters))
::one       ::
>>>
```

```
>>> print("::%10s::" %(three_letters))
::       one::
>>> print("::%-10s::" %(three_letters))
::one       ::
>>>
```

You are able to choose the padding character:

```
>>> print("::{0:_<10}::".format(three_letters))
::one_______::
>>> print("::{0:_>10}::".format(three_letters))
::_______one::
>>>
```

And also center align values:

```
>>> print("::{0:_^10}::".format(three_letters))
::___one____::
>>>
```

4.0.4 Truncating long strings

```
>>> occc = "Orange county community college"
>>> print("::{0:.8}::".format(occc))
::Orange c::
>>> print("::%.8s::" % (occc))
::Orange c::
>>>
```

4.0.5 Numbers

Of course, it is also possible to format numbers.

4.0.5.0 Integers:

```
>>> print("{0:d}".format(314))
314
>>> print("%d" % (314))
314
>>>
```

4.0.5.0 Floats

```
>>> print("{0:f}".format(3.141592653589793))
3.141593
>>> print("%f" % (3.141592653589793))
3.141593
>>>
```

4.0.5.0 Padding numbers

Similar to strings numbers can also be constrained to a specific width.

```
>>> print("::%9d::" % (314))
::      314::
>>> print("::%2d::" % (314))
::314::
>>> print("::{0:9d}::".format(314))
::      314::
>>> print("::{0:2d}::".format(314))
::314::
>>> print("::%10.2f::" % (3.141592653589793))
::      3.14::
>>> print("::%10.6f::" % (3.141592653589793))
::  3.141593::
```

```
>>> print("::%09d::" % (314))
::000000314::
>>> print("::{0:010.2f}::".format(3.141592653589793))
::0000003.14::
>>> print("::{0:010.6f}::".format(3.141592653589793))
::003.141593::
>>>
```

4.1 f-Strings

F-strings provide a way to embed expressions inside string literals, using a minimal syntax. It should be noted that an f-string is really an expression evaluated at run time, not a constant value. In Python source code, an f-string is a literal string, prefixed with 'f', which contains expressions inside braces.
f-strings were introduced in python 3.6 and are not available before that.
When tokenizing source files, f-strings use the same rules as normal strings, raw strings, binary strings, and triple quoted strings. That is, the string must end with the same character that it started with: if it starts with a single quote it must end with a single quote, etc.

Syntax:

f'<text>{<expression><optional !s, !r, or !a><optional : format specifier>}<text>

f string example 1

```
1 #!/usr/bin/python3
2
3 occc = "Orange"
4 scripting = "CIT138"
5
6 st = f"school: {occc} class: {scripting}"
7 print(st)
8
9 print(f"school: {occc} class: {scripting}")
```

Output

```
occc@occc-VirtualBox:~/labs$ ./string.py
school: Orange class: CIT138
school: Orange class: CIT138
occc@occc-VirtualBox:~/labs$
```

Expressions appear within curly braces ” and ”. While scanning the string for expressions, any doubled braces ” or ” inside literal portions of an f-string are replaced by the corresponding single brace.

f string example 2

```
1 #!/usr/bin/python3
2 occc = "Orange"
3 scripting = "CIT138"
4
5 st = f"school: {{occc}} {occc} class: {scripting}"
6 print(st)
```

Output

```
school: {occc} Orange class: CIT138
```

Arbitrary Expressions

Because f-strings are evaluated at runtime, you can put any and all valid Python expressions in them.

You could do something pretty straightforward, like this:

f string example 3

```
#!/usr/bin/python3
if __name__ == '__main__':
    print(f"{2 * 37}")
```

Output

```
74
```

f string example 4

```
#!/usr/bin/python3
if __name__ == '__main__':
    occc = "orange county community college"
    print(occc)
    print(f"{occc.upper()}")
```

Output

```
orange county community college
ORANGE COUNTY COMMUNITY COLLEGE
```

Reading user input and storing it in variables
In order to use input we use a utility(function) called **input**.

Asking the user for input

Sometimes we would like to take the value for a variable from the user via their keyboard. Python provides a built-in function called **input** that gets input from the keyboard,. When this function is called, the program stops and waits for the user to type something. When the user presses **Return** or **Enter**, the program resumes and input returns what the user typed as a **string**.

```
>>> input("Please enter a number ")
Please enter a number 5
'5'
>>> input("Please enter a number ")
Please enter a number 3.14
'3.14'
>>>
```

In python3 all user input from the keyboard is returned as a **string**.

```
>>> user_input = input("Please enter a number ")
Please enter a number 5
>>> print(user_input)
5
>>> type(user_input)
<class 'str'>
>>>
```

In order to save the result for later use we use the assignment operator "=" and direct it to a variable.
If we want to use the result in operations like arithmetic ones we have to convert it to the appropriate data type.

```
 >>> x = 3.14
```

```
>>> user_input = input("Please enter a number ")
Please enter a number 5
>>> y = x + user_input
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'float' and 'str'
>>>
```

```
>>> x = 3.14
>>> user_input = input("Please enter a number ")
Please enter a number 5
>>> y = x + float(user_input)
>>> print(y)
8.14
>>>
```

Decision Making

sequential code

```
1 #!/usr/bin/python3
2 if __name__ == '__main__':
3     first_num = 20
4     second_num = 50
5     print(first_num)
6     print(second_num)
```

The above program will run consecutively from line 1 all the way down to the last line. Sometimes we have to skip some lines in order to achieve our desired outcome. Let's say we only wanted to print out the largest of the two above numbers. We can only have one print statement.

6.1 Relational operators

Relational and comparison operators (`==`, `!=`, `>`, `<`, `>=`, `<=`)

Two expressions can be compared using relational and equality operators. For example, to know if two values are equal or if one is greater than the other.
The result of such an operation is either true or false (i.e., a Boolean value).

Arithmetic Operators

Operator	Operator name	Example	Explication
+,-,*,/	add,subtract,multiply,divide	4/2 = 2	
%	modulo	25 % 5 = 0 9 % 4 = 1	25/5 = 5, remainder = 0 9/4 = 2, remainder =1
**	exponent	2**10 = 1024	
//	floor division	9//4 = 2	Will only take the whole number

```
>>> print(9/4)
2.25
>>> print(9//4)
2
>>> print(9 + 4)
13
>>> print(9%4)
1
>>> print(2**3)
8
>>> print((10 + 4) * (4 / 5))
11.200000000000001
>>>
```

Relational Operators:

operator	description
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal

True 1
False 0

Examples:

(7 == 5)	// evaluates to false
(5 > 4)	// evaluates to true
(3 != 2)	// evaluates to true
(6 >= 6)	// evaluates to true
(5 < 5)	// evaluates to false

All relational operators evaluate to a simple **True** or **False** answer.

```
>>> 7==5
False
>>> 5 > 4
True
>>> 3 != 2
True
>>> 6 >= 6
True
>>> 5 < 5
False
>>>
```

sequence code 2

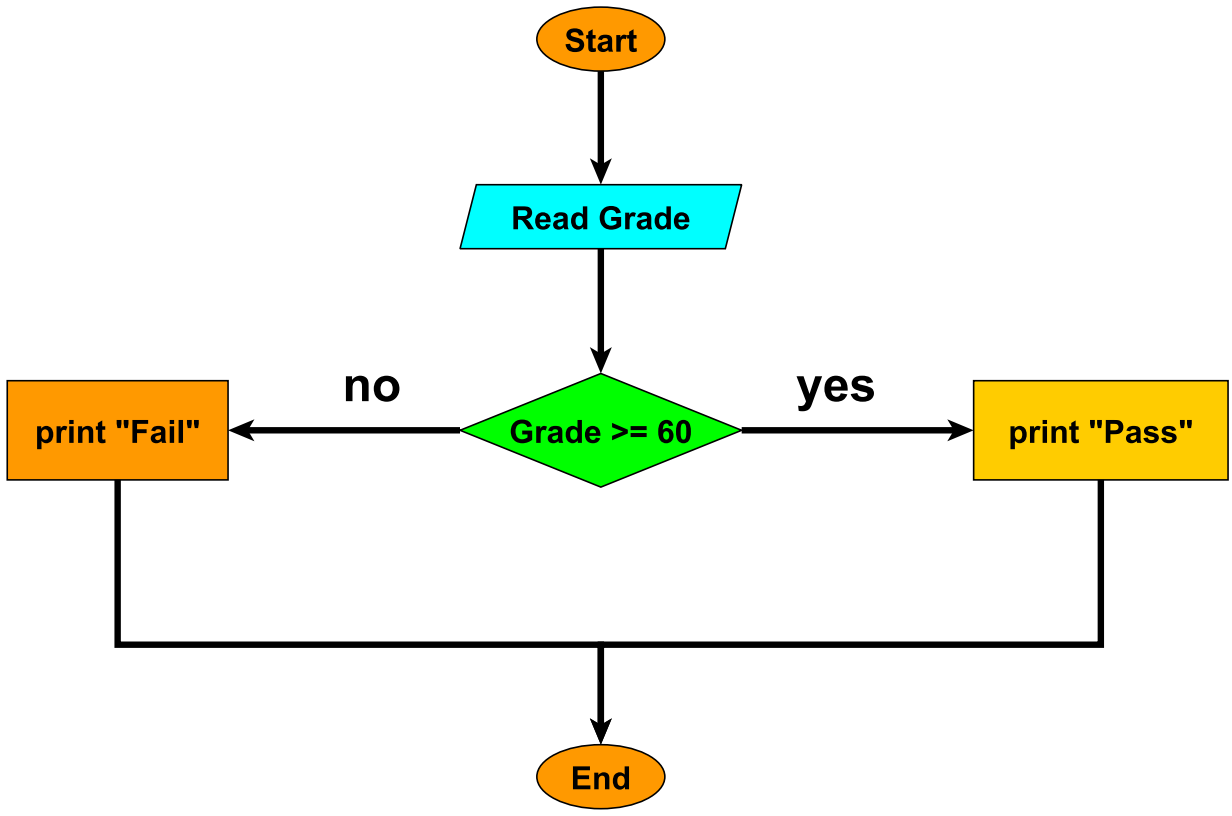
```
1 #!/usr/bin/python3
2 if __name__ == '__main__':
3     x = 0
4     y = 0
5     print(x)
6     x = input("Please enter a number ")
7     print(x)
8     y = int(x) + 10
9     print(y)
```

Output

```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
0
Please enter a number 5
5
15
occc@occc-VirtualBox:~/labs$
```

The above code is sequence code since all the statements are run one after the other.

Sometimes we have to run one set of code depending on user input. Example:



6.2 If statement

People make decisions on a daily basis. What should I have for lunch? What should I do this weekend? Every time you make a decision you base it on some criterion. For example, you might decide what to have for lunch based on your mood at the time, or whether you are on some kind of diet. After making this decision, you act on it. Thus decision-making is a two step process – first deciding what to do based on a criterion, and secondly taking an action.

Decision-making by a computer is based on the same two-step process. In Python, decisions are made with the if statement, also known as the selection statement. When processing an if statement, the computer first evaluates some criterion or conditions. If it is met, the specified action is performed. Here is the syntax for the if statement:

```
if condition:
   if_body
```

if statement example 1

```
#!/usr/bin/python3
if __name__ == '__main__':
   x = 24
   y = 34
   if (x < y):
      print("X is less than y")
```

Output

```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
X is less than y
occc@occc-VirtualBox:~/labs$
```

6.2.1 Logical Blocks

A block is a logical group of statements in a program or script. Usually, it consists of at least one statement and of declarations for the logical block, depending on the programming or scripting language. A language that allows grouping with blocks is called a block-structured language. Generally, blocks can contain blocks as well, so we get a nested block structure. A block in a script or program functions as a mean to group statements to be treated as if they were one statement. In many cases, it also serves as a way to limit the lexical scope of variables and functions.

The if the statement has to have at least one logical block which will have the code that will be run. There is no limit to how many statements the block can have, but it must have at least one indented line of code.

missing logical block

```
1 #!/usr/bin/python3
2 if __name__ == '__main__':
3     x = 24
4     y = 34
5     if(x < y):
6     print("X is less than y")
```

Output

```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
File "./first_python_script.py", line 5
print("X is less than y")
^
IndentationError: expected an indented block
occc@occc-VirtualBox:~/labs$
```

In python3 indentation has to be kept consistent. We can use spaces or tabs, but cannot mix them.

PEP 8 – Style Guide for Python Code prefers spaces.

Use 4 spaces per indentation level.

https://peps.python.org/pep-0008/#:~:text='f'%2C%20)-,Tabs%20or%20Spaces%3F,tabs%20and%20spaces%20for%20indentation.

mixed tabs and spaces

```
1 #!/usr/bin/python3
2 if __name__ == '__main__':
3     x = 24
4     y = 34
5     if(x < y):
6         print("X is less than y")
7         print("white space indent. Above is tab indent")
```

Output

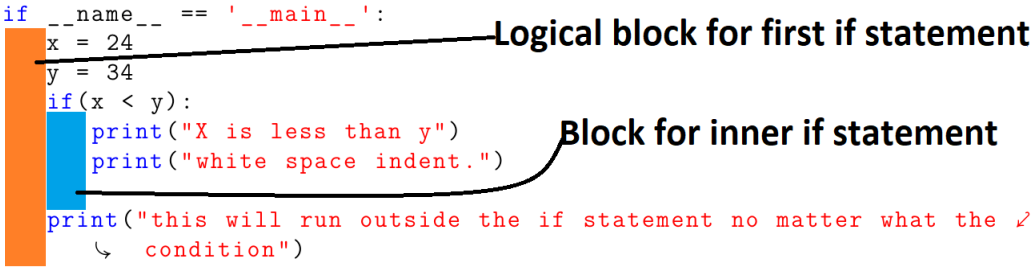
```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
File "./first_python_script.py", line 5
print("X is less than y")
^
IndentationError: expected an indented block
occc@occc-VirtualBox:~/labs$
```

continuation after if statement

```
#!/usr/bin/python3
if __name__ == '__main__':
    x = 24
    y = 34
    if(x < y):
        print("X is less than y")
        print("white space indent.")

    print("this will run outside the if statement no matter ↙
        ↳  what the condition")
```

Output

```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
X is less than y
white space indent.
this will run outside the if statement no matter what the condition
occc@occc-VirtualBox:~/labs$
```



6.2.2 String comparison.

string comparison

```
#!/usr/bin/python3
if __name__ == '__main__':
    x = "occc"
    y = "occc"
    if(x == y):
        print("X is equal to y")
```

Output

```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
X is equal to y
occc@occc-VirtualBox:~/labs$
```

Literals in if statements

comparing literals

```
#!/usr/bin/python3
if __name__ == '__main__':
    x = "occc"
    y = "occc"
    if(x == 'occc'):
        print("X is equal to 'occc'")
```

Output

```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
X is equal to 'occc'
occc@occc-VirtualBox:~/labs$
```

6.3 If else statement

Program to write flow chart that was presented earlier on

if else

```
#!/usr/bin/python3
if __name__ == '__main__':
    print("program to evaluate grades")
    grade = input("Please enter a grade:")
    grade = float(grade)    #convert to numerical format
    if(grade >= 60):
        print("you passed")
    else:
        print("You failed")
```

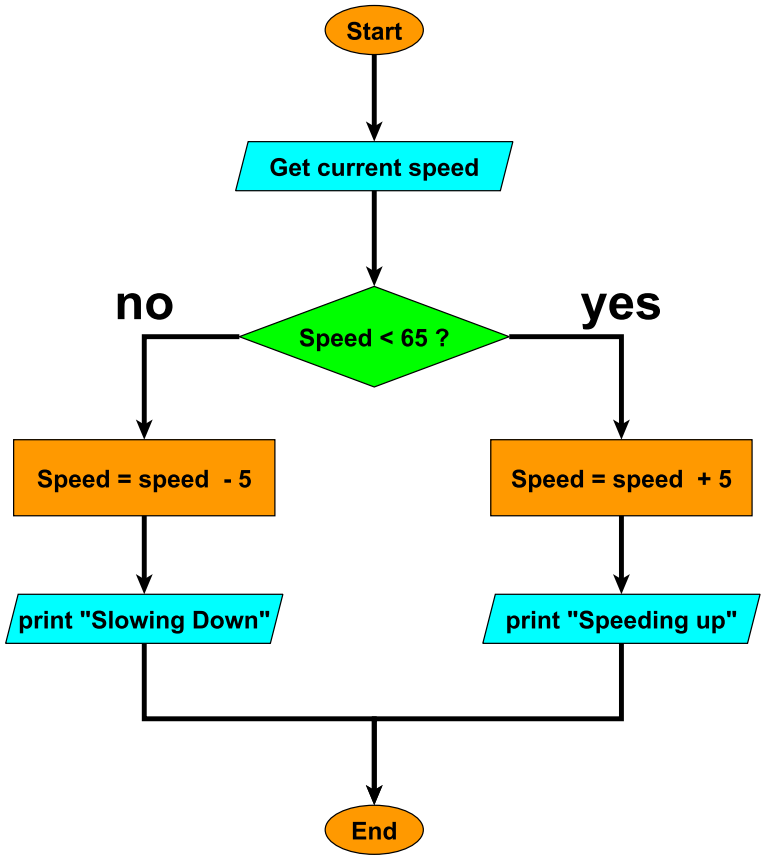
Output

```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
program to evaluate grades
Please enter a grade:98
you passed
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
program to evaluate grades
Please enter a grade:40
You failed
occc@occc-VirtualBox:~/labs$
```

if the **"if(grade >= 60):"** is evaluated and is **true** then the code immediately below will run and the **else** portion will be skipped.

print(”you passed”)

if the **”if(grade >= 60):”** is evaluated and is **false** then the code immediately below will skip and the **else** portion will run.
print(”You failed”)




```
if (speed < 65):
   speed = speed + 5;
   print( "Speeding Up")
else:
   speed = speed - 5;
   print("Slowing Down")
```

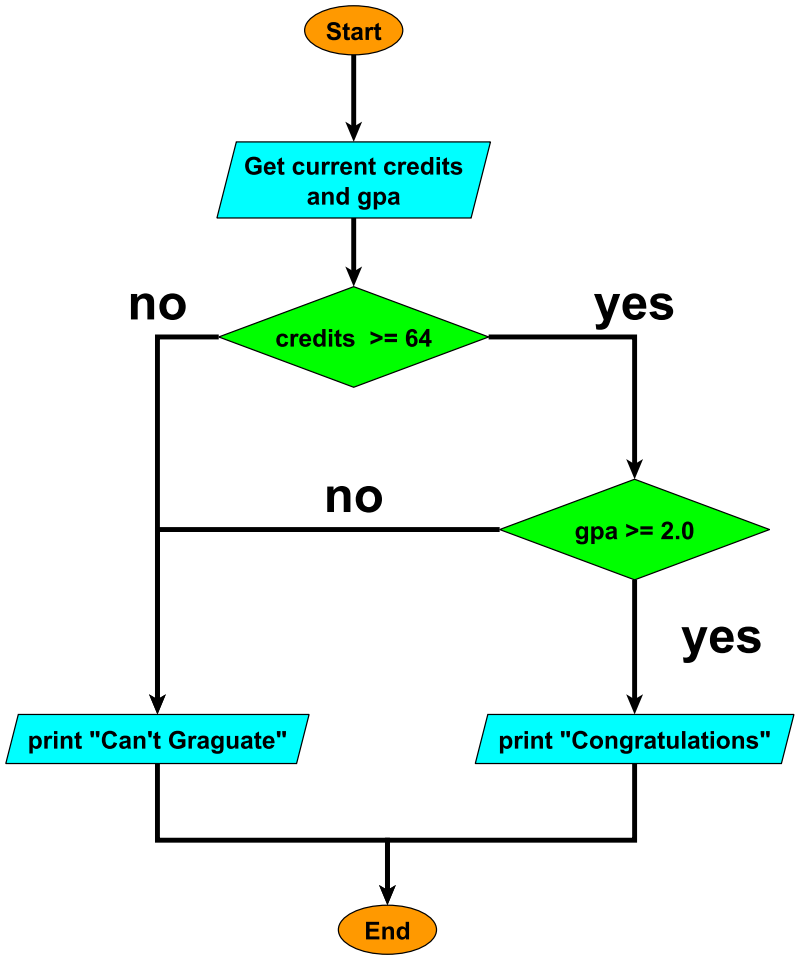
speed

```
1  #!/usr/bin/python3
2  if __name__ == '__main__':
3     print("Speed program")
4     speed = input("Please enter your speed  \n")
5     speed = float(speed)
6
7     if(speed < 65.0):
8        speed = speed + 5
9        print("Speeding up")
10    else:
11       speed = speed - 5
12       print("Slowing down")
13
14    print("your final speed is:  {0}  ".format(speed))
```

Output

```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
Speed program
Please enter your speed
67
Slowing down
your final speed is:  62.0
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
Speed program
Please enter your speed
55
Speeding up
your final speed is:  60.0
occc@occc-VirtualBox:~/labs$
```

6.4 Nested If else statement



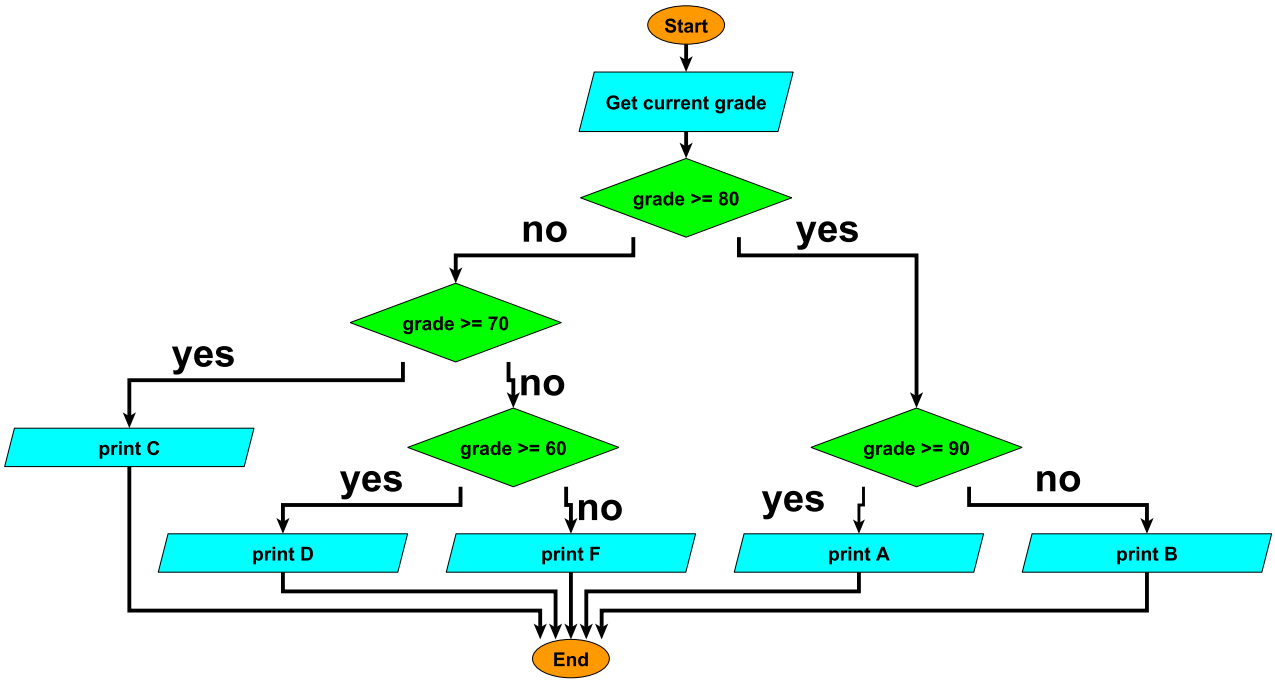

```
#!/usr/bin/python3
print("graduation program")
credits = input("Please enter credits amount\n")
credits = int(credits)

if(credits >= 65):
   gpa = input("Please enter your gpa \n")
   gpa = float(gpa)
   if(gpa >= 2.0):
      print("You can graduate \n")
   else:
      print("You cant graduate \n")
else:
   print("You cant graduate \n")
```

nested if else

```
#!/usr/bin/python3
if __name__ == '__main__':
    print("graduation program")
    credits = input("Please enter credits amount  \n")
    credits = int(credits)

    if(credits >= 65):
        gpa = input("Please enter your gpa \n")
        gpa = float(gpa)
        if(gpa >= 2.0):
            print("You can graduate \n")
        else:
            print("You cant graduate \n")
    else:
        print("You cant graduate \n")
```

Output

```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
graduation program
Please enter credits amount
65
Please enter your gpa
2.0
You can graduate

occc@occc-VirtualBox:~/labs$ ./first_python_script.py
graduation program
Please enter credits amount
67
Please enter your gpa
1.5
You cant graduate

occc@occc-VirtualBox:~/labs$ ./first_python_script.py
graduation program
Please enter credits amount
34
You cant graduate

occc@occc-VirtualBox:~/labs$
```




```
if (grade>=80):
   if (grade >= 90):
      print("A")
   else:
      print("B")
else:
   if (grade >= 70):
      print("C")
   else:
      if (grade >= 60)
         print("D")
      else:
         print("F")
```

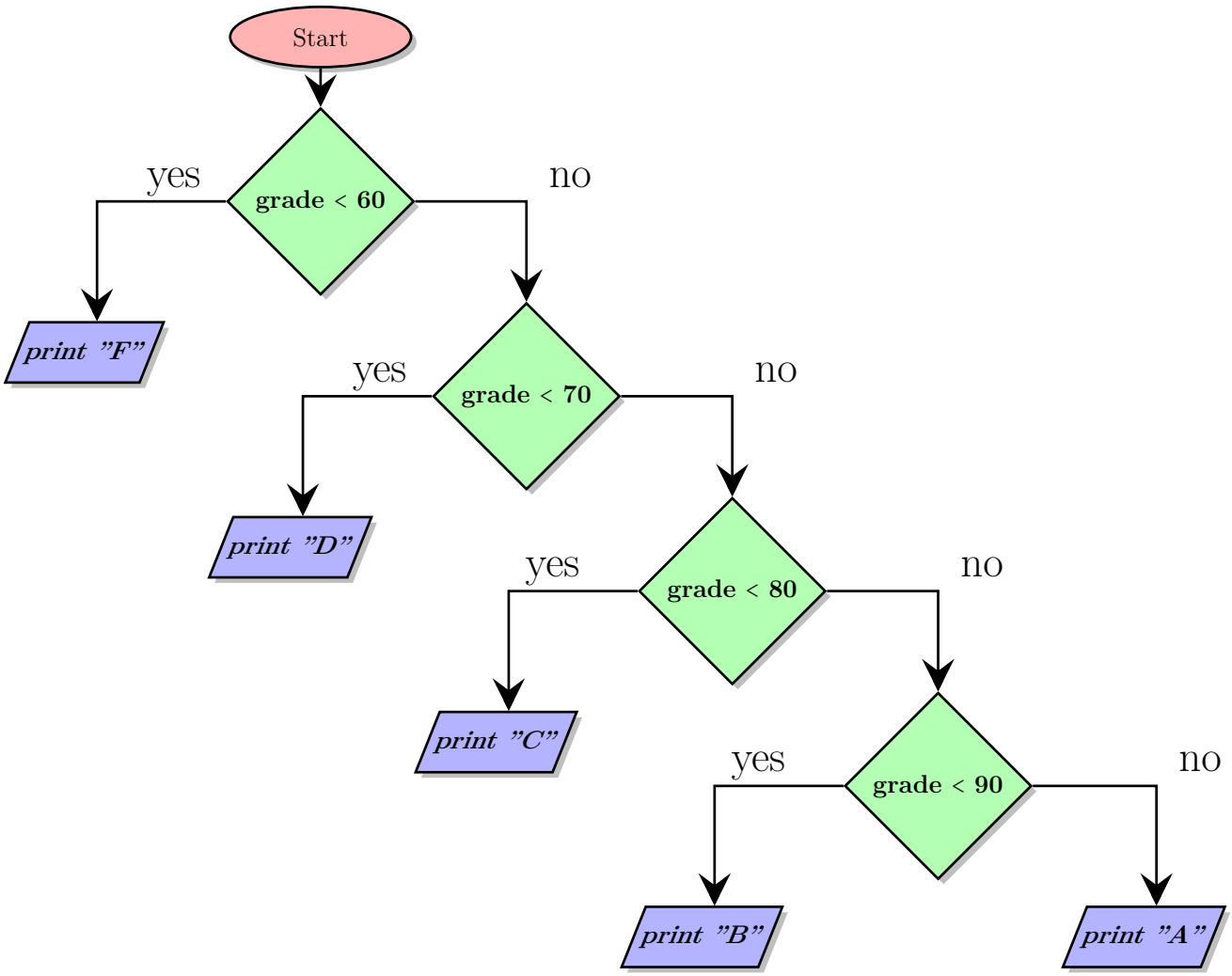
6.5 if / else if

if else if

```
#!/usr/bin/python3
if __name__ == '__main__':
    x = int(input("Please enter an integer: "))

    if x < 0:
        x = 0
        print('Negative changed to zero')
    elif x == 0:
        print('Zero')
    elif x == 1:
        print('Single')
    else:
        print('More')
```

Output

```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
Please enter an integer: 67
More
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
Please enter an integer: -56
Negative changed to zero
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
Please enter an integer: 1
Single
occc@occc-VirtualBox:~/labs$
```




```
if (grade < 60):
   print("F")
elif (grade < 70):
   print("D")
elif (grade < 80):
   print("C")
elif (grade < 90):
   print("B")
else:
   print("A")
```

6.5.1 Trailing else

The last else in if/elif is optional and in many cases is used to catch errors.

6.6 Logical operators (and, or, not)

The operator **"not"** is the python operator for the Boolean operation NOT. It has only one operand, to its right, and inverts it, producing false if its operand is true, and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand. For example:

```
not(5 == 5)     // evaluates to false because the expression at its right (5 == 5) is true
not(6 <= 4)     // evaluates to true because (6 <= 4) would be false
not true        // evaluates to false
not false       // evaluates to true
```

```
>>> not(5 == 5)
False
>>> not(6 <= 4)
True
>>> not True
False
>>> not False
True
>>>
```

The logical operators **"and"** and **"or"** are used when evaluating two expressions to obtain a single relational result. The operator "and" corresponds to the Boolean logical operation AND, which yields true if both its operands are true, and false otherwise.

Logical AND

a	b	a and b
true	true	true
true	false	false
false	true	false
false	false	false

Logical OR

a	b	a or b
true	true	true
true	false	true
false	true	true
false	false	false

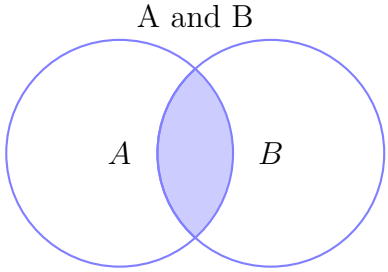
Logical not

a	not a
true	false
false	true

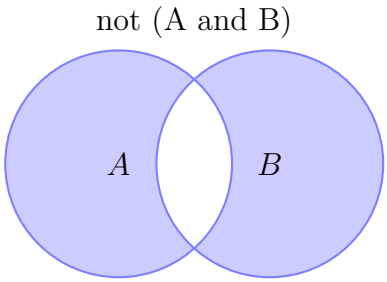
The operator **"or"** corresponds to the Boolean logical operation OR, which yields true if either of its operands is true, thus being false only when both operands are false.

6.6.1 Logical Venn diagrams

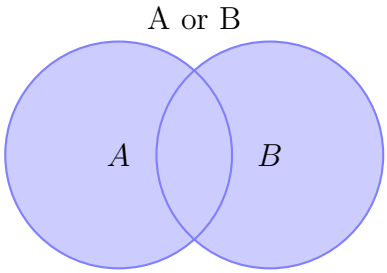
AND



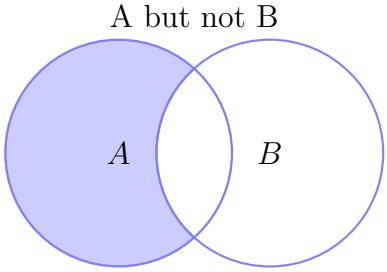
not (A and B)



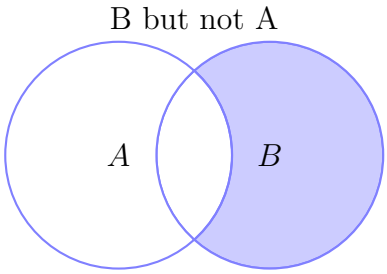
A or B



A - B



B - A




```
( (5 == 5) and (3 > 6) )  // evaluates to false ( true and false )
( (5 == 5) or (3 > 6) )   // evaluates to true ( true or false )
```

Associativity

Operator	Associativity
not	Right
and	Left
or	Left

short circuit

operator	short-circuit
and	if the left-hand side expression is false, the combined result is false (the right-hand side expression is never evaluated).
or	if the left-hand side expression is true, the combined result is true (the right-hand side expression is never evaluated).

logical operators

```
1  #!/usr/bin/python3
2  if __name__ == '__main__':
3      x = 40
4      y = 70
5
6      if((x < 90) and (y == 70)):
7          print('(x < 90) and (y == 70)')
8
9      if((x < 90) and (y != 70)):
10         print('(x < 90) and (y != 70)')
11     else:
12         print(" NOT (x < 90) and (y != 70")
13
14     if((x < 90) or (y == 200)):
15         print('(x < 90) or (y == 200)')
```

Output

```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
(x < 90) and (y == 70)
NOT (x < 90) and (y != 70
(x < 90) or (y == 200)
occc@occc-VirtualBox:~/labs$
```

6.7 try: except:

★ Validating User Input

6.7.1 Validating User Input

When getting data from an unknown source such as a user we must make sure that the user entered the right data.
Example: If you write a program to do some basic math functions such as divide make sure you don't end up with something like divide by zero.

validating user input detect errors

```
1 #!/usr/bin/python3
2 if __name__ == '__main__':
3     x = int(input("please enter an integer\n"))
4     print("user entered {0}".format(x))
5     y = 5 / x
6     print("Result after division {0}".format(y))
```

Output

```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
please enter an integer
5
user entered 5
Result after division 1.0
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
please enter an integer
0
user entered 0
Traceback (most recent call last):
File "./first_python_script.py", line 4, in <module>
y = 5 / x
ZeroDivisionError: division by zero
occc@occc-VirtualBox:~/labs$
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
please enter an integer
occc
Traceback (most recent call last):
File "./first_python_script.py", line 2, in <module>
x = int(input("please enter an integer\n"))
ValueError: invalid literal for int() with base 10: 'occc'
occc@occc-VirtualBox:~/labs$
```

There are several methods we can use to check user input.
do a series of if statements or use try: except:
if statements would have caught the user entering a value of 0(zero), but it would not have done anything for the user not entering a correct input of integer.
In Python, we can try to run a block of code and see if it generates an error. If that happens we can catch the error and make a decision based on it.

try-except

```
#!/usr/bin/python3
if __name__ == '__main__':
    try:
        x = int(input("please enter an positive integer greater
            ↪  than zero \n"))
        print("user entered {0}".format(x))
        y = 5 / x
        print("Result after division {0}".format(y))
    except:
        print("you entered a value of zero or not a integer\n")
```

Output

```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
please enter an positive integer greater than zero
0
user entered 0
you entered a value of zero or not a integer

occc@occc-VirtualBox:~/labs$ ./first_python_script.py
please enter an positive integer greater than zero
occc
you entered a value of zero or not a integer

occc@occc-VirtualBox:~/labs$
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
please enter an positive integer greater than zero
7
user entered 7
Result after division 0.7142857142857143
occc@occc-VirtualBox:~/labs$
```

The try statement works as follows.

First, the try clause (the statement(s) between the try and except keywords) is executed.
If no exception occurs, the except clause is skipped and execution of the try statement is finished.
If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.
If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.

The above code helps of with catching errors, but we don't know what king of error has occurred.
We can have multiple exceptions.

format of try except:

try - except

```
try:
   You do your operations here
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
      ......................
else:
      If there is no exception then execute this block.
```

There is a comprehensive list of exceptions at https://docs.python.org/3/library/exceptions.html
some of the interesting ones for our case above would be:

exception **ZeroDivisionError**
Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.
exception **ValueError**
Raised when an operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as **IndexError**.

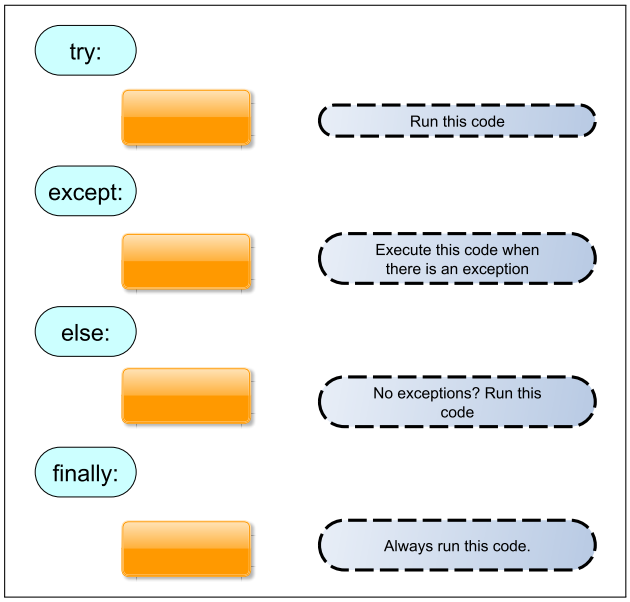
multiple try-except

```
#!/usr/bin/python3
if __name__ == '__main__':
   try:
      x = int(input("please enter an positive integer greater ↙
         ↳  than zero \n"))
      print("user entered {0}".format(x))
      y = 5 / x
      print("Result after division {0}".format(y))
   except ZeroDivisionError:
      print("You tried to divide by zero\n")
   except ValueError:
      print("I could not convert your input into a valid ↙
         ↳  integer\n")
   else:
      print("There were no errors\n")
```

Output

```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
please enter an positive integer greater than zero
7
user entered 7
Result after division 0.7142857142857143
There were no errors

occc@occc-VirtualBox:~/labs$ ./first_python_script.py
please enter an positive integer greater than zero
occc
I could not convert your input into a valid integer

occc@occc-VirtualBox:~/labs$ ./first_python_script.py
please enter an positive integer greater than zero
0
user entered 0
You tried to divide by zero

occc@occc-VirtualBox:~/labs$
```

Imagine that you always had to implement some sort of action to clean up after executing your code. Python enables you to do so using the finally clause.



6.8 string slices

```
1 #!/usr/bin/python3
2 word = "Python"
3 print(word[0])  # character in position 0
4 print(word[3])  # character in position 3
```

```
5 print(word[0:2])  # characters from position 0 (included) to 2 (↙
    ↳ excluded)
6 print(word[2:5])  # characters from position 2 (included) to 5 (↙
    ↳ excluded)
```

Output

```
P
h
Py
tho
```

Indexes of letters,

S	U	N	Y		O	r	a	n	g	e
0	1	2	3	4	5	6	7	8	9	10

Negative indexes.

S	U	N	Y		O	r	a	n	g	e
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

string slices

```
1 #!/usr/bin/python3
2 if __name__ == '__main__':
3     word = "Suny Orange"
4     #positive slices
5     print("word = Suny Orange")
6     print("word[0:4]")
7     print(word[0:4])
8     print("word[-6:-1]")
9     print(word[-6:-1])
```

Output

```
word = Suny Orange
word[0:4]
Suny
word[-6:-1]
Orang
```

6.8.1 Slice stride

String slicing can accept a third parameter in addition to two index numbers. The third parameter specifies the **stride**, which refers to how many characters to move forward after the first character is retrieved from the string. So far, we have omitted the stride parameter, and Python defaults to the stride of 1, so that every character between two index numbers is retrieved.

slice stride

```
#!/usr/bin/python3
if __name__ == '__main__':
    word = "Suny Orange"

    print(word[0:12])  #default
    print(word[0:12:1]) #same as default
    print(word[0:12:2])  #take every second character
    print(word[0:12:3])  #take every third character
```

Output

```
Suny Orange
Suny Orange
Sn rne
Syrg
```

6.9 Length of variables

variable Length

```
#!/usr/bin/python3
if __name__ == '__main__':
    word = "Python"
    word_len = len(word)
    print(word)
    print("length of word = {0}\n".format(word_len))
    List1 = [1,2,3,4,5,6,7,8,9,10]
    print(List1)
    List1_len = len(List1)
    print("Length of List1 = {0}\n".format(List1_len))
```

Output

```
Python
length of word = 6
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Length of List1 = 10
```

List in python is implemented to store the sequence of various types of data.
A list can be defined as a collection of values or items of different types. The items in the list are separated with the comma (,) and enclosed with the square brackets [].

The important characteristics of Python lists are as follows:

- Lists are ordered.
- Lists can contain any arbitrary objects.
- List elements can be accessed by index.
- Lists can be nested to arbitrary depth.
- Lists are mutable.
- Lists are dynamic.

A list can be defined as follows.

list defined

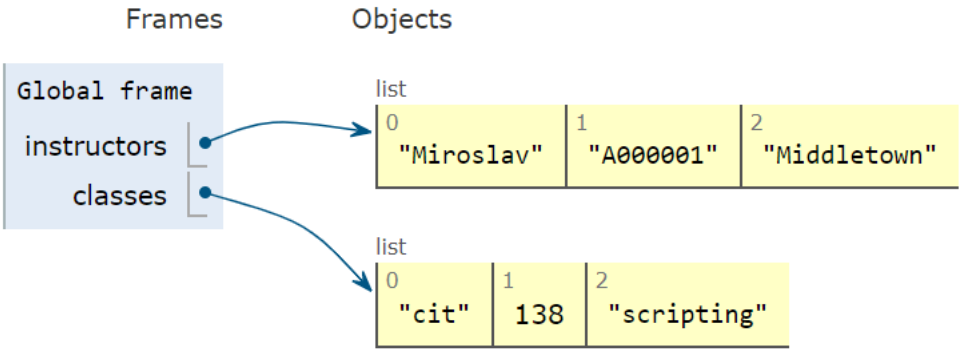
```
1 #!/usr/bin/python3
2 if __name__ == '__main__':
3     instructors = ["Miroslav","A000001","Middletown"]
4     classes = ["cit",138,"scripting"]
5
6     print("instructors at occc: Name: %s, Id: %s, Campus: %s\n"↙
        ↳    % (instructors[0],instructors[1],instructors[2]))
7     print("Instructor: {0} teaches: {1},{2},{3}\n".format(↙
        ↳  instructors[0],classes[0],classes[1],classes[2]))
```

Output

```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
instructors at occc: Name: Miroslav, Id: A000001, Campus: Middletown

Instructor: Miroslav teaches: cit,138,scripting

occc@occc-VirtualBox:~/labs$
```

instructors = [”Miroslav”,”A000001”,”Middletown”]

instructors = [”Miroslav”,”A000001”,”Middletown”]
classes = [”cit”,138,”scripting”]



7.0.1 List indexing and splitting

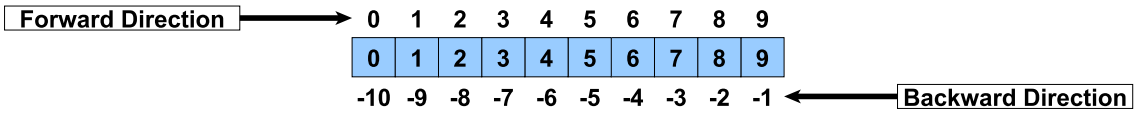
The indexing is processed in the same way as it happens with the strings. The elements of the list can be accessed by using the slice operator [].

The index starts from 0 and goes to length - 1. The first element of the list is stored at the 0th index, the second element of the list is stored at the 1st index, and so on.

Consider the following example.

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

List [0] = 0	List[0:] = [0,1,2,3,4,5,6,7,8,9]
List [1] = 1	List[5:] = [5,6,7,8,9]
List [2] = 2	List[:] = [0,1,2,3,4,5,6,7,8,9]
List [3] = 3	List [2:4]= [2,3]
List [4] = 4	List[:4] = [0,1,2,3]
List [5] = 5	List[-3:-1] = [7,8]
List [6] = 6	List[:-1] = [0,1,2,3,4,5,6,7,8]
List [7] = 7	
List [8] = 8	
List [9] = 9	



list

```
1 #!/usr/bin/python3
2 if __name__ == '__main__':
3     animals_in_zoo = ['Elephant',"Zebra","Kinkajou","Serval"]
4     print(animals_in_zoo)
```

Output

```
['Elephant', 'Zebra', 'Kinkajou', 'Serval']
```

7.0.2 List append

If we get a new animal we can add it to our zoo list with the **.append()**

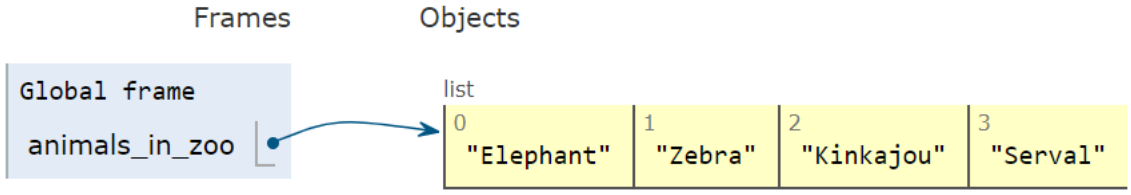
list append

```
1 #!/usr/bin/python3
2 if __name__ == '__main__':
3     animals_in_zoo = ['Elephant',"Zebra","Kinkajou","Serval"]
4     print(animals_in_zoo)
5     animals_in_zoo.append("Caracal")
6     print(animals_in_zoo)
```

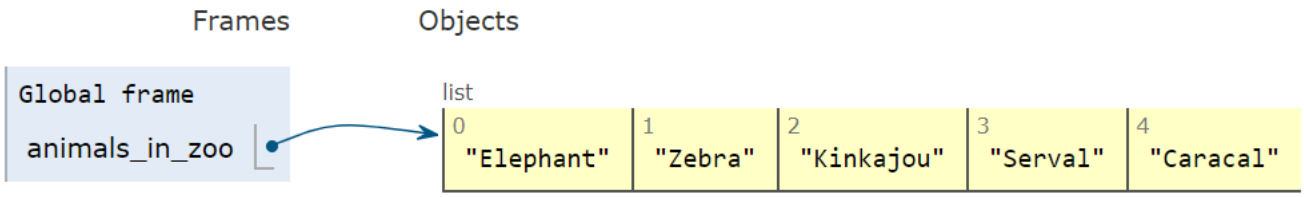
Output

```
['Elephant', 'Zebra', 'Kinkajou', 'Serval']
['Elephant', 'Zebra', 'Kinkajou', 'Serval', 'Caracal']
```

animals_in_zoo = [’Elephant’,”Zebra”,”Kinkajou”,”Serval”]



animals_in_zoo.append(”Caracal”)



Append will always insert items at the end of the list.

7.0.3 list.insert()

The **list.insert(i,x)** method takes two arguments, with ”i” being the index position you would like to add an item to, and x is the item itself.

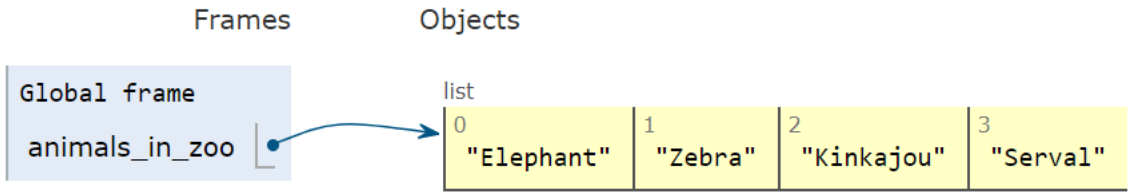
list insert

```
#!/usr/bin/python3
if __name__ == '__main__':
    animals_in_zoo = ['Elephant',"Zebra","Kinkajou","Serval"]
    print(animals_in_zoo)
    animals_in_zoo.append("Caracal")
    print(animals_in_zoo)
    animals_in_zoo.insert(0,"Capybara")
    print(animals_in_zoo)
```

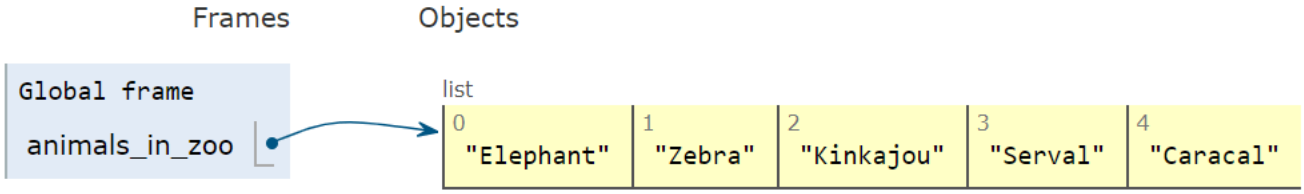
Output

```
['Elephant', 'Zebra', 'Kinkajou', 'Serval']
['Elephant', 'Zebra', 'Kinkajou', 'Serval', 'Caracal']
['Capybara', 'Elephant', 'Zebra', 'Kinkajou', 'Serval', 'Caracal']
```

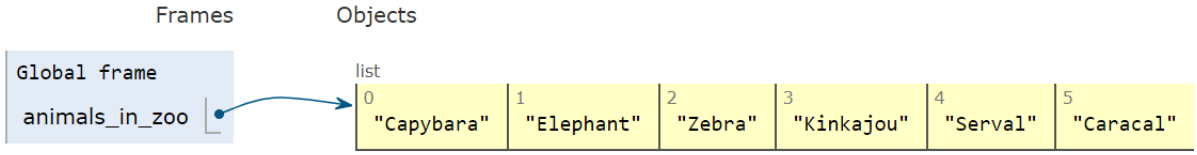
animals_in_zoo = [’Elephant’,”Zebra”,”Kinkajou”,”Serval”]



animals_in_zoo.append(”Caracal”)



animals_in_zoo.insert(0,”Capybara”)



7.0.4 list.extend()

If we want to combine more than one list, we can use the list.extend(L) method, which takes in a second list as its argument.

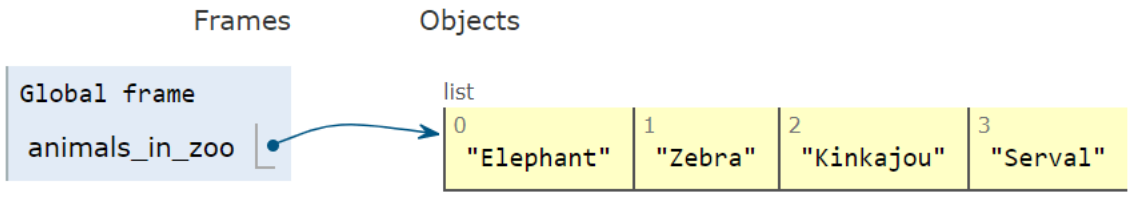
list extend

```
#!/usr/bin/python3
if __name__ == '__main__':
    animals_in_zoo = ['Elephant',"Zebra","Kinkajou","Serval"]
    print(animals_in_zoo)
    reptiles = ["Python","rattle snake"]
    animals_in_zoo.extend(reptiles)
    print(animals_in_zoo)
```

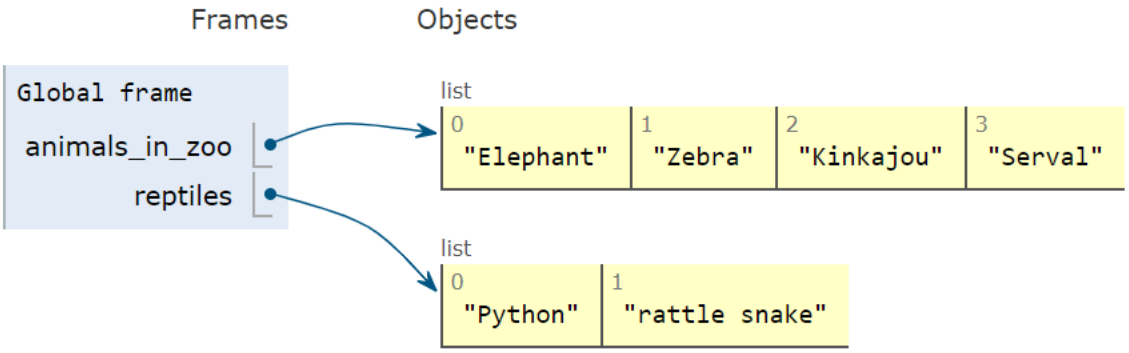
Output

```
['Elephant', 'Zebra', 'Kinkajou', 'Serval']
['Elephant', 'Zebra', 'Kinkajou', 'Serval', 'Python', 'rattle snake']
```

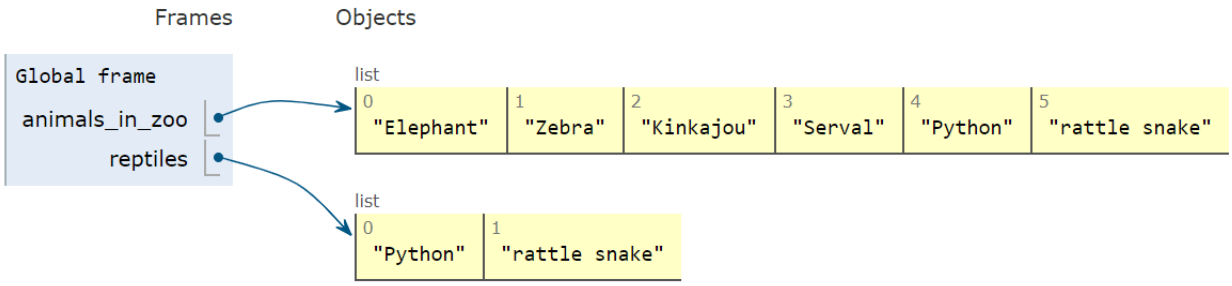
animals_in_zoo = [’Elephant’,”Zebra”,”Kinkajou”,”Serval”]



animals_in_zoo = [’Elephant’,”Zebra”,”Kinkajou”,”Serval”]
reptiles = [”Python”,”rattle snake”]



animals_in_zoo = [’Elephant’,”Zebra”,”Kinkajou”,”Serval”]
reptiles = [”Python”,”rattle snake”]
animals_in_zoo.extend(reptiles)



7.0.5 list.remove()

When we need to remove an item from a list, we'll use the **list.remove(x)** method which removes the first item in a list whose value is equivalent to x.

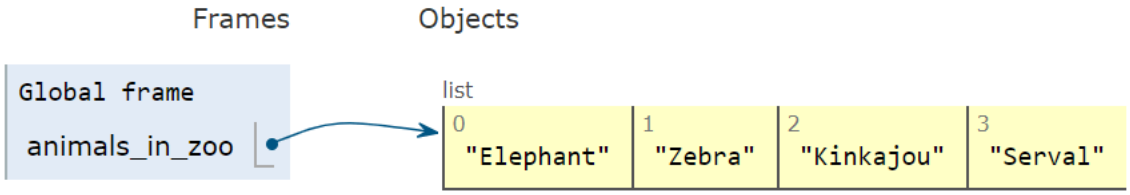
list remove

```
1 #!/usr/bin/python3
2 if __name__ == '__main__':
3     animals_in_zoo = ['Elephant',"Zebra","Kinkajou","Serval"]
4     print(animals_in_zoo)
5     reptiles = ["Python","rattle snake"]
6     animals_in_zoo.extend(reptiles)
7     print(animals_in_zoo)
8     animals_in_zoo.remove("Zebra")
9     print(animals_in_zoo)
```

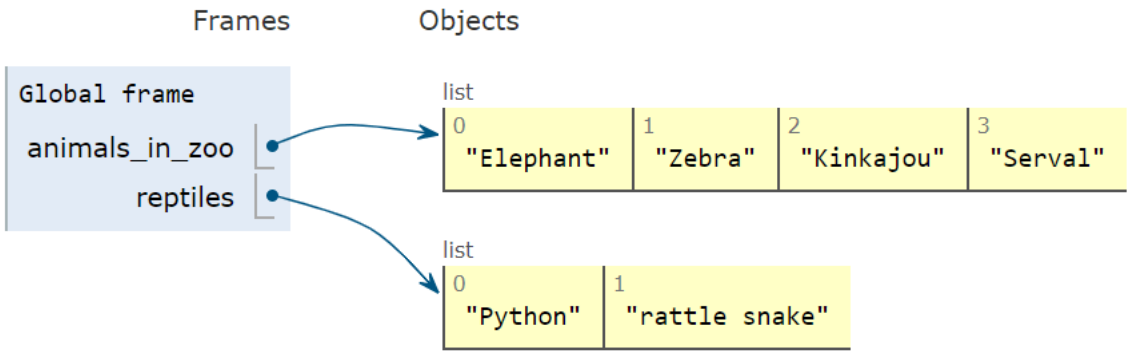
Output

```
['Elephant', 'Zebra', 'Kinkajou', 'Serval']
['Elephant', 'Zebra', 'Kinkajou', 'Serval', 'Python', 'rattle snake']
['Elephant', 'Kinkajou', 'Serval', 'Python', 'rattle snake']
```

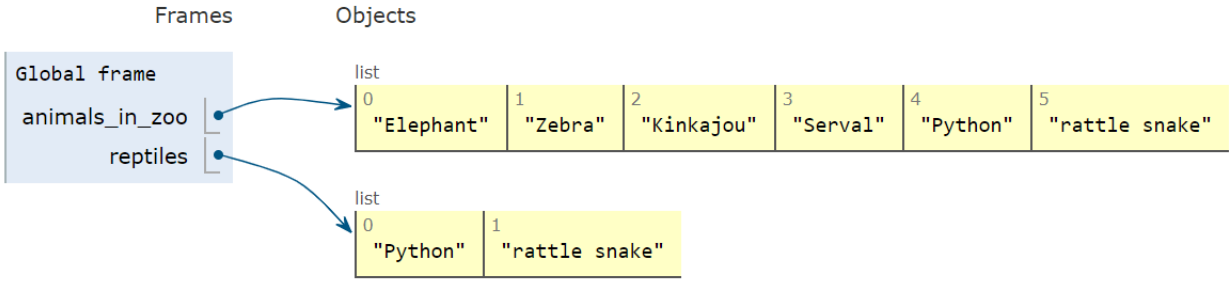
animals_in_zoo = ['Elephant','Zebra","Kinkajou","Serval"]



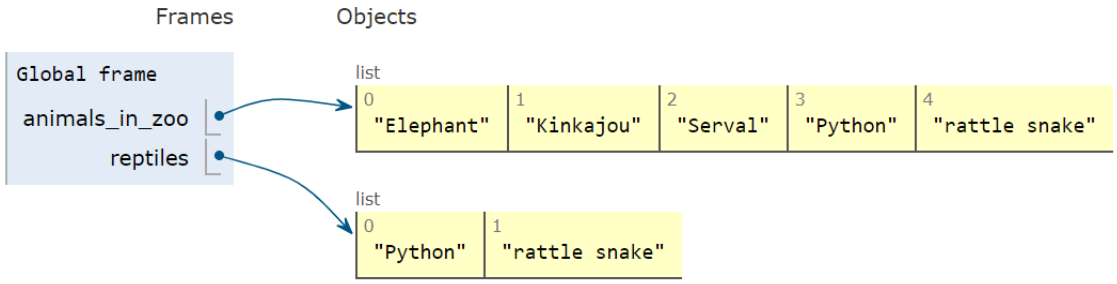
animals_in_zoo = ['Elephant','Zebra","Kinkajou","Serval"]
reptiles = ["Python","rattle snake"]




```
animals_in_zoo = ['Elephant',"Zebra","Kinkajou","Serval"]
reptiles = ["Python","rattle snake"]
animals_in_zoo.extend(reptiles)
```




```
animals_in_zoo = ['Elephant',"Zebra","Kinkajou","Serval"]
reptiles = ["Python","rattle snake"]
animals_in_zoo.extend(reptiles)
animals_in_zoo.remove("Zebra")
```



list remove

```
#!/usr/bin/python3
if __name__ == '__main__':
    animals_in_zoo = ['Elephant',"Zebra","Kinkajou","Serval"]
    print(animals_in_zoo)
    animals_in_zoo.remove("occc") #item that does not exist in list
    print(animals_in_zoo)
```

Output

```
['Elephant', 'Zebra', 'Kinkajou', 'Serval']
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    animals_in_zoo.remove("occc") #item that does not exist in list
ValueError: list.remove(x): x not in list
```

removing an item that does not exist will cause a exception.

list remove try catch

```
1 #!/usr/bin/python3
2 if __name__ == '__main__':
3     animals_in_zoo = ['Elephant', "Zebra", "Kinkajou", "Serval"
      ↪ ]
4     print(animals_in_zoo)
5     try:
6         animals_in_zoo.remove("occc") #item that does not exist
          ↪ in list
7     except ValueError:
8         print("Item was not in the list")
```

Output

```
['Elephant', 'Zebra', 'Kinkajou', 'Serval']
Item was not in the list
```

7.0.6 list.pop()

We can use the **list.pop([i])** method to return the item at the given index position from the list and then remove that item. The square brackets around the i for index tell us that this parameter is optional, so if we don't specify an index (as in fish.pop()), the last item will be returned and removed.
Removing an index that does not exist will cause an exception.

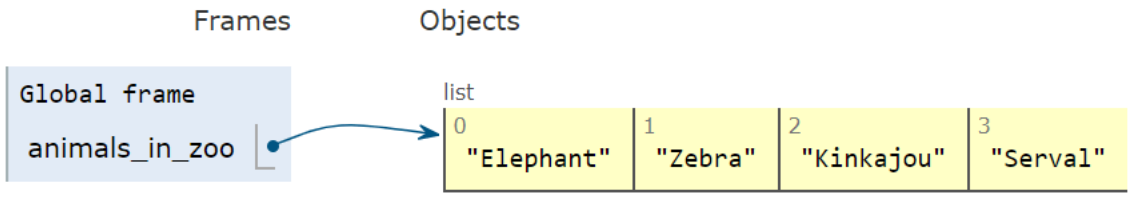
list pop

```
#!/usr/bin/python3
if __name__ == '__main__':
    animals_in_zoo = ['Elephant',"Zebra","Kinkajou","Serval"]
    print(animals_in_zoo)
    reptiles = ["Python","rattle snake"]
    animals_in_zoo.extend(reptiles)
    print(animals_in_zoo)
    animal = animals_in_zoo.pop(1)
    print("Animal removed : {0}\n".format(animal))
    print(animals_in_zoo)
```

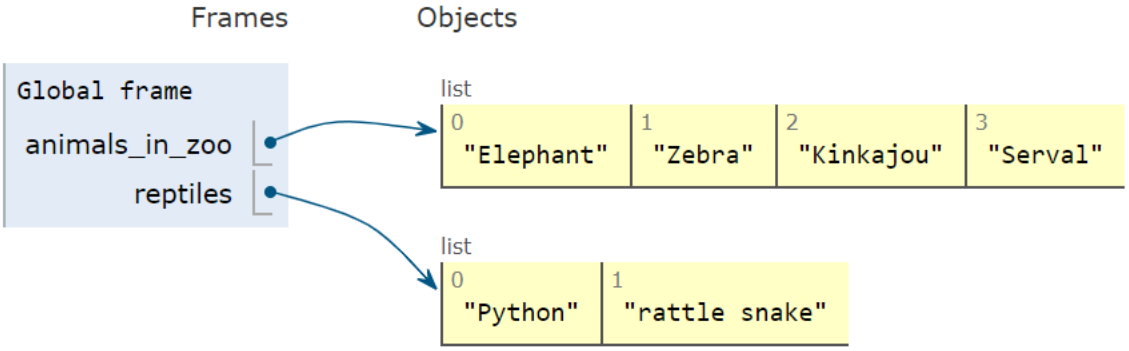
Output

```
['Elephant', 'Zebra', 'Kinkajou', 'Serval']
['Elephant', 'Zebra', 'Kinkajou', 'Serval', 'Python', 'rattle snake']
Animal removed : Zebra

['Elephant', 'Kinkajou', 'Serval', 'Python', 'rattle snake']
```

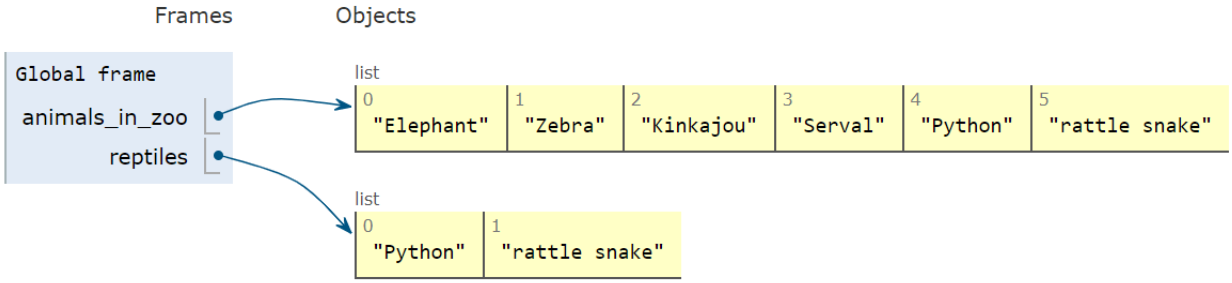
animals_in_zoo = [’Elephant’,”Zebra”,”Kinkajou”,”Serval”]



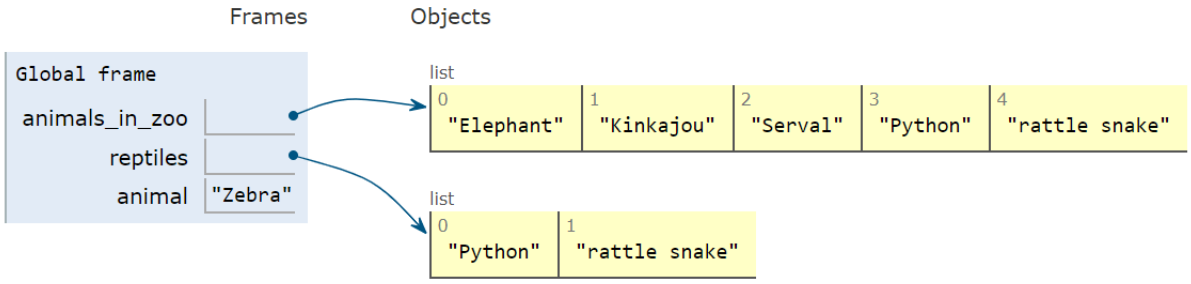
animals_in_zoo = [’Elephant’,”Zebra”,”Kinkajou”,”Serval”]
reptiles = [”Python”,”rattle snake”]



animals_in_zoo = [’Elephant’,”Zebra”,”Kinkajou”,”Serval”]
reptiles = [”Python”,”rattle snake”]
animals_in_zoo.extend(reptiles)



animals_in_zoo = [’Elephant’,”Zebra”,”Kinkajou”,”Serval”]
reptiles = [”Python”,”rattle snake”]
animals_in_zoo.extend(reptiles)
animal = animals_in_zoo.pop(1)



7.0.7 list.index()

When lists start to get long, it becomes more difficult for us to count out our items to determine at what index position a certain value is located. We can use the list.index(x), where x is equivalent to item value, to return the index in the list where that item is located. If there is more than one item with value x, this method will return the first index location.
getting an index of an item that does not exist will cause an exception.

list index

```
1 #!/usr/bin/python3
2 if __name__ == '__main__':
3     animals_in_zoo = ['Elephant',"Zebra","Kinkajou","Serval"]
4     print(animals_in_zoo)
5     zebra_index = animals_in_zoo.index("Zebra")
6     print("Zebra is at index : {0}\n".format(zebra_index))
```

Output

```
['Elephant', 'Zebra', 'Kinkajou', 'Serval']
Zebra is at index : 1
```

7.0.8 list.copy()

When we are working with a list and may want to manipulate it in multiple ways while still having the original list available to us unchanged, we can use list.copy() to make a copy of the list.

list copy

```
1 #!/usr/bin/python3
2 if __name__ == '__main__':
3     animals_in_zoo = ['Elephant',"Zebra","Kinkajou","Serval"]
4     animals2 = animals_in_zoo.copy()
5     animals2.pop(0)
6     print(animals_in_zoo)
7     print(animals2)
```

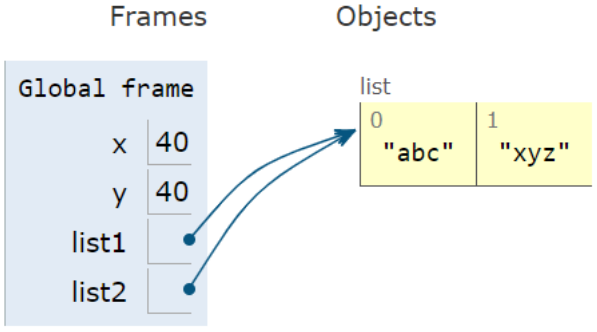
Output

```
['Elephant', 'Zebra', 'Kinkajou', 'Serval']
['Zebra', 'Kinkajou', 'Serval']
```

Unlike variables collections of data like list, tuple,set,dictionary cannot be simply assigned to another variable to make a copy. if we make another variable and assign the first on to it it will create an alias.

list copy alias

```
1     #!/usr/bin/python3
2     x = 40
3     y = x
4     list1 = ['abc','xyz']
5     list2 = list1
```



notice that the list2 is pointing to the same data as list1.

7.0.9 list.count()

The list.count(x) method will return the number of times the value x occurs within a specified list.

list count

```
#!/usr/bin/python3
if __name__ == '__main__':
    animals_in_zoo = ['Elephant',"Zebra","Kinkajou","Serval","Elephant"]
    animals2 = animals_in_zoo.copy()
    animals2.pop(0)
    print(animals_in_zoo)
    print(animals2)
    print("Total Elephants  in zoo: {0} \n".format(animals_in_zoo.count("Elephant")))
    print("Total Elephants in zoo copy: {0}\n".format(animals2.count("Elephant")))
```

Output

```
['Elephant', 'Zebra', 'Kinkajou', 'Serval', 'Elephant']
['Zebra', 'Kinkajou', 'Serval', 'Elephant']
Total Elephants  in zoo: 2

Total Elephants in zoo copy: 1
```

7.0.10 list.sort()

We can use the list.sort() method to sort the items in a list.

```
#!/usr/bin/python3
if __name__ == '__main__':
    animals_in_zoo = ['Elephant',"Zebra","Kinkajou","Serval","Elephant"]
    print(animals_in_zoo)
    print("sorted animals\n")
    animals_in_zoo.sort()
    print(animals_in_zoo)
```

Output

```
['Elephant', 'Zebra', 'Kinkajou', 'Serval', 'Elephant']
sorted animals

['Elephant', 'Elephant', 'Kinkajou', 'Serval', 'Zebra']
```

Items in the list have to be type compatible for sort to work. If the list has a mix of strings and integers it will fail.

list sort incompatible types

```
1 #!/usr/bin/python3
2 if __name__ == '__main__':
3     animals_in_zoo = ['Elephant', "Zebra", "Kinkajou", "Serval"↙
      ↪  ,138]
4     print(animals_in_zoo)
5     animals_in_zoo.sort()
```

Output

```
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    animals_in_zoo.sort()
TypeError: '<' not supported between instances of 'int' and 'str'
['Elephant', 'Zebra', 'Kinkajou', 'Serval', 138]
```

We can solve this by using try,except.

7.0.11 nested lists

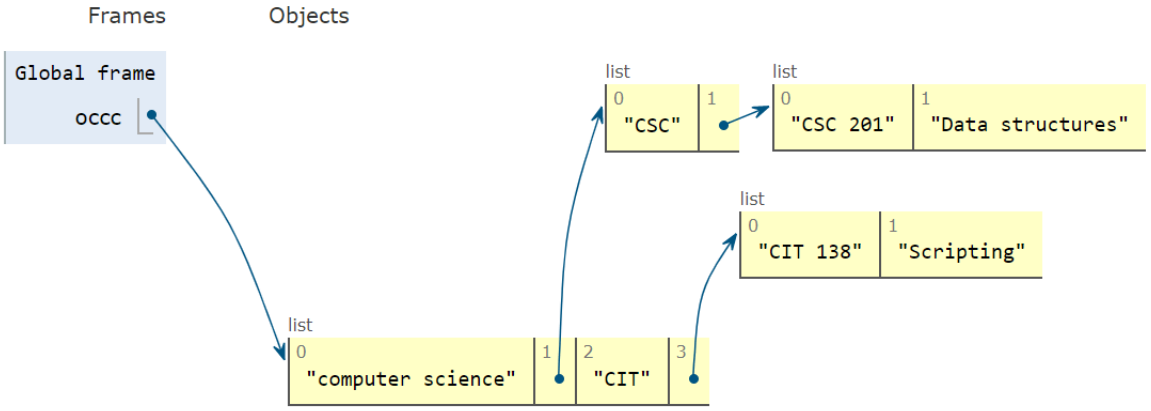
Lists Can Be Nested
You have seen that an element in a list can be any sort of object. That includes another list. A list can contain sublists, which in turn can contain sublists themselves, and so on to arbitrary depth.

nested lists

```
1  #!/usr/bin/python3
2
3  if __name__ == '__main__':
4      occc = ['computer science',
5                  ['CSC',
6                         ['CSC 201', 'Data structures']
7                  ], 'CIT', ['CIT 138', 'Scripting']
8              ]
9
10     print("occc = {0}".format(occc))
11     print("occc[0] = {0} ".format(occc[0]))
12     print("occc[1] = {0}".format(occc[1]))
13     print("occc[1][0] = {0}".format(occc[1][0]))
14     print("occc[1][1] = {0}".format(occc[1][1]))
15     print("occc[1][1][0] = {0}".format(occc[1][1][0]))
```

Output

```
occc@occc-VirtualBox:~/labs$ ./lists.py
occc =['computer science', ['CSC', ['CSC 201', 'Data structures']], 'CIT', ['CIT 138
occc[0] = computer science
occc[1] = ['CSC', ['CSC 201', 'Data structures']]
occc[1][0] = CSC
occc[1][1] = ['CSC 201', 'Data structures']
occc[1][1][0] = CSC 201
occc@occc-VirtualBox:~/labs$
```



7.1 Tuples

Tuples are identical to lists in all respects, except for the following properties:

- Tuples are defined by enclosing the elements in parentheses (()) instead of square brackets ([]).
- Tuples are immutable.
- Tuples are ordered. Meaning that the items have a defined order, and that order will not change.

• Tuples allow duplicates.
• Tuples are indexed. Each item has index like a list `[0]` is first item.
• we can use slices in tuples like we did in lists.
Why use a tuple instead of a list?
• Program execution is faster when manipulating a tuple than it is for the equivalent list. (This is probably not going to be noticeable when the list or tuple is small.)

• Sometimes you don't want data to be modified. If the values in the collection are meant to remain constant for the life of the program, using a tuple instead of list guards against accidental modification.

Tuple Assignment, Packing, and Unpacking

```
>>> t = ('csc','cit','css')
>>> t
('csc', 'cit', 'css')
>>> (x,y,z) = t
>>> x
'csc'
>>> y
'cit'
>>> z
'css'
>>>
```

No Enclosing Delimiters

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples.

tuples

```
#!/usr/bin/python3
if __name__ == '__main__':
    x,y = "occc", 138
    print("x,y = ",x,y)
    z = "occc", 138
    print("z = ",z)
    mynum = 1,000,000
    print("mynum = " ,mynum)
    print("mynum[0] = " ,mynum[0])
    print("len of tuple = ",len(mynum))
    print("max of tuple = ",max(mynum))
    print("min of tuple = ",min(mynum))

    mult = z*4
    print("multiply tuple = ",mult)
    concat = z + mynum
    print("concat = ", concat)
    # nested tuple
    my_tuple = ("occc", [8, 4, 6], (1, 2, 3))
    print(my_tuple)
    print("count = ",mynum.count(0))
    print("index = ", z.index("occc"))
    #tuple with one item needs to have a comma after the item
    orange = ("occc",)
    orange2 = ("occc")
    print("type for orange = ", type(orange))
    print("type for orange2 = ",type(orange2))
```

Output

```
x,y =  occc 138
z =  ('occc', 138)
mynum =  (1, 0, 0)
mynum[0] =  1
len of tuple =  3
max of tuple =  1
min of tuple =  0
multiply tuple =  ('occc', 138, 'occc', 138, 'occc', 138, 'occc', 138)
concat =  ('occc', 138, 1, 0, 0)
('occc', [8, 4, 6], (1, 2, 3))
count =  2
index =  0
type for orange =  <class 'tuple'>
type for orange2 =  <class 'str'>
```

7.2 Sets

A set is a collection of unique elements. If we add multiple copies of the same element to a set, the duplicates will be eliminated, and we will be left with one of each element. To define a set literal, we put a comma-separated list of values inside curly brackets { and }:

- Sets can only contain unique items.
- A set itself may be modified, but the elements contained in the set must be of an immutable type.
- Sets are unordered.

sets

```
#!/usr/bin/python3
if __name__ == '__main__':
    animals_in_zoo = {'Elephant',"Zebra","Kinkajou","Serval","Elephant"}

    print(animals_in_zoo)
    #there will be only one Elephant since we cannot have duplocates
```

Output

```
occc@occc-VirtualBox:~/labs$ ./set.py
{'Elephant', 'Zebra', 'Kinkajou', 'Serval'}
occc@occc-VirtualBox:~/labs$
```

set declaration

```
# this is an empty dictionary
a = {}   # we will get to dictionaries later!!

# this is how we make an empty set
b = set()
```

set immutable

```
#!/usr/bin/python3
if __name__ == '__main__':
    myset = {"occc", 138, "Orange", "occc"}
    print(myset)
    #sets can contain tuples since they are immutable
    myset2 = {(1,'occc',2),"Orange", 138}
    print(myset2)
    #lists are mutable so will generate an error
    myset3 = {[1,2,3],"orange"}
    print(myset3)
```

Output

```
{'Orange', 'occc', 138}
{(1, 'occc', 2), 'Orange', 138}
Traceback (most recent call last):
  File "main.py", line 9, in <module>
   myset3 = {[1,2,3],"orange"}
TypeError: unhashable type: 'list'
```

set operation

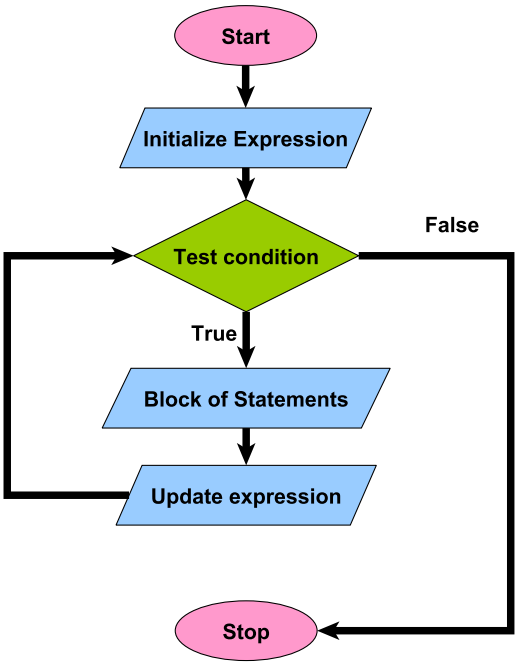
```
#!/usr/bin/python3
if __name__ == '__main__':
    even_numbers = {2, 4, 6, 8, 10}
    big_numbers = {6, 7, 8, 9, 10}

    # subtraction: big numbers which are not even
    print(big_numbers - even_numbers)

    # union: numbers which are big or even
    print(big_numbers | even_numbers)

    # intersection: numbers which are big and even
    print(big_numbers & even_numbers)

    # numbers which are big or even but not both
    print(big_numbers ^ even_numbers)
```

Output

```
occc@occc-VirtualBox:~/labs$ ./set.py
{9, 7}
{2, 4, 6, 7, 8, 9, 10}
{8, 10, 6}
{2, 4, 7, 9}
occc@occc-VirtualBox:~/labs$
```

Loops

Loops are important in Python or in any other programming language as they help you to execute a block of code repeatedly. You will often come face to face with situations where you would need to use a piece of code over and over but you don't want to write the same line of code multiple times.



8.1 loops as extension of if statements

One way we can look at while loop(one of the loop types supported by python) as a repeating if statement.
Given the following code:

simple if else

```
#!/usr/bin/python3
if __name__ == '__main__':
    x = 1          #initial expression to test against

    if(x < 6):
        print("value of x inside if statemen is: {0}".format(x))
        x = x + 1
    else:
        print("x is : {0}".format(x))
```

Output

```
value of x inside if statemen is: 1
```

Simply replacing the if keyword with while keyword yields.

if to while loop

```
#!/usr/bin/python3
if __name__ == '__main__':
    x = 1          #initial expression to test against

    while(x < 6):
        print("value of x inside while statement is: {0}".↙
            ↳  format(x))
        x = x + 1
    else:
        print("x is : {0}".format(x))
```

Output

```
value of x inside while statement is: 1
value of x inside while statement is: 2
value of x inside while statement is: 3
value of x inside while statement is: 4
value of x inside while statement is: 5
x is : 6
```

8.2 While loops

The simplest loop is the while-loop.

```
while test_expression:
    Body of while
```

simple while loop

```
#!/usr/bin/python3
if __name__ == '__main__':
    x = 1          #initial expression to test against

    while(x < 6): #repeat until the condition is false
        print(x)   #print out the test expression
        x = x + 1  #increment expression. !!!!IMPORTANT
```

Output

```
1
2
3
4
5
```

Just as **"if"** statements the body of code that will be repeated has to be **indented**. We can think of loops as if statements that will repeat over and over until the condition is false. We can use any type of test condition that we can use in an if statement.It is extremely important to do the test condition update. If we don't do this the loop condition will be true for ever and we will never exit.
This is called an **infinite loop**. We also must have a valid initial values to test against. The while loop requires relevant variables to be ready and initialized.

8.2.0.0 possible infinite loop errors

infinite loop errors

```
x = 0
while x < 3:
    y += 1 # wrong variable updated

product = 1
count = 1

while count <= 10:
    product *= count
# forgot to update count

x = 0
while x < 5:
    print(x)
x += 1 # update statement is indented one level too little, so it's outside the loop body

x = 0
while x != 5:
    print(x)
    x += 2 # x will never equal 5, because we are counting in even numbers!
```

8.2.0.0 loop control variable

Loop controlled by a counter

A common use of loops is the one in which the loop makes use of a variable (called **control variable**) that at each iteration is changed by a constant value, and whose value determines the end of the loop.
In the above example x was the control variable. It was initialized to 1 and incremented by one each time we went through the loop. It was also tested against a constant that would end the loop at a predetermined amount of iterations.
We can take advantage of this contant increment of the loop variable to perform some tasks. One wa is that we can use the loop control variable directly to perfomr some task, such as adding a summation of numbers.
If we wanted to add all the numbers from 1 to 100 we could use an accumulator and then add the content of the loop control variable to our accumulator.

summation and accumulator

```
#!/usr/bin/python3
if __name__ == '__main__':
    x = 1
    accumulator = 0
    while(x<=100):
        accumulator = accumulator + x
        x = x + 1
    print(accumulator)
```

Output

```
5050
```

squares of loop control variable

```
#!/usr/bin/python3
if __name__ == '__main__':
    x = 1
    while(x<=10):
        print('square of '+ str(x) + '  ' + str(x**2))
        x = x + 1
```

Output

```
square of 1  1
square of 2  4
square of 3  9
square of 4  16
square of 5  25
square of 6  36
square of 7  49
square of 8  64
square of 9  81
square of 10  100
```

many times if given a sequence of desired output we can use the control variable to achieve the output.
ex. 1,3,5,7,9
we can try to find a relationship between the loop control variable and the output.

loop control variable content (i)	desired output	relationship
1	1	i * 2 -1 1*2-1 = 1
2	3	i * 2 -1 2*2-1 = 3
3	5	i * 2 -1 3*2-1 = 5
4	7	i * 2 -1 4*2-1 = 7
5	9	i * 2 -1 5*2-1 = 9

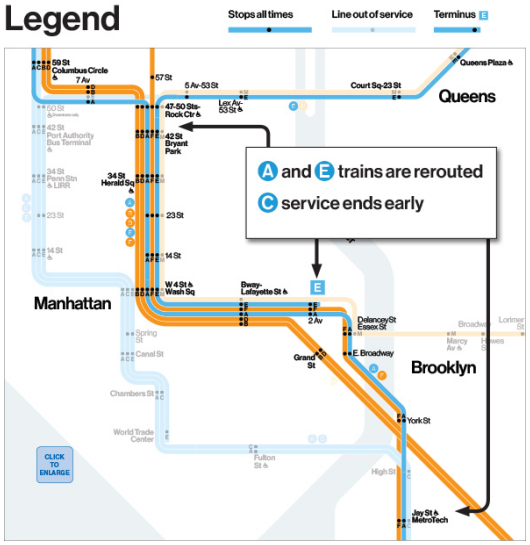
control variable and output relationship

```
#!/usr/bin/python3
if __name__ == '__main__':
    x = 1
    while(x<=5):
        print('x: '+str(x) + '  output: ' + str(x*2-1))
        x = x + 1
```

Output

```
x: 1  output: 1
x: 2  output: 3
x: 3  output: 5
x: 4  output: 7
x: 5  output: 9
```

ex2: Given a sequence of numbers: 4,14,23,34,42 can we find a relationship to a control variable? First lets try to find the next number in the sequence to see how it follows.
In this case the next sequence is 47-50. These are the train stops in Manhattan F train subway line.



There is no relationship between the numbers and the loop control variable. We will have to find some other way to generate the data. Even if we can find a relationship it may not be always the best solution to the problem. Looking back the summation can we find a better solution? The answer is yes.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

If we have a formula it is always better to use it over a loop. If no formula exists then try to find the relationship between loop control variable and the output.

8.2.0.0 priming loop and user controlled loops

Loop controlled by input

priming a while loop

```
1  #!/usr/bin/python3
2  if __name__ == '__main__':
3     try:
4        user_input = int(input("Please enter a number between 1 and 10\n"))
5        while((user_input > 10) or (user_input < 1)):
6           print("You did not input a number in the specified range. Try again\n")
7           user_input = int(input("Please enter a number between 1 and 10\n"))
8
9        print("Thank you. Your number is {0}\n".format(user_input))
10
11    except ValueError:
12       print("something went wrong. I could not undestand what you typed in\n")
```

Output

```
occc@occc-VirtualBox:~/labs$ ./first_python_script.py
Please ente a number between 1 and 10
12
You did not input a number in the specified range. Try again

Please enter a number between 1 and 10
-40
You did not input a number in the specified range. Try again

Please enter a number between 1 and 10
5
Thank you. Your number is 5

occc@occc-VirtualBox:~/labs$ ./first_python_script.py
Please ente a number between 1 and 10
occc
something went wrong. I could not undestand what you typed in

occc@occc-VirtualBox:~/labs$
```

user_input	(user_input >10)	(user_input <1)	(user_input >10) or (user_input <1)	try except
5	False	False	False	Except not invoked
-40	False	True	True	Except not invoked
12	True	False	True	Except not invoked
occc	Never test because of error	Never test because of error	Never test because of error	**Except is invoked**

When the number is between 1 and 10 both of the test conditions are false and just as with an "if" statement we never execute the code. We will only execute the code if the condition is True

8.2.1 break statement

With the break statement we can stop the loop even if the while condition is true:

loop break statement

```
1  #!/usr/bin/python3
2  if __name__ == '__main__':
3      print("While loop with no break statement\n")
4      i = 1
5      while i < 6:
6          print(i)
7          i += 1
8      print("While loop with a break statement\n")
9      i = 1
10     while i < 6:
11         print(i)
12         if i == 3:
13             break  #this will stop the loop
14         i += 1
```

Output

```
While loop with no break statement

1
2
3
4
5
While loop with a break statement

1
2
3
```

Even though we can do this and many programs incorporate the break statement it is generally considered a bad programming practice. With a break statement, it is impossible to mathematically or formally validate that the program is correct and it will not result in bad results such as infinite loops or wrong output produced. For most programs such validation is not usually necessary, however, in cases of medical equipment, airplane avionics, aerospace(rockets)

and nuclear power plants such formal validation is necessary and often required by law. Break statements will invalidate such software as defective. We can always rewrite a break statement in such a way so that we don't have to invoke it.

alternate to break statement

```
#!/usr/bin/python3
if __name__ == '__main__':
    i = 1
    while i < 6:
        print(i)
        if i == 3:
          i = 100000
        i += 1
```

Output

```
While loop with no break statement
1
2
3
```

8.2.2 The continue statement

With the continue statement we can stop the current iteration, and continue with the next:

continue statement

```
#!/usr/bin/python3
i = 0
while i < 9:
    i = i + 1
    if (i > 3):
        if (i < 7):
            continue  #force to go back to top of loop
    print(i)
```

Output

```
While loop with no break statement
1
2
3
7
8
9
```

8.3 Range Function

The most general syntax is

range(start , pastEnd , step) With the range function we can generate a sequence of numbers in the specified range including stepping.

```
>>> numbers = range(1,10)
>>> print(list(numbers))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> numbers = range(1,10,3)
>>> print(list(numbers))
[1, 4, 7]
>>> numbers = range(10,1,-1)
>>> print(list(numbers))
[10, 9, 8, 7, 6, 5, 4, 3, 2]
>>>
```

Range can take literal as well as variables for the start and end of the range
Range takes one to three arguments

A **start** argument is a starting number of the sequence. i.e., lower limit. By default, it starts with **0** if not specified.

A **pastEnd** argument is an upper limit. i.e., generate numbers up to this number, The range() function **doesn't** include this number in the result.

The **step** is a difference between each number in the result. The default value of the step is 1 if not specified.

range example

```
#!/usr/bin/python3
if __name__ == '__main__':
    x = 5
    print(list(range(x)))
    print(list(range(1,x)))
    print(list(range(-10,x,2)))
    #we can even go backwards
    print(list(range(x, -10, -1)))
```

Output

```
[0, 1, 2, 3, 4]
[1, 2, 3, 4]
[-10, -8, -6, -4, -2, 0, 2, 4]
[5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

8.4 in keyword

The **in** keyword has two purposes:

The in keyword is used to check if a value is present in a sequence (list, range, string, etc.).

The in keyword is also used to iterate through a sequence in a for loop:

in keyword

```
#!/usr/bin/python3
if __name__ == '__main__':
   if('A' in ('B','C','D')):
      print("A is in the tuple\n")
   else:
      print("A is not in the tuple\n")
```

Output

```
A is not in the tuple
```

in keyword examples

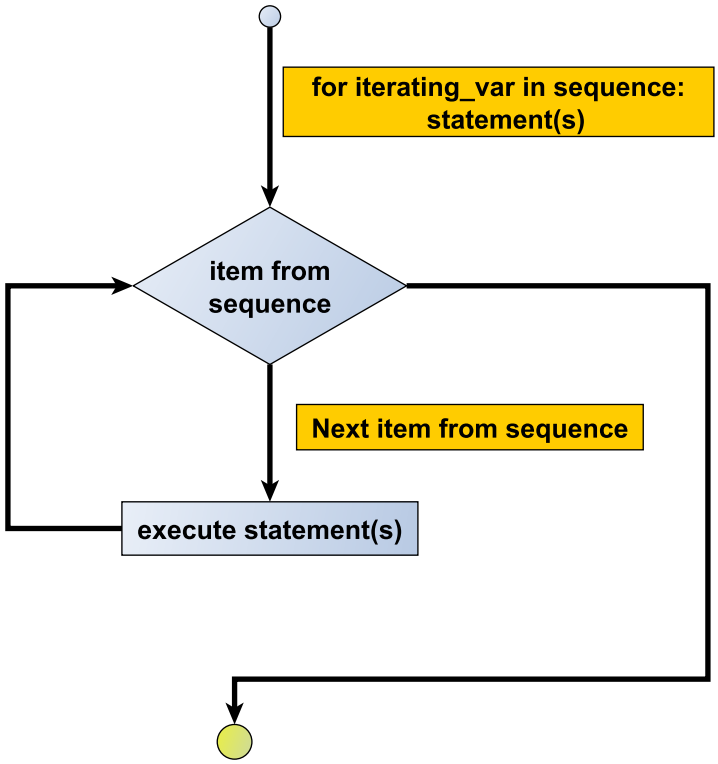
```
#!/usr/bin/python3
if __name__ == '__main__':
   numbers = range(1, 10)
   print(numbers)
   if (42 in numbers):
      # in order to print out the above range we must convert
         ↪ it first
      # Converting integer list to string list
      s = [str(i) for i in numbers]

      print("42 is in {0}\n".format(s))
   else:
      # in order to print out the above range we must convert
         ↪ it first
      # Converting integer list to string list
      s = [str(i) for i in numbers]

      print("42 does not exist in the list {0}\n".format(s))
```

Output

```
occc@occc-VirtualBox:~/labs$ ./in.py
range(1, 10)
42 does not exist in the list ['1', '2', '3', '4', '5', '6', '7', '8', '9']
occc@occc-VirtualBox:~/labs$
```

8.5 for loops

There are two types of iteration:
Definite iteration, in which the number of repetitions is specified explicitly in advance
Indefinite iteration, in which the code block executes until some condition is met
In Python, indefinite iteration is performed with a while loop.
Definite iteration is done by a **for** loop.



8.5.1 Numeric Range Loop

The most basic for loop is a simple numeric range statement with start and end values.

```
for i = 1 to 10:
   <loop body>
```

for loop range

```
#!/usr/bin/python3
if __name__ == '__main__':
    for i in range(1,11):
        print("i = {0}".format(i), end=', ')
    print("")
    for i in range(-10,1):
        print("i = {0}".format(i), end=', ')
    print("")
```

Output

```
occc@occc-VirtualBox:~/labs$ ./for.py
i = 1, i = 2, i = 3, i = 4, i = 5, i = 6, i = 7, i = 8, i = 9, i = 10,
i = -10, i = -9, i = -8, i = -7, i = -6, i = -5, i = -4, i = -3, i = -
2, i = -1, i = 0,
occc@occc-VirtualBox:~/labs$
```

8.5.2 Collection-Based loops

This type of loop iterates over a collection of objects, rather than specifying numeric values or conditions:

```
for <var> in <iterable>:
   <statement(s)>
```

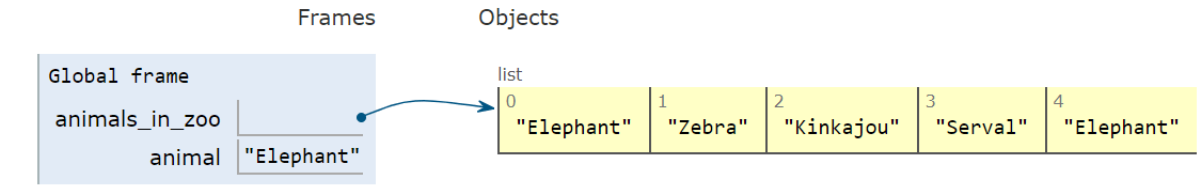
<iterable> is a collection of objects—for example, a list or tuple. The <statement(s)> in the loop body are denoted by indentation, as with all Python control structures, and are executed once for each item in <iterable>. The loop variable <var> takes on the value of the next element in <iterable> each time through the loop.

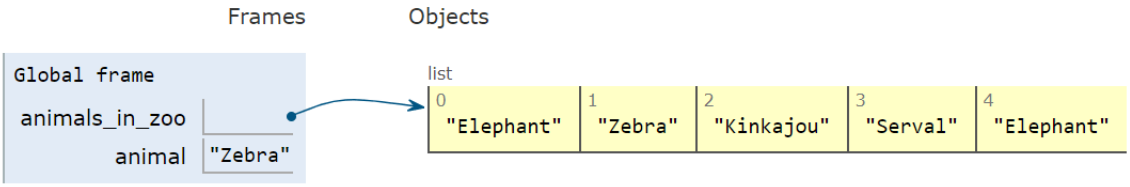
for loop iterate

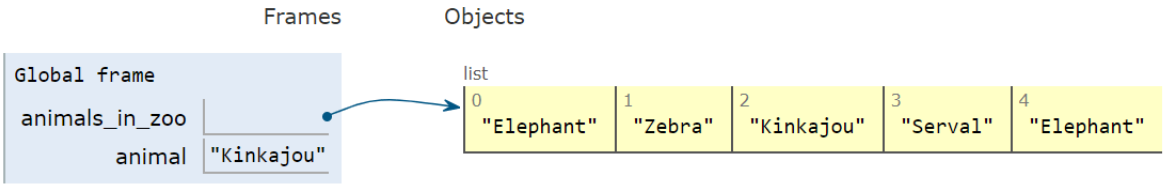
```
#!/usr/bin/python3
if __name__ == '__main__':
    animals_in_zoo = ['Elephant',"Zebra","Kinkajou","Serval","↙
        ↳  Elephant"]

    for animal in animals_in_zoo:
        print("{0}".format(animal))
```

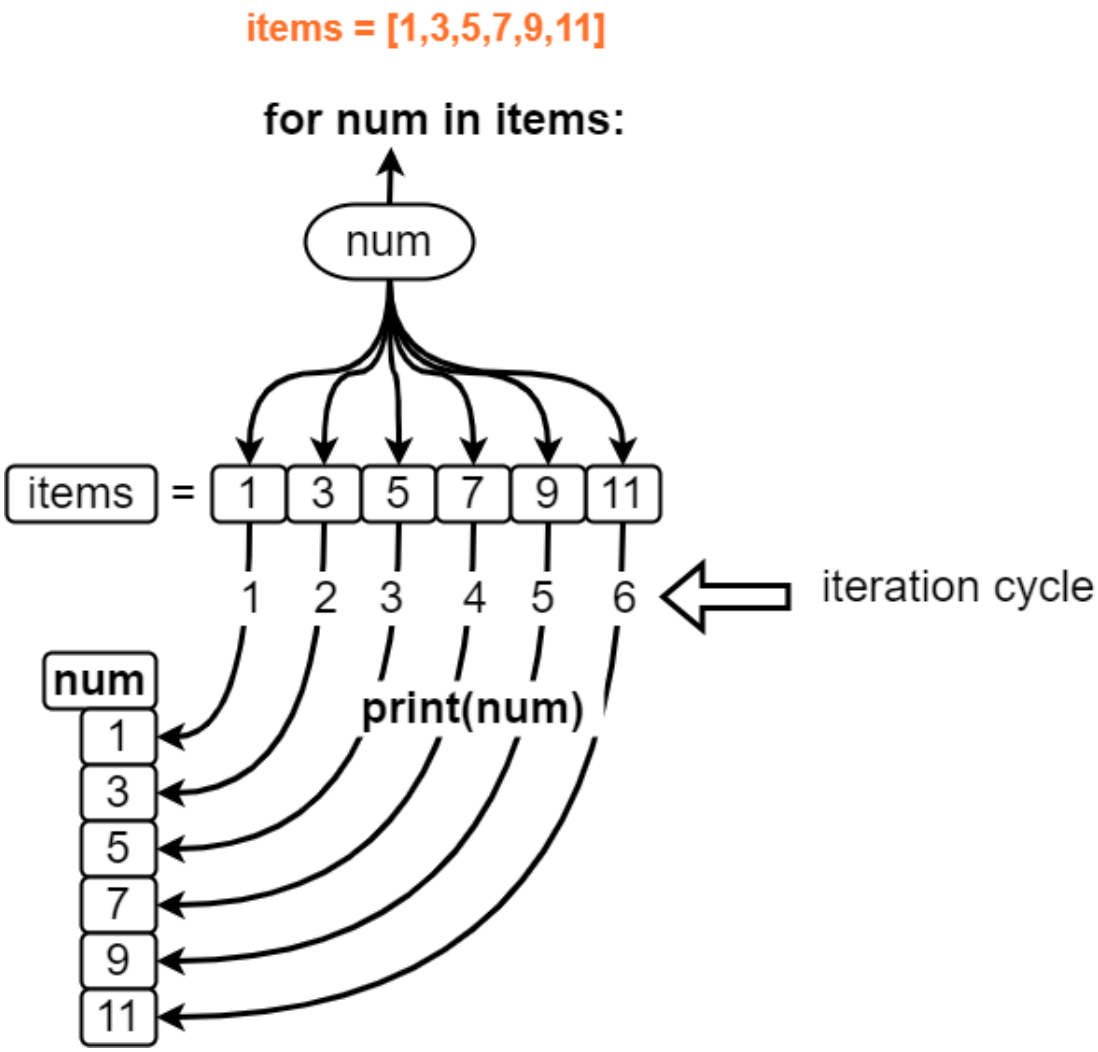
Output

```
occc@occc-VirtualBox:~/labs$ ./for.py
Elephant
Zebra
Kinkajou
Serval
Elephant
occc@occc-VirtualBox:~/labs$
```







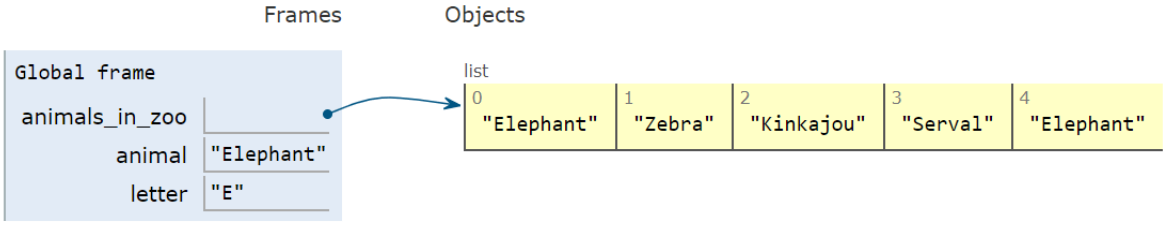


iterate over string

```
1 #!/usr/bin/python3
2 if __name__ == '__main__':
3     animals_in_zoo = ['Elephant',"Zebra","Kinkajou","Serval","↙
      ↪  Elephant"]
4     animal = animals_in_zoo[0]  #get first element in list
5     for letter in animal:       #iterate over each letter in ↙
      ↪  the string
6       print("{0}".format(letter))
```

Output

```
E
l
e
p
h
a
n
t
```



8.5.3 Iterables

In Python, **iterable** means an object can be used in the iteration.
If an object is an iterable, it can be passed to the built-in Python function iter(), which returns something called an iterator. Each of the objects in the following example is an iterable and returns some type of iterator when passed to iter():

```
>>> iter("occc")                                    #String
<str_iterator object at 0x7fb170c566a0>
>>> iter(['cit138','csc201'])                       #List
<list_iterator object at 0x7fb170c56748>
>>> iter(('cit138','csc201'))                       #Tuple
<tuple_iterator object at 0x7fb170c56668>
>>> iter({'cit138','csc201'})                       #Set
<set_iterator object at 0x7fb170c57af8>
>>> iter({'cit138':'scripting','csc201':'Data structures'})    #↙
    ↳  Dictionary
<dict_keyiterator object at 0x7fb1729fe098>
>>>
```

```
>>> iter(42)                                        #integer
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
>>> iter(3.14)                                      #float
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'float' object is not iterable
>>>
```

8.5.4 Iterators

Okay, now you know what it means for an object to be iterable, and you know how to use iter() to obtain an iterator from it. Once you've got an iterator, what can you do with it?

An iterator is essentially a value producer that yields successive values from its associated iterable object. The built-in function **next()** is used to obtain the next value from in iterator.

Here is an example using the same list as above:

```
>>> comp_sci = ['cit138','csc201']
>>> itr = iter(comp_sci)
>>> itr
<list_iterator object at 0x7fb1705a0438>
>>> next(itr)
'cit138'
>>> next(itr)
'csc201'
>>> next(itr)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
>>>
```

If all the values from an iterator have been returned already, a subsequent next() call raises a StopIteration exception. Any further attempts to obtain values from the iterator will fail.

You can only obtain values from an iterator in one direction. You can't go backward. There is no prev() function.

If you want to grab all the values from an iterator at once, you can use the built-in list() function. Among other possible uses, list() takes an iterator as its argument, and returns a list consisting of all the values that the iterator yielded:

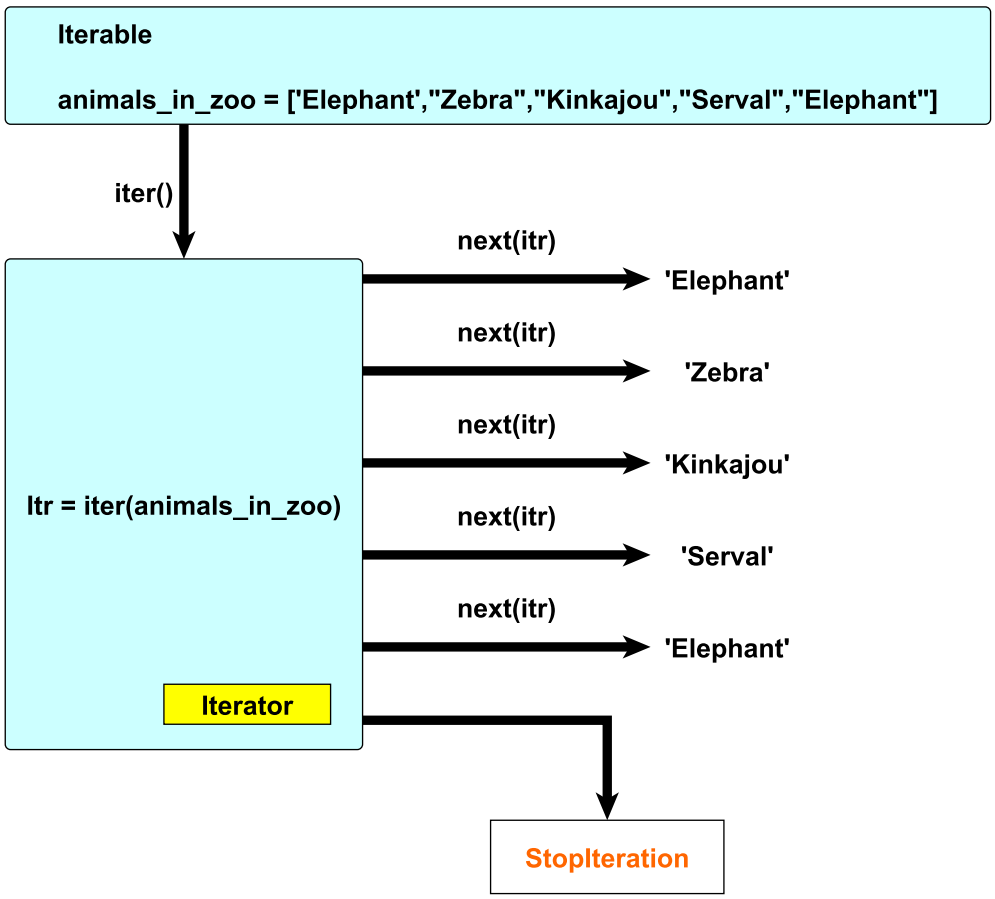
```
>>> comp_sci = ['cit138','csc201']
>>> itr = iter(comp_sci)
>>> list(itr)
['cit138', 'csc201']
>>>
```

```
animals_in_zoo = ['Elephant',"Zebra","Kinkajou","Serval","Elephant
    ↪  "]

for animal in animals_in_zoo:
  print("{0}".format(animal))
```

This loop can be described entirely in terms of the concepts you have just learned about. To carry out the iteration this for loop describes, Python does the following:

Calls iter() to obtain an iterator for a
Calls next() repeatedly to obtain each item from the iterator in turn
Terminates the loop when next() raises the StopIteration exception
The loop body is executed once for each item next() returns, with loop variable animal set to the given item for each iteration.

This sequence of events is summarized in the following diagram:



We can also user loops to construct list and other data types from scratch.

user iterator to build list

```
1 if __name__ == '__main__':
2     integers = []
3
4     for i in range(0,10):
5         integers.append(i)
6
7     print(integers)
```

Output

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

8.6 Nested loops

Just like if statements loops can be nested.

Syntax:

```
for iterating_var in sequence:
   for iterating_var in sequence:
      statements(s)
   statements(s)
```

nested for loops

```
#!/usr/bin/python3
if __name__ == '__main__':
   start = 1     #star or our range
   end = 5       #end of our range

   for i in range(start,end):    #loop that will go 1 to 4 and
      ↪  store each iteration in i
      print("outer loop iteration {0}".format(i))
      for j in range(i,end):    # start of loop will be i,
         ↪  dependent on outer loop
         print(i, j)
```

Output

```
occc@occc-VirtualBox:~/labs$ ./for.py
outer loop iteration 1
1 1
1 2
1 3
1 4
outer loop iteration 2
2 2
2 3
2 4
outer loop iteration 3
3 3
3 4
outer loop iteration 4
4 4
occc@occc-VirtualBox:~/labs$
```

nested while loops

```
#!/usr/bin/python3
if __name__ == '__main__':
   i = 1
   j = 5
   while i < 4:
      while j < 8:
         print(i, ",", j)
         j = j + 1
         i = i + 1
```

Output

```
occc@occc-VirtualBox:~/labs$ ./while.py
1 , 5
2 , 6
3 , 7
occc@occc-VirtualBox:~/labs$
```

8.7 Enumerate

The **enumerate()** method adds counter to an iterable and returns it (the enumerate object).

enumerate() Parameters
The enumerate() method takes two parameters:

iterable - a sequence, an iterator, or objects that supports iteration
start (optional) - enumerate() starts counting from this number.
If start is omitted, 0 is taken as start.

Return Value from enumerate()

The enumerate() method adds counter to an iterable and returns it. The returned object is a enumerate object.

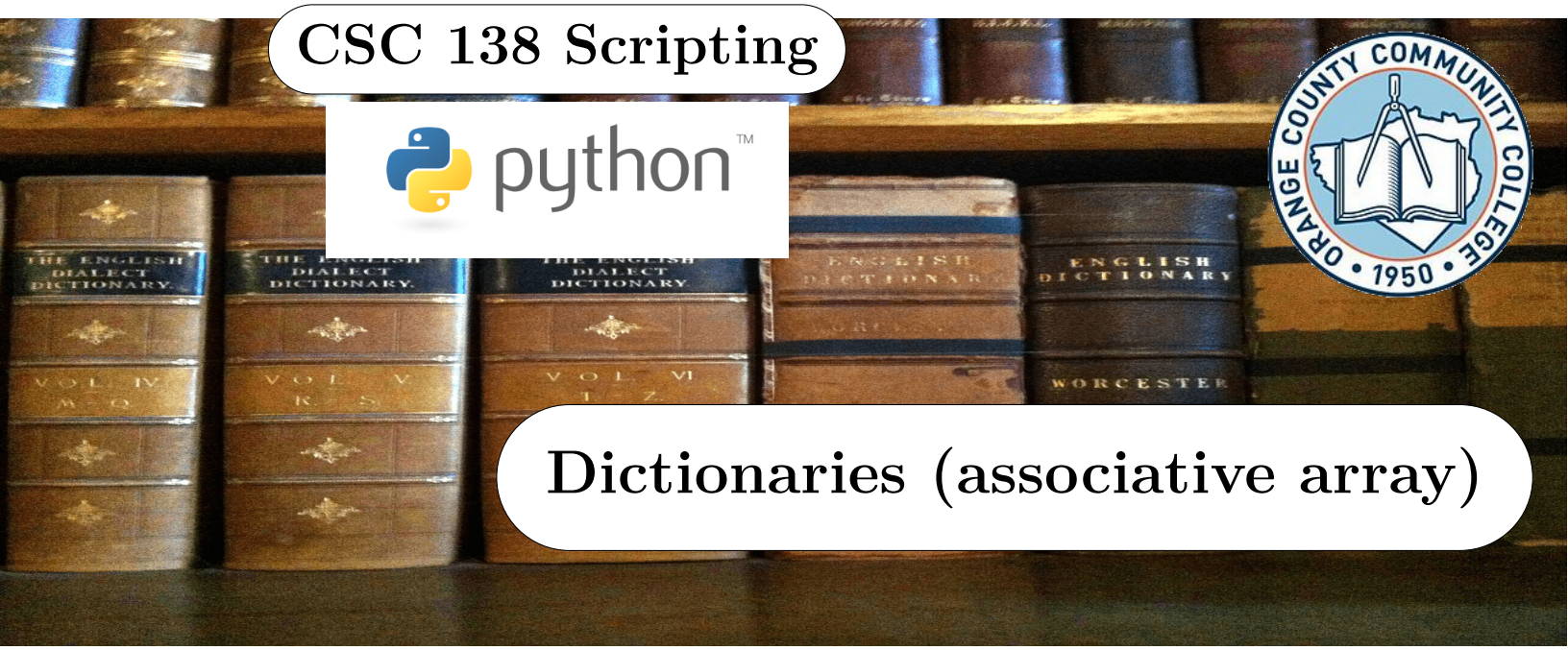
You can convert enumerate objects to list and tuple using list() and tuple() method respectively.

enumerate

```
1  #!/usr/bin/python3
2  # Python program to illustrate
3  # enumerate function in loops
4  if __name__ == '__main__':
5     animals_in_zoo = ['Elephant',"Zebra","Kinkajou","Serval","↙
       ↪  Elephant"]
6
7     # printing the tuples in object directly
8     for animal in enumerate(animals_in_zoo):
9        print("Full animal tuple: ({0:<2},{1:<10}), counter: ↙
          ↪  {2:<20}".format(animal[0],animal[1],animal[0]))
10    print()
11    # changing index and printing separately
12    for count,ele in enumerate(animals_in_zoo,100):
13       print (count,ele)
```

Output

```
Full animal tuple: (0 ,Elephant  ), counter: 0
Full animal tuple: (1 ,Zebra     ), counter: 1
Full animal tuple: (2 ,Kinkajou  ), counter: 2
Full animal tuple: (3 ,Serval    ), counter: 3
Full animal tuple: (4 ,Elephant  ), counter: 4

100 Elephant
101 Zebra
102 Kinkajou
103 Serval
104 Elephant
```



Dictionaries (associative array)

Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a **"key : value"** pair.
Dictionaries are optimized to retrieve values when the key is known.
How to create a dictionary?
Creating a dictionary is as simple as placing items inside curly braces separated by a comma.

An item has a key and the corresponding value expressed as a pair, key: value.

While values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be **unique** .

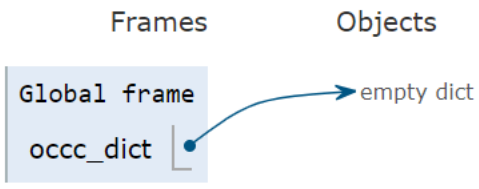
dictionary example 1

```python
#!/usr/bin/python3
if __name__ == '__main__':
    occc_dict = {}  # empty dictionary

    # Dictionary with integers as keys
    occc_dict_int = {1: "CIT",
                     2: "CSC",
                     3: "CSS"}

    occc_dict_mixed = {"name": "Bob",
                       1: ["CSC", "CIT"]}

    # using dict()
    occc_dict_temp = dict({1: "CIT",
                           2: "CSC",
                           3: "CSS"})
    # converting list of tuples to dictionary
    occc_dict_temp2 = dict([(1, 'apple'), (2, 'ball')])

    print(occc_dict)
    print(occc_dict_int)
    print(occc_dict_mixed)
    print(occc_dict_temp)
    print(occc_dict_temp2)
```

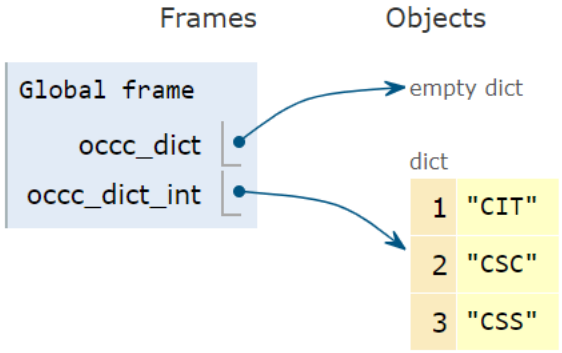
Output

```
occc@occc-VirtualBox:~/labs$ ./dictionary.py
{}
{1: 'CIT', 2: 'CSC', 3: 'CSS'}
{'name': 'Bob', 1: ['CSC', 'CIT']}
{1: 'CIT', 2: 'CSC', 3: 'CSS'}
{1: 'apple', 2: 'ball'}
occc@occc-VirtualBox:~/labs$
```

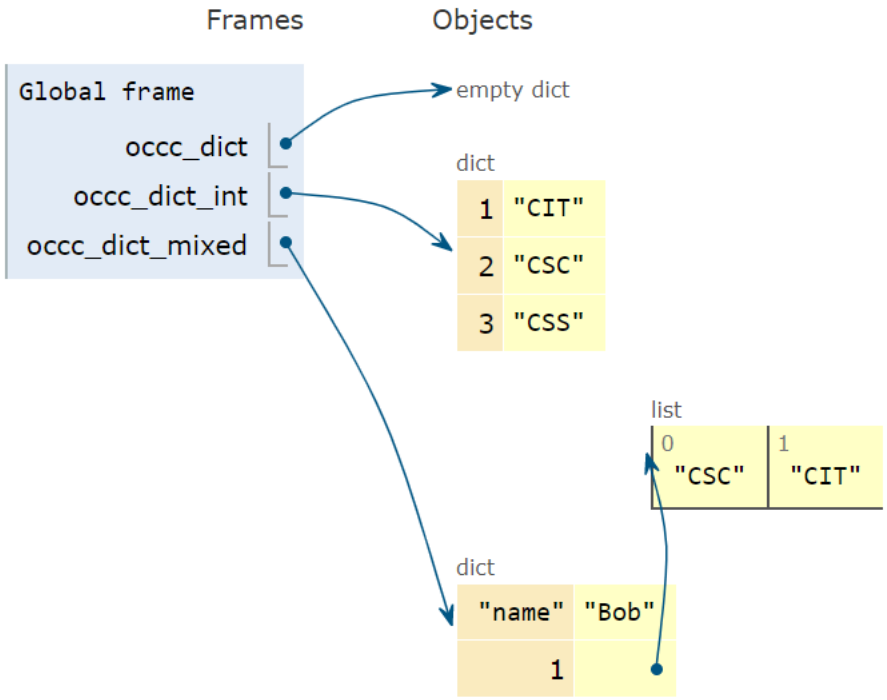
occc_dict = {}



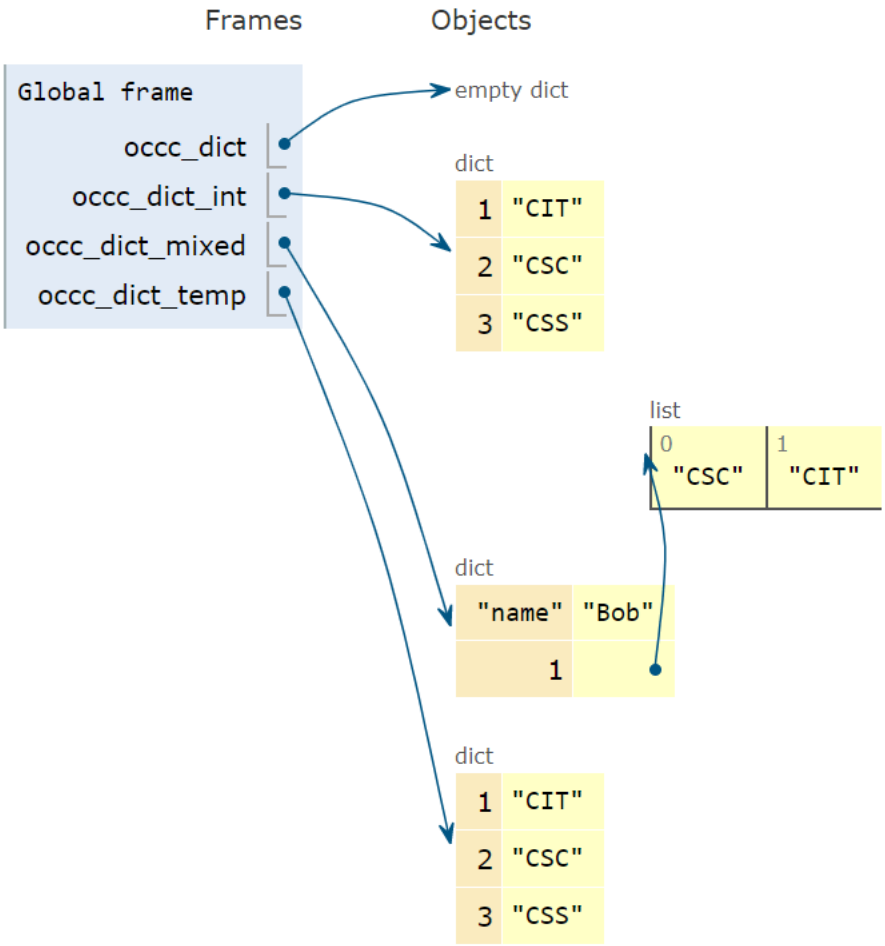
```
occc_dict = {}

occc_dict_int = {1: ”CIT”, 2: ”CSC”, 3: ”CSS”}
```



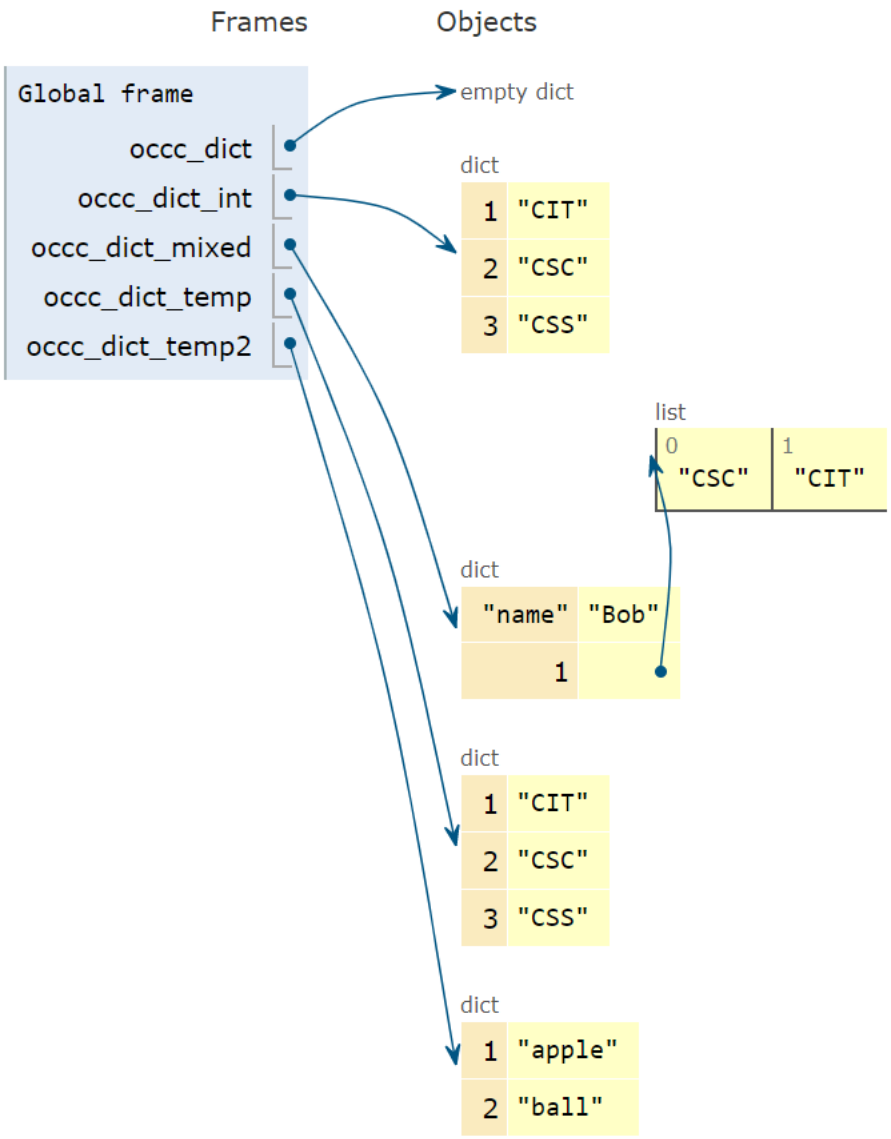

```
occc_dict = {}
occc_dict_int = {1: ”CIT”, 2: ”CSC”, 3: ”CSS”}
occc_dict_mixed = {”name”: ”Bob”, 1: [”CSC”, ”CIT”]}
```




```
occc_dict = {}
occc_dict_int = {1: ”CIT”, 2: ”CSC”, 3: ”CSS”}
occc_dict_mixed = {”name”: ”Bob”, 1: [”CSC”, ”CIT”]}
occc_dict_temp = dict({1: ”CIT”, 2: ”CSC”,3: ”CSS”})
```



occc_dict = {}
occc_dict_int = {1: ”CIT”, 2: ”CSC”, 3: ”CSS”}
occc_dict_mixed = {”name”: ”Bob”, 1: [”CSC”, ”CIT”]}
occc_dict_temp = dict({1: ”CIT”, 2: ”CSC”,3: ”CSS”})
occc_dict_temp2 = dict([(1, ’apple’), (2, ’ball’)])



9.1 Accessing data in a dictionary

While indexing is used with other container types to access values, dictionary uses keys. Key can be used either inside square brackets or with the get() method.

The difference while using get() is that it returns None instead of KeyError, if the key is not found.

rerieving items from a dictionary

```
#!/usr/bin/python3
if __name__ == '__main__':
    occc = {"CIT138": "Python scripting",
            "CSC101": "Computer science 1",
            "CSC102": "Computer science 2"}

    # using square brackets []
    print(occc["CIT138"])
    # using get
    print(occc.get("CSC101"))
```

Output

```
occc@occc-VirtualBox:~/labs$ ./dictionary.py
Python scripting
Computer science 1
occc@occc-VirtualBox:~/labs$
```

try except with dictionary

```
#!/usr/bin/python3
if __name__ == '__main__':
    occc = {"CIT138":"Python scripting",
            "CSC101":"Computer science 1",
            "CSC102":"Computer science 2"}
    print(occc)
    # Trying to access keys which doesn't exist throws error
    # occc.get('address')
    # occc['address']
    try:
        temp = occc["CIT116"]
    except KeyError:
        print("key not found")
```

Output

```
{'CIT138': 'Python scripting', 'CSC101': 'Computer science 1', 'CSC102': 'Computer s
key not found
```

9.2 How to change or add elements in a dictionary?

modify dictionaries

```
#!/usr/bin/python3
if __name__ == '__main__':
    occc = {"CIT138":"Python scripting",
            "CSC101":"Computer science 1",
            "CSC102":"Computer science 2"}
    print(occc)
    #update a value

    occc["CIT138"] = "PYTHON"

    #adding a value
    occc["CIT116"] = "Networking 1"

    print(occc["CIT138"])
    print(occc["CIT116"])
    print(occc)
```

Output

```
occc@occc-VirtualBox:~/labs$ ./dictionary.py
{'CIT138': 'Python scripting', 'CSC101': 'Computer science 1', 'CSC102': 'Computer s
PYTHON
Networking 1
{'CIT138': 'PYTHON', 'CSC101': 'Computer science 1', 'CSC102': 'Computer science 2',
'CIT116': 'Networking 1'}
occc@occc-VirtualBox:~/labs$
```

9.3 Unique keys in a dictionary

Keys have to be unique. If we create another key with the same value it will overwrite.
When we look at the above example we can see that the value was changed. Adding a non-unique key will simply overwrite the existing values for that key.

9.4 How to delete or remove elements from a dictionary?

We can remove a particular item in a dictionary by using the method **pop()**. This method removes an item with the provided key and returns the value.

The method, **pop() item** can be used to remove and return an arbitrary item (key, value) from the dictionary. All the items can be removed at once using the **clear()** method.

We can also use the **del** keyword to remove individual items or the entire dictionary itself.

delete from dictionary

```
#!/usr/bin/python3
if __name__ == '__main__':
    occc = {"CIT138":"Python scripting",
            "CSC101":"Computer science 1",
            "CSC102":"Computer science 2"}
    print(occc)

    val = occc.pop("CSC102")

    print(val)
    print(occc)

    # delete a particular item
    del occc["CIT138"]

    print(occc)

    # remove all items
    occc.clear()

    # delete the dictionary itself
    del occc
```

Output

```
occc@occc-VirtualBox:~/labs$ ./dictionary.py
{'CIT138': 'Python scripting', 'CSC101': 'Computer science 1', 'CSC102': 'Computer s
Computer science 2
{'CIT138': 'Python scripting', 'CSC101': 'Computer science 1'}
{'CSC101': 'Computer science 1'}
occc@occc-VirtualBox:~/labs$
```

9.5 Python Dictionary Methods

Methods that are available with dictionary.

Dictionary functions

Method	Description
clear()	Remove all items form the dictionary.
copy()	Return a shallow copy of the dictionary.
fromkeys(seq[, v])	Return a new dictionary with keys from seq and value equal to v (defaults to None).
get(key[,d])	Return the value of key. If key does not exit, return d (defaults to None).
items()	Return a new view of the dictionary's items (key, value).
keys()	Return a new view of the dictionary's keys.
pop(key[,d])	Remove the item with key and return its value or d if key is not found. If d is not provided and key is not found, raises KeyError.
popitem()	Remove and return an arbitary item (key, value). Raises KeyError if the dictionary is empty.
setdefault(key[,d])	If key is in the dictionary, return its value. If not, insert key with a value of d and return d (defaults to None).
update([other])	Update the dictionary with the key/value pairs from other, overwriting existing keys.
values()	Return a new view of the dictionary's values

dictionary functions examples

```
#!/usr/bin/python3
if __name__ == '__main__':
    occc = {"CIT138":"Python scripting",
            "CSC101":"Computer science 1",
            "CSC102":"Computer science 2"}
    print("original occc")
    print(occc)

    occc2 = occc.copy()
    print("copy of occc")
    print(occc2)
    print("deletting from occc")
    del occc["CIT138"]
    print("original occc")
    print(occc)
    print("copy of occc")
    print(occc2)

    keys = {"CIT138","CSC101","CSC102"}
    classes =occc.fromkeys(keys)
    print("keys used in from keys {0}".format(keys))
    print("dictionary from keys")
    print(classes)
```

```
    values = "class description"
    classes =occc.fromkeys(keys,values)
    print("dictionary using from keys with a value")
    print(classes)

    val = [1]     #use a list as valuse
    classes2 = occc.fromkeys(keys,val)
    print("dictionary from keys using a list for values")
    print(classes2)
    #update val
    val.append(2)
    print("dictionary fromkeys after updating list")
    print(classes2)
    #new val showed up in origianl classes2 dictionary
    #to avaoid this use dictional comprehention
    # using fromkeys() to convert sequence to dict
    # using dict. comprehension
    dict2 = { key : list(val) for key in keys }
    print("dictionary using dictionary comprehention")
    print(dict2)
    val.append(4)
    print("diction using comprehention after list update")
    print(dict2)
    # Dictionary with three items
    occc_dict2 = { 'CIT116': 'Net 1' }

    # Dictionary before Updation
    print("Original Dictionary:")
    print(occc)

    # update the value of key 'B'
    occc.update(occc_dict2)
    print("Dictionary after updation:")
    print(occc)
    dic_values = occc.values()
```

Output

```
occc@occc-VirtualBox:~/labs$ ./dictionary.py
original occc
{'CIT138': 'Python scripting', 'CSC101': 'Computer science 1', 'CSC102': 'Computer s
copy of occc
{'CIT138': 'Python scripting', 'CSC101': 'Computer science 1', 'CSC102': 'Computer s
deletting from occc
original occc
{'CSC101': 'Computer science 1', 'CSC102': 'Computer science 2'}
copy of occc
{'CIT138': 'Python scripting', 'CSC101': 'Computer science 1', 'CSC102': 'Computer s
keys used in from keys {'CIT138', 'CSC101', 'CSC102'}
dictionary from keys
{'CIT138': None, 'CSC101': None, 'CSC102': None}
dictionary using from keys with a value
{'CIT138': 'class description', 'CSC101': 'class description', 'CSC102': 'class desc
dictionary from keys using a list for values
{'CIT138': [1], 'CSC101': [1], 'CSC102': [1]}
dictionary fromkeys after updating list
{'CIT138': [1, 2], 'CSC101': [1, 2], 'CSC102': [1, 2]}
dictionary using dictionary comprehention
{'CIT138': [1, 2], 'CSC101': [1, 2], 'CSC102': [1, 2]}
diction using comprehention after list update
{'CIT138': [1, 2], 'CSC101': [1, 2], 'CSC102': [1, 2]}
Original Dictionary:
{'CSC101': 'Computer science 1', 'CSC102': 'Computer science 2'}
Dictionary after updation:
{'CSC101': 'Computer science 1', 'CSC102': 'Computer science 2', 'CIT116': 'Net 1'}
occc@occc-VirtualBox:~/labs$
```

9.6 Testing for keys in dictionaries

testing for keys in dictionary

```
#!/usr/bin/python3
if __name__ == '__main__':
   occc = {"CIT138":"Python scripting",
           "CSC101":"Computer science 1",
           "CSC102":"Computer science 2"}
   key = "CIT138"
   if key in occc.keys():
      print("Key Present,{0} ".format(key), end =" ")
      print("value =", occc[key])
   else:
      print("Not present")

   key = "ABC"
   if key in occc.keys():
      print("Present, ", end =" ")
      print("value =", occc[key])
   else:
      print("Key Not present {0}".format(key))
   val = "123"
   if val not in occc.keys():
      print("Key not in {0}".format(val))
   val = "CIT138"
   if val in occc:
      print("Present, ", end =" ")
      print("value =", occc[val])
   else:
      print("Not present")
```

Output

```
Key Present,CIT138  value = Python scripting
Key Not present ABC
Key not in 123
Present,  value = Python scripting
occc@occc-VirtualBox:~/labs$
```

9.7 Looping over dictionaries

iterating over dictionaries

```
#!/usr/bin/python3
if __name__ == '__main__':
   occc = {"CIT138":"Python scripting",
           "CSC101":"Computer science 1",
           "CSC102":"Computer science 2"}

   for key in occc:
      print("Key: {0} , Value: {1}".format(key,occc[key]))
   print("_" * 40)

   for (key,value) in occc.items():
      print("Key: {0} , Value: {1}".format(key,value))
   print("_" * 40)

   for key in occc.keys():
      print("Key: {0} , Value: {1}".format(key,occc[key]))
```

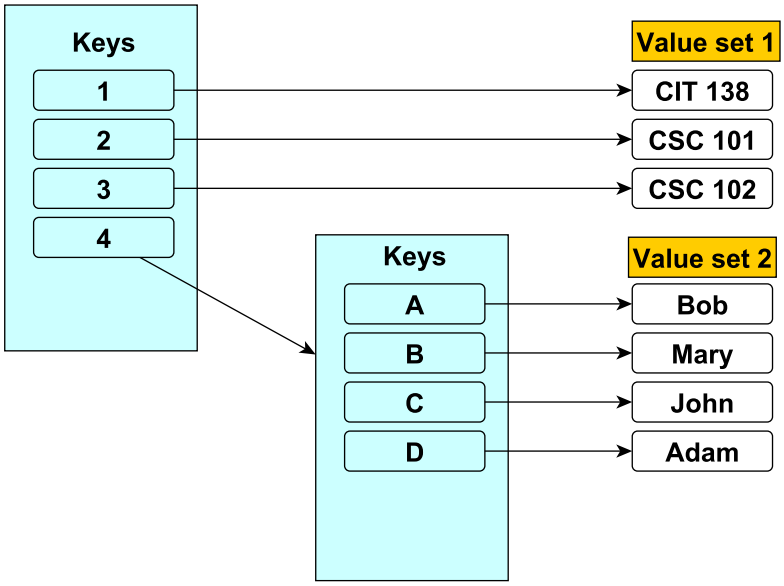
Output

```
occc@occc-VirtualBox:~/labs$ ./dictionary.py
Key: CIT138 , Value: Python scripting
Key: CSC101 , Value: Computer science 1
Key: CSC102 , Value: Computer science 2

----------------------------------------
Key: CIT138 , Value: Python scripting
Key: CSC101 , Value: Computer science 1
Key: CSC102 , Value: Computer science 2

----------------------------------------
Key: CIT138 , Value: Python scripting
Key: CSC101 , Value: Computer science 1
Key: CSC102 , Value: Computer science 2
occc@occc-VirtualBox:~/labs$
```

dictionary functions

Function	Description
all()	Return True if all keys of the dictionary are true (or if the dictionary is empty).
any()	Return True if any key of the dictionary is true. If the dictionary is empty, return False.
len()	Return the length (the number of items) in the dictionary.
cmp()	Compares items of two dictionaries.
sorted()	Return a new sorted list of keys in the dictionary.

9.8 Nested dictionaries

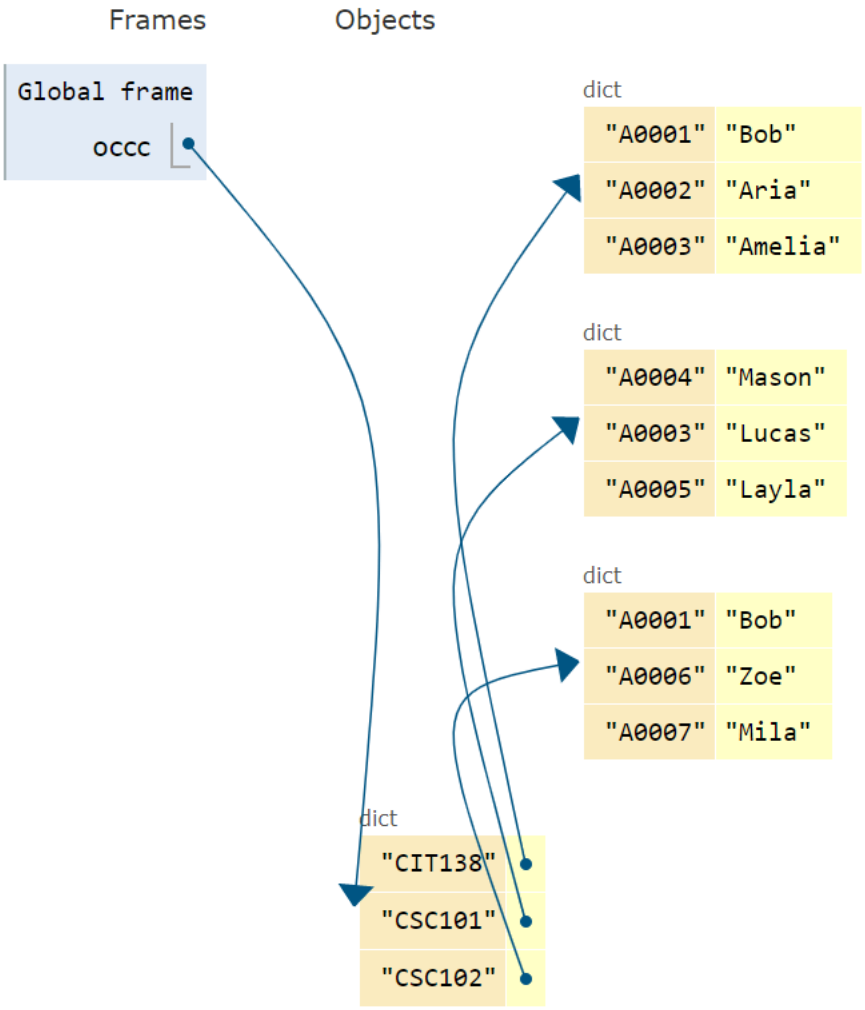


nested dictionaries

```
#!/usr/bin/python3
if __name__ == '__main__':
    occc = {"CIT138":
                    {"A0001":"Bob",
                     "A0002":"Aria",
                     "A0003":"Lucas",
                     "A0003":"Amelia"},
            "CSC101":
                    {"A0004":"Mason",
                     "A0003":"Lucas",
                     "A0005":"Layla"},
            "CSC102":
                    {"A0001":"Bob",
                    "A0006":"Zoe",
                    "A0007":"Mila"}
            }

    for key in occc:
        print("Key: {0} , Value: {1}".format(key,occc[key]))
    print("_" * 40)

    for (key,value) in occc.items():
        print("Key: {0} , Value: {1}".format(key,value))
    print("_" * 40)

    for key in occc.keys():
        print("Key: {0} , Value: {1}".format(key,occc[key]))
```

Output

```
Key: CIT138 , Value: {'A0001': 'Bob', 'A0002': 'Aria', 'A0003': 'Amelia'}
Key: CSC101 , Value: {'A0004': 'Mason', 'A0003': 'Lucas', 'A0005': 'Layla'}
Key: CSC102 , Value: {'A0001': 'Bob', 'A0006': 'Zoe', 'A0007': 'Mila'}

----------------------------------------
Key: CIT138 , Value: {'A0001': 'Bob', 'A0002': 'Aria', 'A0003': 'Amelia'}
Key: CSC101 , Value: {'A0004': 'Mason', 'A0003': 'Lucas', 'A0005': 'Layla'}
Key: CSC102 , Value: {'A0001': 'Bob', 'A0006': 'Zoe', 'A0007': 'Mila'}

----------------------------------------
Key: CIT138 , Value: {'A0001': 'Bob', 'A0002': 'Aria', 'A0003': 'Amelia'}
Key: CSC101 , Value: {'A0004': 'Mason', 'A0003': 'Lucas', 'A0005': 'Layla'}
Key: CSC102 , Value: {'A0001': 'Bob', 'A0006': 'Zoe', 'A0007': 'Mila'}
```



9.9 isinstance()

Sometimes we can get a list,dictionary,tuple,etc.etc. in the same variable. We need to be able to find the data types that are nested together.

★ isinstance() Parameters

The isinstance() takes two parameters:

object - object to be checked

classinfo - class, type, or tuple of classes and types

testing for datatype

```
#!/usr/bin/python3
if __name__ == '__main__':
   occc = {"CIT138":
               {"A0001":"Bob",
                "A0002":"Aria",
                "A0003":"Lucas",
                "A0003":"Amelia"},
           "CSC101":
               {"A0004":"Mason",
                "A0003":"Lucas",
                "A0005":"Layla"},
           "CSC102":
               {"A0001":"Bob",
                "A0006":"Zoe",
                 "A0007":"Mila"}}
   if isinstance(occc,list):
      print("occc is a list")
   elif isinstance(occc,tuple):
      print("occc is a list")
   elif isinstance(occc,str):
      print("occc is a string")
   elif isinstance(occc,dict):
      print("occc is a dictionary")
   else:
      print("occc is some other data type")
```

Output

```
occc@occc-VirtualBox:~/labs$ ./dictionary.py
occc is a dictionary
occc@occc-VirtualBox:~/labs$
```

nested dictionaries and lists

```
#!/usr/bin/python3
if __name__ == '__main__':
   occc = {"CIT138":
             {"A0001":["Bob",98,78,90],
              "A0002":["Aria",100,93,67],
              "A0003":["Lucas",78,91,89],
              "A0003":["Amelia",99,89,70]},
           "CSC101":
              {"A0004":["Mason",67,90,89],
               "A0003":["Lucas",67,76,80],
               "A0005":["Layla",45,89,56]},
           "CSC102":
             {"A0001":["Bob",90,89,92],
              "A0006":["Zoe",89,78,77],
              "A0007":["Mila",97,92,67]}}

   if isinstance(occc,dict):
      for (key,val) in occc.items():
         print("Class: {0}".format(key))
         #print(type(val))
         if isinstance(val,dict):
            for (anum,name_grades)  in val.items():
               print("   A number: {0}".format(anum))
               if isinstance(name_grades,list):
                  for item in name_grades:
                     if isinstance(item,str):
                        print("      Name: {0}".format(item))
                        print("         Grades :",end = '')
                     else:
                        print(" {0}, ".format(item),end = '')
                  print("")
```

Output

```
Class: CIT138
   A number: A0001
      Name: Bob
         Grades : 98,  78,  90,
   A number: A0002
      Name: Aria
         Grades : 100,  93,  67,
   A number: A0003
      Name: Amelia
         Grades : 99,  89,  70,
Class: CSC101
   A number: A0004
      Name: Mason
         Grades : 67,  90,  89,
   A number: A0003
      Name: Lucas
         Grades : 67,  76,  80,
   A number: A0005
      Name: Layla
         Grades : 45,  89,  56,
Class: CSC102
   A number: A0001
      Name: Bob
         Grades : 90,  89,  92,
   A number: A0006
      Name: Zoe
         Grades : 89,  78,  77,
   A number: A0007
      Name: Mila
         Grades : 97,  92,  67,
```

9.10 Other ways to print dictionaries and data

printing dictionaries

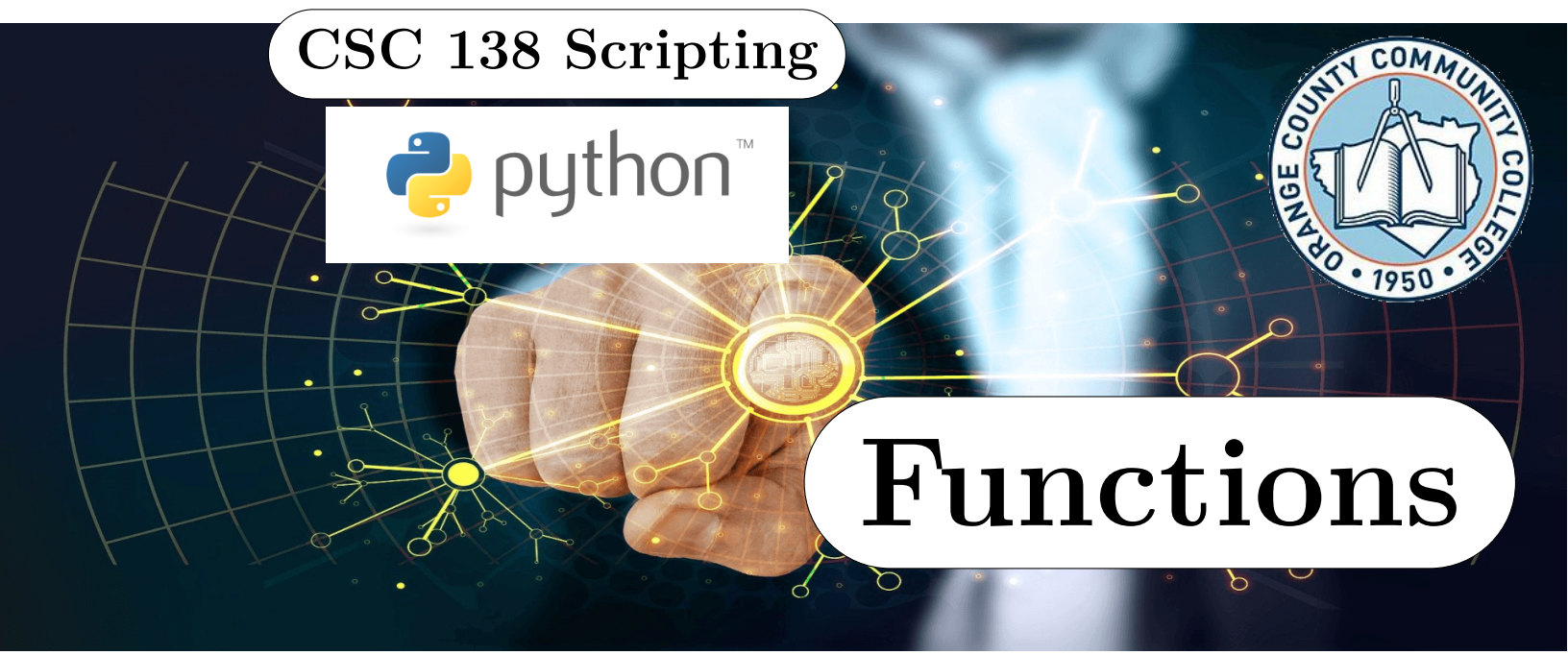
```
#!/usr/bin/python3
import pprint
import json
import yaml
from string import ascii_letters, digits, hexdigits
if __name__ == '__main__':
    occc = {"CIT138":
                {"A0001":["Bob",98,78,90],
                 "A0002":["Aria",100,93,67],
                 "A0003":["Lucas",78,91,89],
                 "A0003":["Amelia",99,89,70]},
            "CSC101":
                {"A0004":["Mason",67,90,89],
                 "A0003":["Lucas",67,76,80],
                 "A0005":["Layla",45,89,56]},
            "CSC102":
                {"A0001":["Bob",90,89,92],
                 "A0006":["Zoe",89,78,77],
                 "A0007":["Mila",97,92,67]}}


    pp = pprint.PrettyPrinter(indent=4)
    print("pretty print")
    print("_" * 50)
    pp.pprint(occc)
    print("_" * 50)

    print("Json print")
    print("_" * 50)

    print(json.dumps(occc, sort_keys=True, indent=4))
    print("_" * 50)
    print("yaml print")
    print("_" * 50)
    print(yaml.dump(occc, default_flow_style=False))
    print("_" * 50)
```

```
occc@occc-VirtualBox:~/labs$


pretty print
__________________________________________________
{   'CIT138': {   'A0001': ['Bob', 98, 78, 90],
                  'A0002': ['Aria', 100, 93, 67],
                  'A0003': ['Amelia', 99, 89, 70]},
    'CSC101': {   'A0003': ['Lucas', 67, 76, 80],
```

```
        'A0004': ['Mason', 67, 90, 89],
        'A0005': ['Layla', 45, 89, 56]},
    'CSC102': {   'A0001': ['Bob', 90, 89, 92],
        'A0006': ['Zoe', 89, 78, 77],
        'A0007': ['Mila', 97, 92, 67]}}
------------------------------------------------
Json print
------------------------------------------------
{
    "CIT138": {
        "A0001": [
        "Bob",
        98,
        78,
        90
        ],
        "A0002": [
        "Aria",
        100,
        93,
        67
        ],
        "A0003": [
        "Amelia",
        99,
        89,
        70
        ]
    },
    "CSC101": {
        "A0003": [
        "Lucas",
        67,
        76,
        80
        ],
        "A0004": [
        "Mason",
        67,
        90,
        89
        ],
        "A0005": [
        "Layla",
        45,
        89,
        56
        ]
    },
    "CSC102": {
        "A0001": [
```

```
        "Bob",
        90,
        89,
        92
        ],
        "A0006": [
        "Zoe",
        89,
        78,
        77
        ],
        "A0007": [
        "Mila",
        97,
        92,
        67
        ]
    }
}
--------------------------------------------------
yaml print
--------------------------------------------------
CIT138:
A0001:
- Bob
- 98
- 78
- 90
A0002:
- Aria
- 100
- 93
- 67
A0003:
- Amelia
- 99
- 89
- 70
CSC101:
A0003:
- Lucas
- 67
- 76
- 80
A0004:
- Mason
- 67
- 90
- 89
A0005:
- Layla
```

```
- 45
- 89
- 56
CSC102:
A0001:
- Bob
- 90
- 89
- 92
A0006:
- Zoe
- 89
- 78
- 77
A0007:
- Mila
- 97
- 92
- 67
```



A function is a **named** block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing. As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called user-defined functions.

10.1 Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

• Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).

• Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.

• The first statement of a function can be an optional statement - the documentation string of the function or docstring.

• The code block within every function starts with a colon (:) and is indented.

• The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax

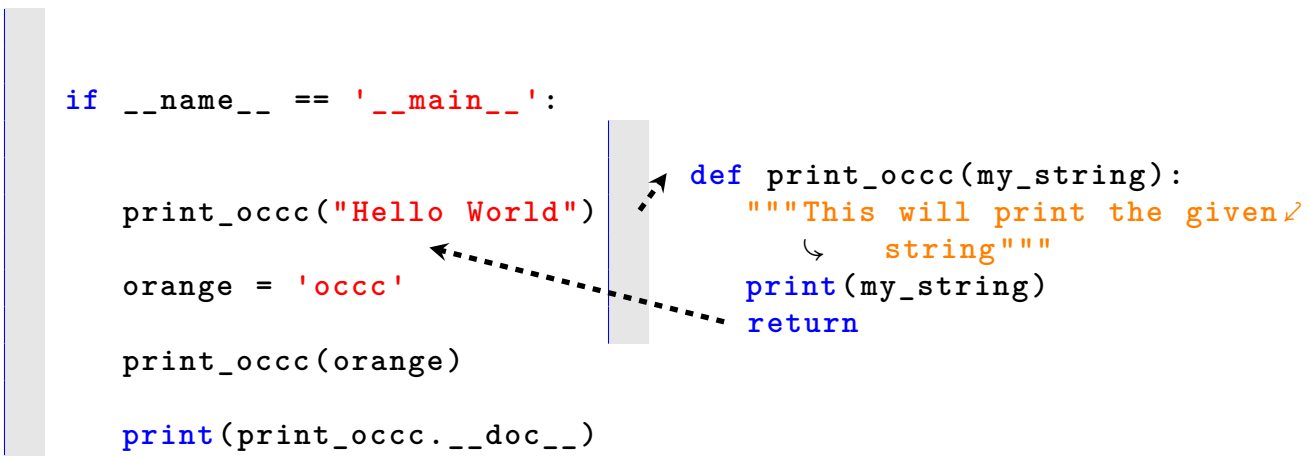
```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

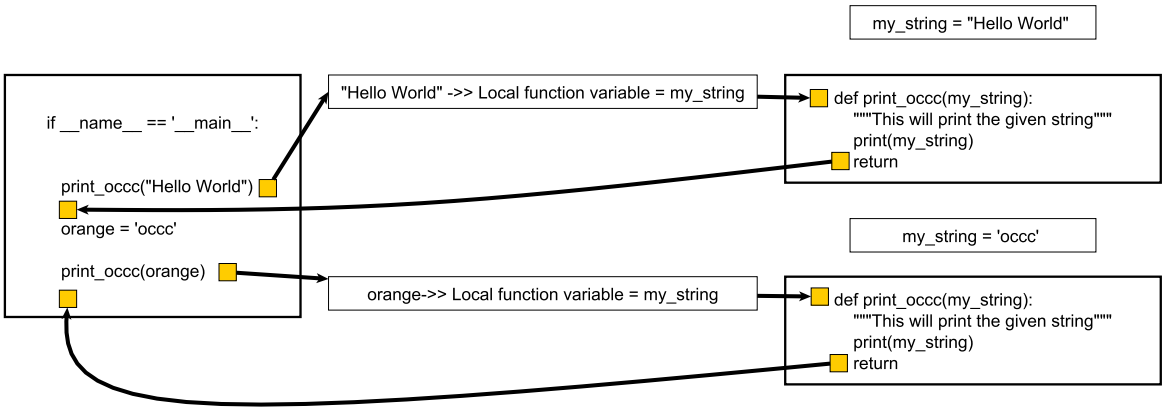
defining functions

```
1  #!/usr/bin/python3
2  #define the function. this will create a named block of code
       ↪ called print_occc
3  def print_occc(my_string):
4      """This will print the given string"""
5      print(my_string)
6      return
7  if __name__ == '__main__':
8      # we can start using the function by calling it
9      print_occc("Hello World")
10     orange = 'occc'           #variable that will be passed to
           ↪ the function
11
12     print_occc(orange)
13
14     print(print_occc.__doc__)  #print out the help information
           ↪ on the function
```

Output

```
occc@occc-VirtualBox:~/labs$ ./function.py
Hello World
occc
This will print the given string
occc@occc-VirtualBox:~/labs$
occc@occc-VirtualBox:~/labs$
```





Just like variables a function name cannot have the same name as any of the reserved keywords.

10.2 docstring

A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the **___doc___** attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

Docstring just like comments are important for us to have since it allows the programmer to find out what your code does and how to use it.

calling functions from print()

embeding function calls in print

```
1  #!/usr/bin/python3
2  def fahrenheit(T_in_celsius):
3     """ returns the temperature in degrees Fahrenheit """
4     return (T_in_celsius * 9 / 5) + 32
5
6  if __name__ == '__main__':
7     for t in (0.0, 25.8, 27.3, 29.8, 32.0):
8        print(t, "\t: ", fahrenheit(t))
9     print("using format___________________")
10    for t in (0.0, 25.8, 27.3, 29.8, 32.0):
11       print("{0} \t: {1}".format(t, fahrenheit(t)))
```

Output

```
0.0   :  32.0
25.8   :  78.44
27.3   :  81.14
29.8   :  85.64
32.0   :  89.6
using format__________________
0.0   : 32.0
25.8   : 78.44
27.3   : 81.14
29.8   : 85.64
32.0   : 89.6
```

10.3 Optional Parameters

Functions can have optional parameters, also called default parameters. Default parameters are parameters, which don't have to be given, if the function is called. In this case, the default values are used.

optional parameters

```
1  #!/usr/bin/python3
2  def fahrenheit(T_in_celsius=0.0):
3      """ returns the temperature in degrees Fahrenheit """
4      return (T_in_celsius * 9 / 5) + 32
5
6  if __name__ == '__main__':
7      x = fahrenheit(32.0)  #pass a value to function
8      y = fahrenheit()      #will use the default value of 0.0
9
10     print("x : {0}, y = {1}".format(x,y))
```

Output

```
occc@occc-VirtualBox:~/labs$ ./function.py
x : 89.6, y = 32.0
occc@occc-VirtualBox:~/labs$
```

10.3.1 Return Values

It is not mandatory to have a return statement. If we don't have the return statement the function will return a value of 'None'

return statement

```
1  #!/usr/bin/python3
2  def fahrenheit(T_in_celsius=0.0):
3     """ returns the temperature in degrees Fahrenheit """
4     return (T_in_celsius * 9 / 5) + 32
5
6  def Hello():
7     print("Hello World")
8  if __name__ == '__main__':
9    x = fahrenheit(32.0)
10   y = Hello()
11
12   print("x : {0}, y = {1}".format(x,y))
```

Output

```
occc@occc-VirtualBox:~/labs$ ./function.py
Hello World
x : 89.6, y = None
occc@occc-VirtualBox:~/labs$
```

10.4 Returning Multiple Values

return multiple items

```
1  #!/usr/bin/python3
2
3  def occc():
4     scripting = 'cit 138'
5     number_of_students = 15
6     return(scripting,number_of_students)
7
8  if __name__ == '__main__':
9     (x,y) = occc()
10    print(x)
11    print(y)
```

Output

```
occc@occc-VirtualBox:~/labs$ ./function.py
cit 138
15
occc@occc-VirtualBox:~/labs$
```

10.5 Using a list to return multiple values

return list item

```
1  #!/usr/bin/python3
2
3  def occc():
4     scripting = 'cit 138'
5     number_of_students = 15
6     return([scripting,number_of_students]) #uses a list to ↙
         ↳  return values
7
8  if __name__ == '__main__':
9     List = occc()
10    print(List)
```

Output

```
occc@occc-VirtualBox:~/labs$ ./function.py
['cit 138', 15]
occc@occc-VirtualBox:~/labs$
```

10.6 Local and Global Variables in Functions

Variable names are by default local to the function, in which they get defined.
Global variables are variables that are declared outside of any function

global variables

```
1  #!/usr/bin/python3
2  def occc():
3     print(scripting)  #will print out global variable
4
5  scripting = "CIT 138"
6  if __name__ == '__main__':
7     occc()
```

Output

```
occc@occc-VirtualBox:~/labs$ ./function.py
CIT 138
occc@occc-VirtualBox:~/labs$
```

local variables

```
#!/usr/bin/python3
def occc():
    scripting = "Python"      #local variable that exist only inside this function
                              #Takes precedence over global one
    print(scripting)

scripting = "CIT 138"
if __name__ == '__main__':
    occc()
    print(scripting)
```

Output

```
occc@occc-VirtualBox:~/labs$ ./function.py
Python
CIT 138
occc@occc-VirtualBox:~/labs$
```

confusion between local and global

```
#!/usr/bin/python3
def occc():
    print(scripting)    #will use the global variable, but another one exist here local ERROR!!!
                        #At this point the interpreter will get confused and throw an error
    scripting = "Python" #local variable that exist only inside this function
    print(scripting)    #Which one do we want?? local or global . ERROR!!!!!!!

scripting = "CIT 138"
if __name__ == '__main__':
    occc()
    print(scripting)
```

Output

```
occc@occc-VirtualBox:~/labs$ ./function.py
Traceback (most recent call last):
File "./function.py", line 9, in <module>
occc()
File "./function.py", line 3, in occc
print(scripting)    #will use the global variable, but another one exist here local
UnboundLocalError: local variable 'scripting' referenced before assignment
occc@occc-VirtualBox:~/labs$
```

global keyword

```
1  #!/usr/bin/python3
2  def occc():
3     global scripting    #any change to this variable will affect the global one
4     print(scripting)
5     scripting = "Python"
6     print(scripting)
7
8  scripting = "CIT 138"
9  if __name__ == '__main__':
10    print(scripting)
11    occc()
12    print(scripting)
```

Output

```
occc@occc-VirtualBox:~/labs$ ./function.py
CIT 138
CIT 138
Python
Python
occc@occc-VirtualBox:~/labs$
```

10.7 Arbitrary Number of Parameters

There are many situations in programming, in which the exact number of necessary parameters cannot be determined a-priori. An arbitrary parameter number can be accomplished in Python with so-called tuple references. An asterisk "*" is used in front of the last parameter name to denote it as a tuple reference.

arbitrary number of parameters

```
#!/usr/bin/python3

def arithmetic_average(first, *values):
    """ This function calculates the arithmetic average of a
        non-empty
        arbitrary number of numerical values """

    return (first + sum(values)) / (1 + len(values))
if __name__ == '__main__':
    print(arithmetic_average(42,37,189,74))
    print(arithmetic_average(8.8,78.78,453,778.73))
    print(arithmetic_average(42,36))
    print(arithmetic_average(42))
```

Output

```
occc@occc-VirtualBox:~/labs$ ./function.py
85.5
329.8275
39.0
42.0
occc@occc-VirtualBox:~/labs$
```

arbitrary number of parameters version 2

```
#!/usr/bin/python3

def arithmetic_average(*values):
    """ This function calculates the arithmetic average of a
        empty/non-empty  arbitrary number of numerical values
        """

    try:
        ret_val = sum(values) / len(values)
    except:
        ret_val = None

    return (ret_val)
if __name__ == '__main__':
    print(arithmetic_average(42,37,189,74))
    print(arithmetic_average(8.8,78.78,453,778.73))
    print(arithmetic_average(42,36))
    print(arithmetic_average(42))
    print(arithmetic_average())
```

Output

```
85.5
329.8275
39.0
42.0
None
```

10.8 Multiple returns in functions

multiple returns

```
#!/usr/bin/python3
import math
def prime(val):
   if(val == 0):
      return False,None
   if(val == 1):
      return True,1
   sq = int(math.sqrt(val))  #we only have to go upto a square
      ↪   of a number for tests
   for i in range(2,(sq+1)):
      if((val % i) == 0):
         return False,i

   return True,None

if __name__ == '__main__':
   for num in range(1,21):
      yn,div = prime(num)
      if(yn == True):
         print("Number {0} is prime".format(num))
      else:
         print("Number {0} is NOT prime divisible by
            ↪  {1},{0}/{1} = {2}".format(num,div,(num/div)))
```

Output

```
Number 1 is prime
Number 2 is prime
Number 3 is prime
Number 4 is NOT prime divisible by 2,4/2 = 2.0
Number 5 is prime
Number 6 is NOT prime divisible by 2,6/2 = 3.0
Number 7 is prime
Number 8 is NOT prime divisible by 2,8/2 = 4.0
Number 9 is NOT prime divisible by 3,9/3 = 3.0
Number 10 is NOT prime divisible by 2,10/2 = 5.0
Number 11 is prime
Number 12 is NOT prime divisible by 2,12/2 = 6.0
Number 13 is prime
Number 14 is NOT prime divisible by 2,14/2 = 7.0
Number 15 is NOT prime divisible by 3,15/3 = 5.0
Number 16 is NOT prime divisible by 2,16/2 = 8.0
Number 17 is prime
Number 18 is NOT prime divisible by 2,18/2 = 9.0
Number 19 is prime
Number 20 is NOT prime divisible by 2,20/2 = 10.0
```

external libraries and imports

Sometimes we need to have access to functions and libraries that are not part of the default Python interpreter. Since Python is extensible and comes with over 313 thousand different packages we need to incorporate these packages into our Python environment. We do this by using the "import" keyword.

11.1 import

★ Python import statement

We can import a module using **import** statement and access the definitions inside it using the dot operator. Here is an example.

import statement

```
1 #!/usr/bin/python3
2 import math
3 print("Value of pi is {0}".format(math.pi))
```

Output

```
occc@occc-VirtualBox:~/labs$ ./import.py
Value of pi is 3.141592653589793
occc@occc-VirtualBox:~/labs$
```

11.2 getting help on a module

```
1 >>> import math
2 >>> dir(math)
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
>>> math.__doc__
'This module is always available.  It provides access to the mathematical functions defined by the C standard.'
>>> help(math)

>>>
```

11.2.1 dir()

prints all the available functions in the given module.

11.2.2 help()

shows a whole help system on all the functions in a module.

read in command line parameters

```
#!/usr/bin/python3
import math   #import math function utilities
import sys    #import operating system utilities
if __name__ == '__main__':
   #try to read in command line arguments that were passed to the program
   try:
      print(sys.argv[0])  #name of our program
      print(sys.argv[1])  #first parameter
   except:
      print("you did not entered any parameters")
```

Output

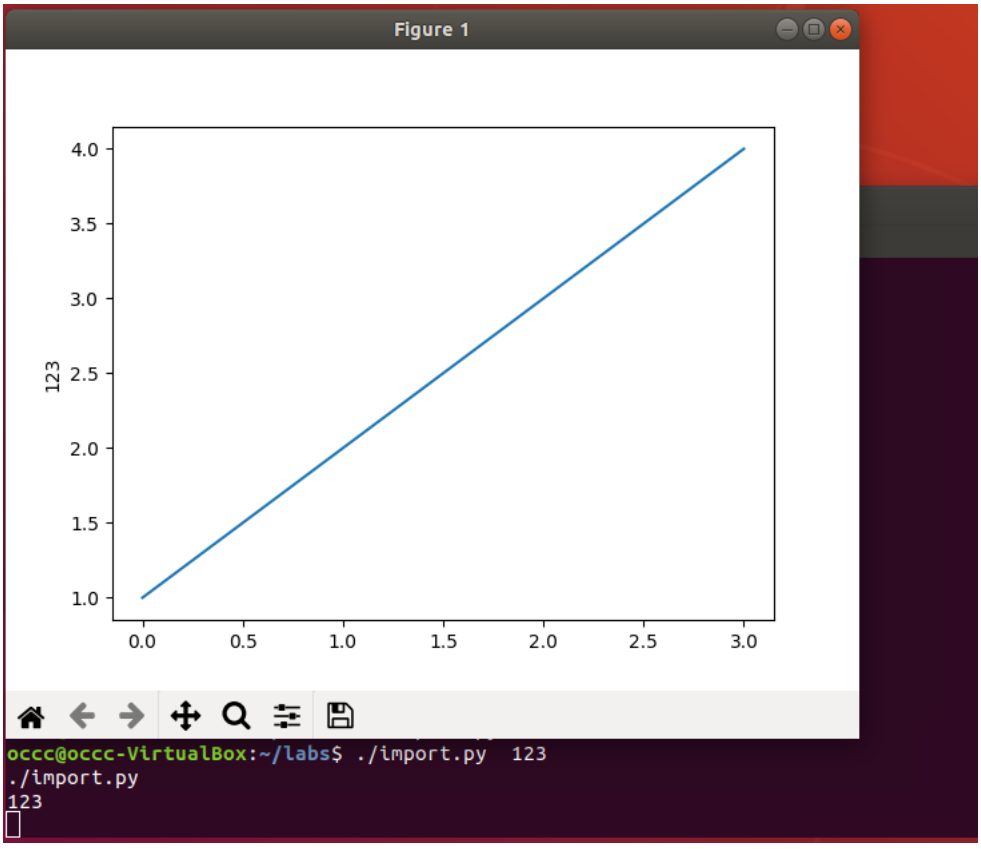
```
occc@occc-VirtualBox:~/labs$ ./import.py
./import.py
you did not entered any parameters
occc@occc-VirtualBox:~/labs$ ./import.py  123
./import.py
123
occc@occc-VirtualBox:~/labs$
```

11.3 Installing modules

```
occc@occc-VirtualBox:~/labs$ sudo pip3 install matplotlib


Collecting matplotlib
Downloading https://files.pythonhosted.org/packages/da/83/d989ee20c78117c737ab40e0318e
3.1.0-cp36-cp36m-manylinux1_x86_64.whl (13.1MB)
100% || 13.1MB 164kB/s
Collecting kiwisolver>=1.0.1 (from matplotlib)
Downloading https://files.pythonhosted.org/packages/f8/a1/5742b56282449b1c0968197f63ea
1.1.0-cp36-cp36m-manylinux1_x86_64.whl (90kB)
100% || 92kB 6.1MB/s
Collecting pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 (from matplotlib)
Using cached https://files.pythonhosted.org/packages/dd/d9/3ec19e966301a6e25769976999b
2.4.0-py2.py3-none-any.whl
Collecting python-dateutil>=2.1 (from matplotlib)
Using cached https://files.pythonhosted.org/packages/41/17/c62faccbfbd163c7f57f3844689
2.8.0-py2.py3-none-any.whl
Collecting numpy>=1.11 (from matplotlib)
Downloading https://files.pythonhosted.org/packages/c1/e2/4db8df8f6cddc98e7d7c537245ef
1.16.3-cp36-cp36m-manylinux1_x86_64.whl (17.3MB)
100% || 17.3MB 120kB/s
Collecting cycler>=0.10 (from matplotlib)
Using cached https://files.pythonhosted.org/packages/f7/d2/e07d3ebb2bd7af696440ce7e754
0.10.0-py2.py3-none-any.whl
Collecting setuptools (from kiwisolver>=1.0.1->matplotlib)
Using cached https://files.pythonhosted.org/packages/ec/51/f45cea425fd5cb0b0380f5b0f04
41.0.1-py2.py3-none-any.whl
Collecting six>=1.5 (from python-dateutil>=2.1->matplotlib)
Using cached https://files.pythonhosted.org/packages/73/fb/00a976f728d0d1fecfe898238ce
1.12.0-py2.py3-none-any.whl
Installing collected packages: setuptools, kiwisolver, pyparsing, six, python-dateutil
Successfully installed cycler-0.10.0 kiwisolver-1.1.0 matplotlib-3.1.0 numpy-1.16.3 py
2.4.0 python-dateutil-2.8.0 setuptools-41.0.1 six-1.12.0
```

import mathplot example

```
#!/usr/bin/python3
import math
import sys
import matplotlib.pyplot as plt
#import pyplot as name plt
#try to read in command line arguments that were passed to the program
if __name__ == '__main__':
    try:
        print(sys.argv[0])  #name of our program
        print(sys.argv[1])  #first parameter
    except:
        print("you did not entered any parameters")
        exit()


    plt.plot([1, 2, 3, 4])
    plot_label =  sys.argv[1]
    plt.ylabel(plot_label)
    plt.show()
```



11.4 Importing our own modules

If we create a function we can save it into a python script file and then import it into some other program.

function for module import

```
1 def occc_print(str):
2     print("This string was printed from occc_print")
3     print(str)
4     return
```

Save the above file as occc.py

import custom module

```
1 #!/usr/bin/python3
2 import occc
3 if __name__ == '__main__':
4     occc.occc_print("This is a test")
```

In the same directory create and run this script.

Output

```
occc@occc-VirtualBox:~/labs_import$ ./test_import.py
This string was printed from occc_print
This is a test
occc@occc-VirtualBox:~/labs_import$
```

```
occc@occc-VirtualBox:~/labs_import$ ls -l
total 12
-rw-r--r-- 1 occc occc  101 May 22 14:33 occc.py
drwxr-xr-x 2 occc occc 4096 May 22 14:34 __pycache__
-rwxr-xr-x 1 occc occc   66 May 22 14:34 test_import.py
occc@occc-VirtualBox:~/labs_import$
```

Using from ... import
To refer to items from a module within your program's namespace, you can use the from ... import statement. When you import modules this way, you can refer to the functions by name rather than through dot notation

import only a function

```
1 #!/usr/bin/python3
2 from occc import occc_print
3 if __name__ == '__main__':
4     occc_print("This is a test")
```

In the same directory create and run this script.

Output

```
occc@occc-VirtualBox:~/labs_import$ ./test_import.py
This string was printed from occc_print
This is a test
occc@occc-VirtualBox:~/labs_import$
```

11.5 Aliasing Modules

It is possible to modify the names of modules and their functions within Python by using the **"as"** keyword.

You may want to change a name because you have already used the same name for something else in your program, another module you have imported also uses that name, or you may want to abbreviate a longer name that you are using a lot.

import [module] as [another_name]

We did this above with the "import matplotlib.pyplot as plt"

Another example :

alias function names

```
1 #!/usr/bin/python3
2 from occc import occc_print as op
3 if __name__ == '__main__':
4     op("This is a test")
```

Output

```
occc@occc-VirtualBox:~/labs_import$ ./test_import.py
This string was printed from occc_print
This is a test
occc@occc-VirtualBox:~/labs_import$
```

11.6 ___name___

When the Python interpeter reads a source file, it first defines a few special variables. In this case, we care about the **___name___** variable.

When Your Module Is the Main Program

If you are running your module (the source file) as the main program, e.g.

python foo.py the interpreter will assign the hard-coded string "___main___" to the ___name___ variable, i.e.
It's as if the interpreter inserts this at the top
of your module when run as the main program.
___name___ = "___main___"
When your script is run by passing it as a command to the Python interpreter,

python3 myscript.py

all of the code that is at indentation level **0** gets executed. Functions and classes that are defined are, well, defined, but none of their code gets run. Unlike other languages, there's no main() function that gets run automatically - the main() function is implicitly all the code at the top level.

Example:

main file one

```
# file one.py
def func():
    print("func() in one.py")

print("top-level in one.py")

if __name__ == "__main__":
    print("one.py is being run directly")
else:
    print("one.py is being imported into another module")
```

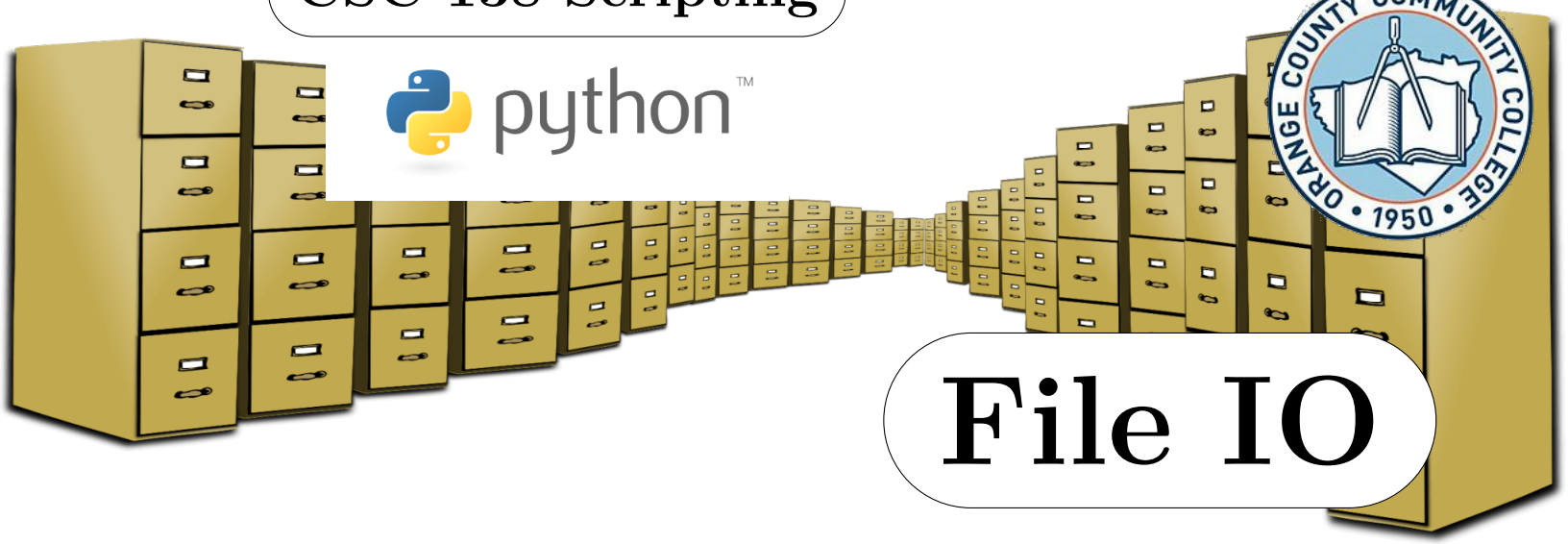
main file two

```
# file two.py
import one

print("top-level in two.py")
one.func()

if __name__ == "__main__":
    print("two.py is being run directly")
else:
    print("two.py is being imported into another module")
```

Output

```
occc@occc-VirtualBox:~/p$ python3 one.py
top-level in one.py
one.py is being run directly



occc@occc-VirtualBox:~/p$ python3 two.py
top-level in one.py
one.py is being imported into another module
top-level in two.py
func() in one.py
two.py is being run directly
occc@occc-VirtualBox:~/p$
```

CSC 138 Scripting



File IO

The open Function
Before you can read or write a file, you have to open it using Python's built-in open() function. This function creates a file object.
Syntax:
file object = open(file_name [, access_mode][, buffering])
Here are parameter details

file_name - The file_name argument is a string value that contains the name of the file that you want to access.
access_mode - The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is an optional parameter and the default file access mode is read (r).
buffering - If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).

modes of opening a file

Mode	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

Lets create a basic text file with the following content:

```
CIT 138
CSC 201
CSC 101
CSC 102
CIT 105
```

Save this file in your home directory and call it "courses.txt"
Example full path: "/home/occc/labs/courses.txt"

12.1 Opening a File

To open a file in Python, we first need some way to associate the file on disk with a variable in Python. This process is called opening a file. We begin by telling Python where the file is. The location of your file is often referred to as the file path. In order for Python to open your file, it requires the path.

file name

```
#!/usr/bin/python3

path = '/home/occc/labs/courses.txt'  #string that will hold
    ↪  the path to our file
```

We will then use Python's **open()** function to open our courses.txt file. The open() function requires as its first argument the file path. The function also allows for many other parameters. However, most important is the optional mode parameter. Mode is an optional string that specifies the mode in which the file is opened. The mode you choose will depend on what you wish to do with the file.

opening a file

```
#!/usr/bin/python3
if __name__ == '__main__':
    path = '/home/occc/labs/courses.txt'  #string that will
        ↪  hold the path to our file

    courses  = open(path,'r')  #open our file for reading
```

Specify path in python3
The path has to be in unix format.
example on windows the correct path in file explorer to a file c:\temp\example.txt will not open the file in python. the correct way to do this is to reverse the slashes and use a Unix directory format.
c:/temp/example.txt

12.2 Error checking for open file.

file open error try catch

```
#!/usr/bin/python3
if __name__ == '__main__':
    path = '/home/occc/labs/courses.txt'  #string that will
        ↪  hold the path to our file
    try:
        courses  = open(path,'r')  #open our file for reading
    except OSError:
        print("The file was not found or you don't have
            ↪  permissions to open the file")
        exit()
    courses.close()
```

12.3 Reading a File

Since our file was opened, we can now manipulate it (i.e. read from it) through the variable we assigned to it.

The first operation **<file>.read()** returns the entire contents of the file as a single string.

read whole file into a variable

```
1  #!/usr/bin/python3
2  if __name__ == '__main__':
3      path = '/home/occc/labs/courses.txt'  #string that will ↙
           ↳  hold the path to our file
4      try:
5          courses  = open(path,'r')  #open our file for reading
6      except OSError:
7          print("The file was not found or you dont have ↙
               ↳  permissions to open the file")
8          exit()
9      whole_file = courses.read()
10
11     print(whole_file)
12     courses.close()
```

Output

```
occc@occc-VirtualBox:~/labs$ ./files.py
CIT 138
CSC 201
CSC 101
CSC 102
CIT 105
occc@occc-VirtualBox:~/labs$
```

The second operation **<file>.readline()** returns the next line of the file, returning the text up to and including the next newline character. More simply put, this operation will read a file line-by-line.

readline file operation

```
#!/usr/bin/python3
if __name__ == '__main__':
    path = '/home/occc/labs/courses.txt'  #string that will hold the path to our file
    try:
        courses  = open(path,'r')  #open our file for reading
    except OSError:
        print("The file was not found or you dont have permissions to open the file")
        exit()
    whole_file = courses.read()

    print(whole_file)

    line1 = courses.readline()
    print("content without seek(0) {0}".format(line1))
    #this did not work!! the file was already read fully.
    #we have to set the position to 0

    courses.seek(0)
    line1 = courses.readline()
    print("content of first line after seek(0)")
    print(line1)
    line1 = courses.readline()
    print(line1)
    line1 = courses.readline()
    print(line1)
    line1 = courses.readline()
    print(line1)
    line1 = courses.readline()
    print(line1)
    line1 = courses.readline()
    print(line1)
    line1 = courses.readline()
    print(line1)
    line1 = courses.readline()
    print(line1)
    line1 = courses.readline()
    print(line1)
    courses.close()
```

Output

```
occc@occc-VirtualBox:~/labs$ ./files.py
CIT 138
CSC 201
CSC 101
CSC 102
CIT 105

content without seek(0)
content of first line after seek(0)
CIT 138

CSC 201

CSC 101

CSC 102

CIT 105

occc@occc-VirtualBox:~/labs$
```

<file>.seek()

Syntax

file. seek(offset[, whence])

offset

Required. The offset from the beginning of the file.

whence

Optional. The whence argument is optional and defaults to os.SEEK_SET or 0 (absolute file positioning); other values are os.SEEK_CUR or 1 (seek relative to the current position) and os.SEEK_END or 2 (seek relative to the file's end). There is no return value.

Given a file that contains "Orange County Community College"

seek file operation

```
1  if __name__ == '__main__':
2      seekexample = open('/home/occc/labs/seek.txt', 'r')  # open↙
           ↳   our file for reading
3      whole_file = seekexample.read()
4      print(whole_file)
5      seekexample.seek(3)
6      whole_file = seekexample.read()
7      print(whole_file)
8      seekexample.seek(7)
9      whole_file = seekexample.read()
10     print(whole_file)
```

Output

```
Orange County Community College
nge County Community College
County Community College
```

The next operation, **<file>.readlines()** returns a list of the lines in the file, where each item of the list represents a single line.

readlines

```
1  #!/usr/bin/python3
2  if __name__ == '__main__':
3     path = '/home/occc/labs/courses.txt'  #string that will ↙
          ↳  hold the path to our file
4     try:
5        courses  = open(path,'r')  #open our file for reading
6     except OSError:
7        print("The file was not found or you dont have ↙
             ↳  permissions to open the file")
8        exit()
9     list_of_lines = courses.readlines()
10
11    print(list_of_lines)
12    courses.close()
```

Output

```
occc@occc-VirtualBox:~/labs$ ./files.py
['CIT 138\n', 'CSC 201\n', 'CSC 101\n', 'CSC 102\n', 'CIT 105\n']
occc@occc-VirtualBox:~/labs$
```

12.4 Using loops to read a file

using loops for file reading

```
#!/usr/bin/python3

path = '/home/occc/labs/courses.txt'  #string that will hold the path to our file
try:
   courses  = open(path,'r')  #open our file for reading
except OSError:
   print("The file was not found or you dont have permissions to open the file")
   exit()
#using a for loop
print("using a for loop")
for line in courses.readlines():
   print(line)

#rewind the file
courses.seek(0)

#using a while loop example 1
print("using while true loop")

while(True):
   line = courses.readline()
   if not line:
      break     #there are no lines left stop the loop
   print(line)

#rewind the file
courses.seek(0)
#while loop example 2
print("Using a while loop that is primed")
line = courses.readline()
while(line):
   print(line)
   line = courses.readline()

#close the file
courses.close()
```

Output

```
using a for loop
CIT 138

CSC 201

CSC 101

CSC 102

CIT 105

using while true loop
CIT 138

CSC 201

CSC 101

CSC 102

CIT 105

Using a while loop that is primed
CIT 138

CSC 201

CSC 101

CSC 102

CIT 105

occc@occc-VirtualBox:~/labs$
```

12.5 Writing a file

We use the ”w” operation mode to open the file for writing.

python write to a file

```
#!/usr/bin/python3
if __name__ == '__main__':
    path = '/home/occc/labs/courses_writing.txt'  #string that will hold the path to our file
    try:
        courses  = open(path,'w')  #open our file for reading
    except OSError:
        print("The file was not found or you dont have permissions to open the file")
        exit()

    courses.write("ART 204\n")
    courses.write("CIT 116\n")
    courses.write("CSS 224\n")

    courses.close()
```

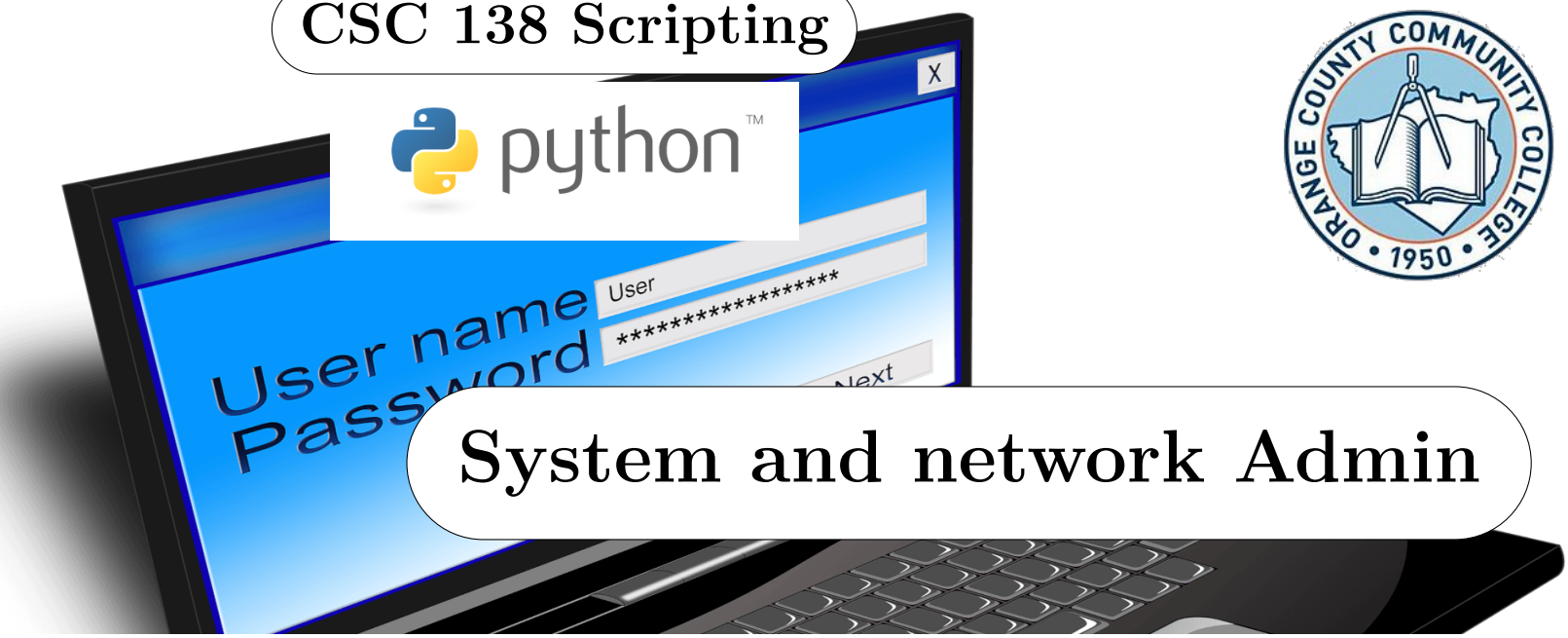
Output

```
occc@occc-VirtualBox:~/labs$ ./file2.py
occc@occc-VirtualBox:~/labs$ more /home/occc/labs/courses_writing.txt
ART 204
CIT 116
CSS 224
occc@occc-VirtualBox:~/labs$
```

12.6 Closing a File

Closing a file makes sure that the connection between the file on disk and the file variable is finished. Closing files also ensures that other programs are able to access them and keeps your data safe. So, always make sure to close your files. Close all our files using the **<file>.close()** function.

CSC 138 Scripting



System and network Admin

13.1 subprocess

The subprocess module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.

13.1.1 check_output

subprocess output

```
#!/usr/bin/python3

import subprocess

#get a directory listing (ls -l of /home/occc)
dir_list = subprocess.check_output('ls -l /home/occc', encoding='UTF-8', shell=True)
print(dir_list)
#the output is a string
#print(type(dir_list))
#if we want to do anything usefull with it we have to split it up and extract useful information
"""
total 208
drwxr-xr-x 2 occc occc   4096 Apr 30  2018 Desktop
drwxr-xr-x 2 occc occc   4096 Apr 30  2018 Documents
drwxr-xr-x 2 occc occc   4096 Feb  9 15:54 Downloads
-rw-r--r-- 1 occc occc   8980 Apr 30  2018 examples.desktop
"""

for lines in dir_list.split('\n'):  #split it along the new lines this will give us new strings
   #filter out the first line
   #print(lines)
   if(lines[:5] == 'total'):
      continue                      #ignore the first line
   elif len(lines) == 0:            #ignore empty lines
      continue
   else:                            # get the rest of information
      line_items = lines.split()  #split on spaces
      print(line_items[8])
```

Output

```
occc@occc-VirtualBox:~/lab3$ ./process.py
total 208
drwxr-xr-x 2 occc occc   4096 Apr 30  2018 Desktop
drwxr-xr-x 2 occc occc   4096 Apr 30  2018 Documents
drwxr-xr-x 2 occc occc   4096 Feb  9 15:54 Downloads
-rw-r--r-- 1 occc occc   8980 Apr 30  2018 examples.desktop
drwxr-xr-x 2 occc occc   4096 May  2 23:36 intellij
drwxr-xr-x 2 occc occc   4096 May 24 15:49 lab3
drwxr-xr-x 3 occc occc   4096 May 24 09:50 labs
drwxr-xr-x 2 occc occc   4096 Apr 30  2018 Music
drwxr-xr-x 2 occc occc   4096 Feb  9 15:31 Pictures
drwxr-xr-x 2 occc occc   4096 Apr 30  2018 Public
drwxr-xr-x 3 occc occc   4096 May  2 23:35 snap
drwxr-xr-x 2 occc occc   4096 Apr 30  2018 Videos

Desktop
Documents
Downloads
examples.desktop
intellij
lab3
labs
Music
Pictures
Public
snap
Videos
occc@occc-VirtualBox:~/lab3$
```

file type

```
#!/usr/bin/python3

import magic
import os
import subprocess
print(os.name)
if(os.name == 'posix'):
   print("unix like os")
   dir_list = subprocess.check_output('ls -l /home/occc',↙
      ↪  encoding='UTF-8',shell=True)
   for lines in dir_list.split('\n'):  #split it along the new↙
      ↪   lines this will give us new strings
      #filter out the first line
      #print(lines)
      if(lines[:5] == 'total'):
         continue                        #ignore the first line
      elif len(lines) == 0:              #ignore empty lines
         continue
      else:                              # get the rest of ↙
         ↪  information
         line_items = lines.split()  #split on spaces
         absolute_path = '/home/occc/' + line_items[8]
         if os.path.isfile(absolute_path):   `#detects if the ↙
            ↪  path is a regular file. exclude directories
           print("File: {0} is of type: {1}".format(line_items↙
              ↪  [8],magic.from_file(absolute_path)))
```

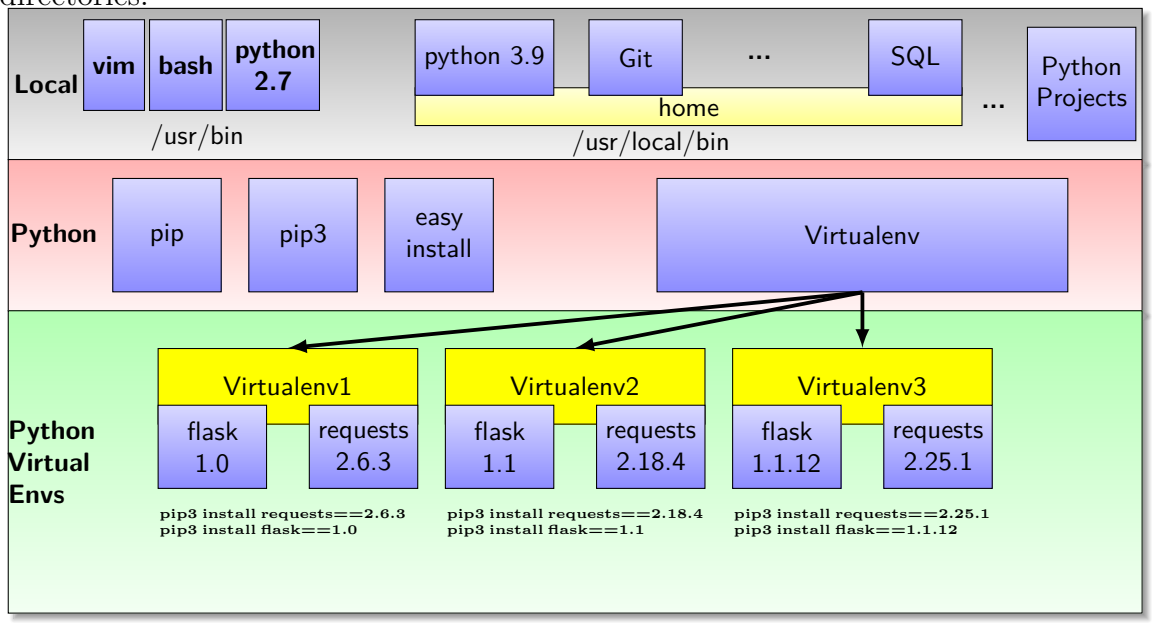
Output

```
occc@occc-VirtualBox:~/lab$ ./magic_file.py
posix
unix like os
File: cpp.pdf is of type: PDF document, version 1.5
File: drawrobot.m is of type: ASCII text
File: examples.desktop is of type: UTF-8 Unicode text
File: frame0001.png is of type: PNG image data, 800 x 600, 8-
bit/color RGB, non-interlaced
File: gauss.txt is of type: ASCII text
File: PlotFilteredReadings.m is of type: ASCII text
File: script.sh is of type: POSIX shell script, ASCII text executable
occc@occc-VirtualBox:~/lab$
```

13.2 virtenv Virtual Environments

The **venv** module provides support for creating lightweight “virtual environments” with their own site directories, optionally isolated from system site directories. Each virtual environment

has its own Python binary (which matches the version of the binary that was used to create this environment) and can have its own independent set of installed Python packages in its site directories.



Creation of virtual environments is done by executing the command venv:

python3 -m venv /path/to/new/virtual/environment

Running this command creates the target directory (creating any parent directories that don't exist already) and places a pyvenv.cfg file in it with a home key pointing to the Python installation from which the command was run. It also creates a bin (or Scripts on Windows) subdirectory containing a copy/symlink of the Python binary/binaries (as appropriate for the platform or arguments used at environment creation time). It also creates an (initially empty) lib/pythonX.Y/site-packages subdirectory (on Windows, this is Lib
site-packages). If an existing directory is specified, it will be re-used.

Deprecated since version 3.6: pyvenv was the recommended tool for creating virtual environments for Python 3.3 and 3.4, and is deprecated in Python 3.6.

Changed in version 3.5: The use of venv is now recommended for creating virtual environments.

```
occc@occc-VirtualBox:~$ mkdir virtenv
occc@occc-VirtualBox:~$ cd virtenv/
occc@occc-VirtualBox:~/virtenv$ python3 -m venv web
The virtual environment was not created successfully because ensurepip is not
available.  On Debian/Ubuntu systems, you need to install the python3-venv
package using the following command.

apt-get install python3-venv

You may need to use sudo with that command.  After installing the python3-venv
package, recreate your virtual environment.

Failing command: ['/home/occc/virtenv/web/bin/python3', '-Im', 'ensurepip', '--↙
    ↳  upgrade', '--default-pip']

occc@occc-VirtualBox:~/virtenv$ sudo apt-get install python3-venv
[sudo] password for occc:
Reading package lists... Done
Building dependency tree
```

```
Reading state information... Done
The following additional packages will be installed:
python3.6-venv
The following NEW packages will be installed:
python3-venv python3.6-venv
0 upgraded, 2 newly installed, 0 to remove and 0 not upgraded.
Need to get 7,392 B of archives.
After this operation, 44.0 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://us.archive.ubuntu.com/ubuntu bionic-updates/universe amd64 python3↙
    ↳  .6-venv amd64 3.6.7-1~18.04 [6,184 B]
Get:2 http://us.archive.ubuntu.com/ubuntu bionic-updates/universe amd64 python3-↙
    ↳  venv amd64 3.6.7-1~18.04 [1,208 B]
Fetched 7,392 B in 0s (47.2 kB/s)
Selecting previously unselected package python3.6-venv.
(Reading database ... 224223 files and directories currently installed.)
Preparing to unpack .../python3.6-venv_3.6.7-1~18.04_amd64.deb ...
Unpacking python3.6-venv (3.6.7-1~18.04) ...
Selecting previously unselected package python3-venv.
Preparing to unpack .../python3-venv_3.6.7-1~18.04_amd64.deb ...
Unpacking python3-venv (3.6.7-1~18.04) ...
Setting up python3.6-venv (3.6.7-1~18.04) ...
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
Setting up python3-venv (3.6.7-1~18.04) ...
occc@occc-VirtualBox:~/virtenv$ python3 -m venv web
occc@occc-VirtualBox:~/virtenv$ ls
web
occc@occc-VirtualBox:~/virtenv$ ls -l web
total 20
drwxr-xr-x 2 occc occc 4096 May 24 16:30 bin
drwxr-xr-x 2 occc occc 4096 May 24 16:29 include
drwxr-xr-x 3 occc occc 4096 May 24 16:29 lib
lrwxrwxrwx 1 occc occc    3 May 24 16:29 lib64 -> lib
-rw-r--r-- 1 occc occc   69 May 24 16:30 pyvenv.cfg
drwxr-xr-x 3 occc occc 4096 May 24 16:30 share
occc@occc-VirtualBox:~/virtenv$ ls -l web/bin
total 32
-rw-r--r-- 1 occc occc 2196 May 24 16:30 activate
-rw-r--r-- 1 occc occc 1252 May 24 16:30 activate.csh
-rw-r--r-- 1 occc occc 2416 May 24 16:30 activate.fish
-rwxr-xr-x 1 occc occc  252 May 24 16:30 easy_install
-rwxr-xr-x 1 occc occc  252 May 24 16:30 easy_install-3.6
-rwxr-xr-x 1 occc occc  224 May 24 16:30 pip
-rwxr-xr-x 1 occc occc  224 May 24 16:30 pip3
-rwxr-xr-x 1 occc occc  224 May 24 16:30 pip3.6
lrwxrwxrwx 1 occc occc    7 May 24 16:29 python -> python3
lrwxrwxrwx 1 occc occc   16 May 24 16:29 python3 -> /usr/bin/python3
occc@occc-VirtualBox:~/virtenv$
```

13.2.1 activate virtenv

```
occc@occc-VirtualBox:~/virtenv$ source web/bin/activate
(web) occc@occc-VirtualBox:~/virtenv$
```

13.2.2 Install local packages

```
(web) occc@occc-VirtualBox:~/virtenv$ pip3 install bs4
Collecting bs4
Using cached https://files.pythonhosted.org/packages/10/ed/7e8b97591f6f456174139ec089c769f89a94a1a4025fe967691de971f314/bs4-0.0.1.tar.gz
Requirement already satisfied: beautifulsoup4 in ./web/lib/python3.6/site-packages (from bs4)
Requirement already satisfied: soupsieve>=1.2 in ./web/lib/python3.6/site-packages (from beautifulsoup4->bs4)
Building wheels for collected packages: bs4
Running setup.py bdist_wheel for bs4 ... error
Complete output from command /home/occc/virtenv/web/bin/python3 -u -c "import setuptools, tokenize;__file__='/tmp/pip-build-4d8vyrvu/bs4/setup.py';f=getattr(tokenize, 'open', open)(__file__);code=f.read().replace('\r\n', '\n');f.close();exec(compile(code, __file__, 'exec'))" bdist_wheel -d /tmp/tmptb44dtkmpip-wheel- --python-tag cp36:
usage: -c [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
or: -c --help [cmd1 cmd2 ...]
or: -c --help-commands
or: -c cmd --help

error: invalid command 'bdist_wheel'

----------------------------------------
Failed building wheel for bs4
Running setup.py clean for bs4
Failed to build bs4
Installing collected packages: bs4
Running setup.py install for bs4 ... done
Successfully installed bs4-0.0.1
(web) occc@occc-VirtualBox:~/virtenv$
```

Find the new absolute path to python3

```
(web) occc@occc-VirtualBox:~/virtenv$ which python3
/home/occc/virtenv/web/bin/python3
(web) occc@occc-VirtualBox:~/virtenv$
```

Install all other necessary modules.

```
(web) occc@occc-VirtualBox:~/virtenv$ pip3 install requests
Collecting requests
....
```

python from virtenv

```
#!/home/occc/virtenv/web/bin/python3
import json
import subprocess
from bs4 import BeautifulSoup
import requests as req
#get the index page from google
resp = req.request(method='GET', url="http://www.google.com")
print(resp.status_code)
print(BeautifulSoup(resp.text, 'html.parser').prettify())
```

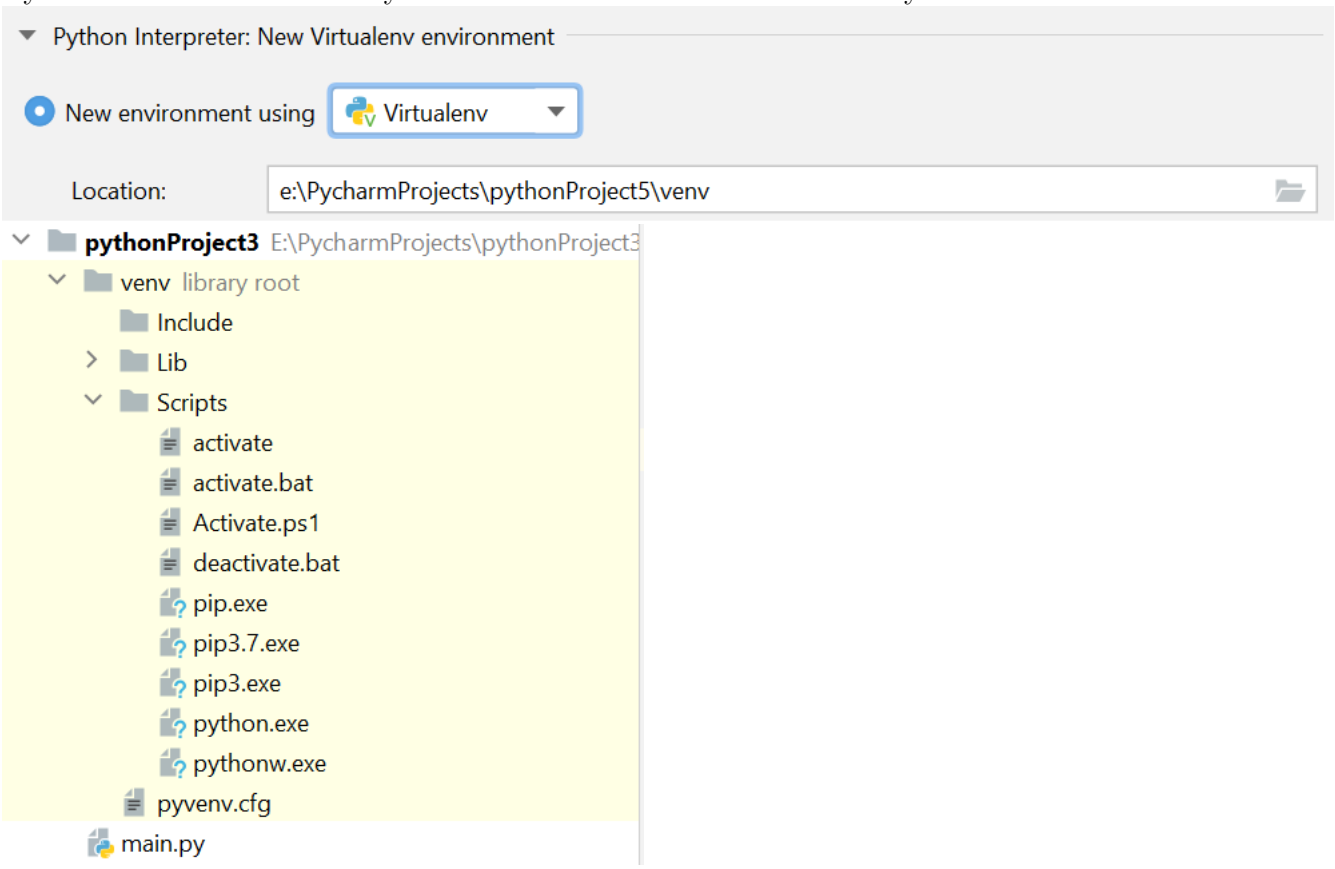
```
(web) occc@occc-VirtualBox:~/virtenv$ ./web.py
200
<!DOCTYPE doctype html>
<html itemscope="" itemtype="http://schema.org/WebPage" lang="en">
<head>
<meta content="Search the world's information, including webpages,↙
   ↪  images, videos and more. Google has many special features ↙
   ↪  to help you find exactly what you're looking for." name="↙
   ↪  description"/>
<meta content="noodp" name="robots"/>
<meta content="text/html; charset=utf-8" http-equiv="Content-Type"↙
   ↪  />
<meta content="/images/branding/googleg/1x/↙
   ↪  googleg_standard_color_128dp.png" itemprop="image"/>
<title>
Google
</title>
<script nonce="OsP7YJNujVoXQbpf5jvgdw==">
...
  </script>
</body>
</html>
```

13.2.3 exiting virt environment

```
(web) occc@occc-VirtualBox:~/virtenv$ deactivate
occc@occc-VirtualBox:~/virtenv$
occc@occc-VirtualBox:~/virtenv$ python3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more ↙
   ↪  information.
>>> from bs4 import BeautifulSoup
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
ModuleNotFoundError: No module named 'bs4'
>>>
```

Pycharms IDE automatically creates this virtual environment for you.



13.3 creating a simple webserver

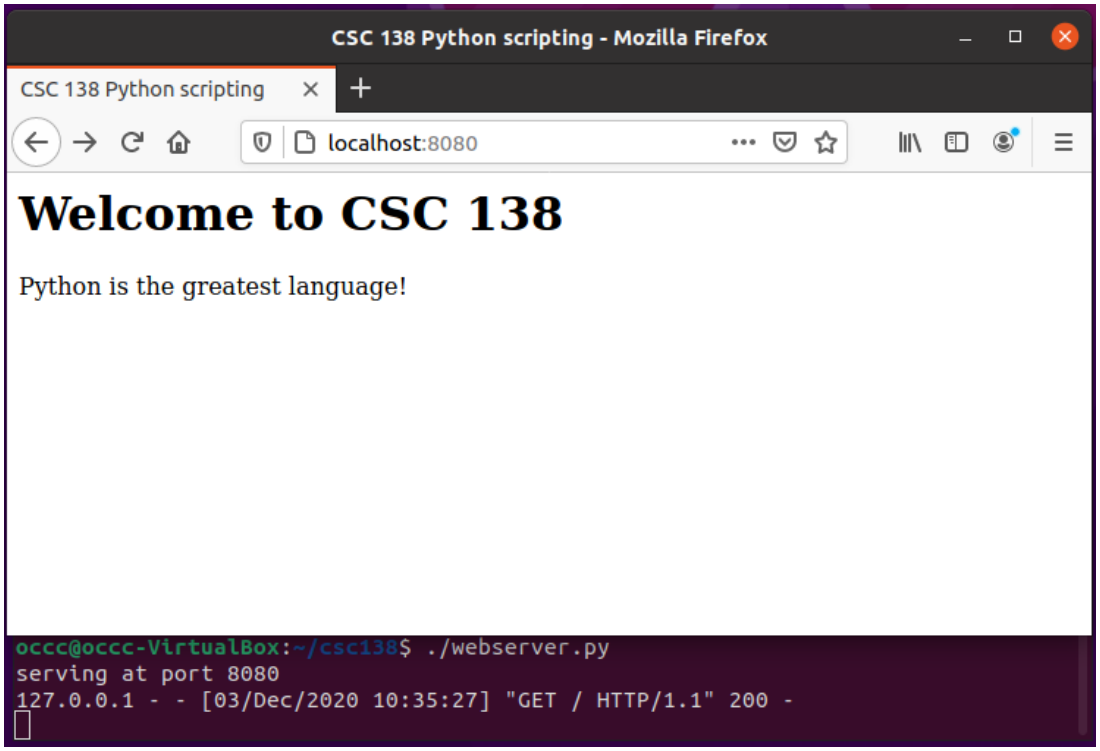
https://docs.python.org/3.6/library/http.server.html
Start by creating a simple html page save it as index.html

index.html

```
<html>
   <head>
     <title>CSC 138 Python scripting</title>
   </head>
   <body>
      <h1>Welcome to CSC 138</h1>
      <p>Python is the greatest language!</p>
   </body>
</html>
```

simple web server

```
1  #!/usr/bin/python3
2  import http.server
3  import socketserver
4
5  PORT = 8080     #port to which we wil bind. needs to be greater
   ↳    than 1024 otherwise we need to be root
6
7  Handler = http.server.SimpleHTTPRequestHandler
8
9  with socketserver.TCPServer(("", PORT), Handler) as httpd:
10    print("serving at port", PORT)
11    httpd.serve_forever()
```



13.4 Logging syslog

The **syslog** utility is used to create and store readable event notification messages so system administrators can manage their systems. we can log events from our scripts to this log facility to be analyzed for later use.

```
1  #!/usr/bin/python3
2  import syslog
3
4  syslog.syslog('Processing started')
5  #do some stuff
```

```
6 syslog.syslog('Processing ended')
```

Output

```
occc@occc-VirtualBox:~/lab3$ ./log.py
occc@occc-VirtualBox:~/lab3$ tail /var/log/syslog
May 25 18:22:21 occc-VirtualBox /log.py: Processing started
May 25 18:22:21 occc-VirtualBox /log.py: Processing ended
occc@occc-VirtualBox:~/lab3$
```

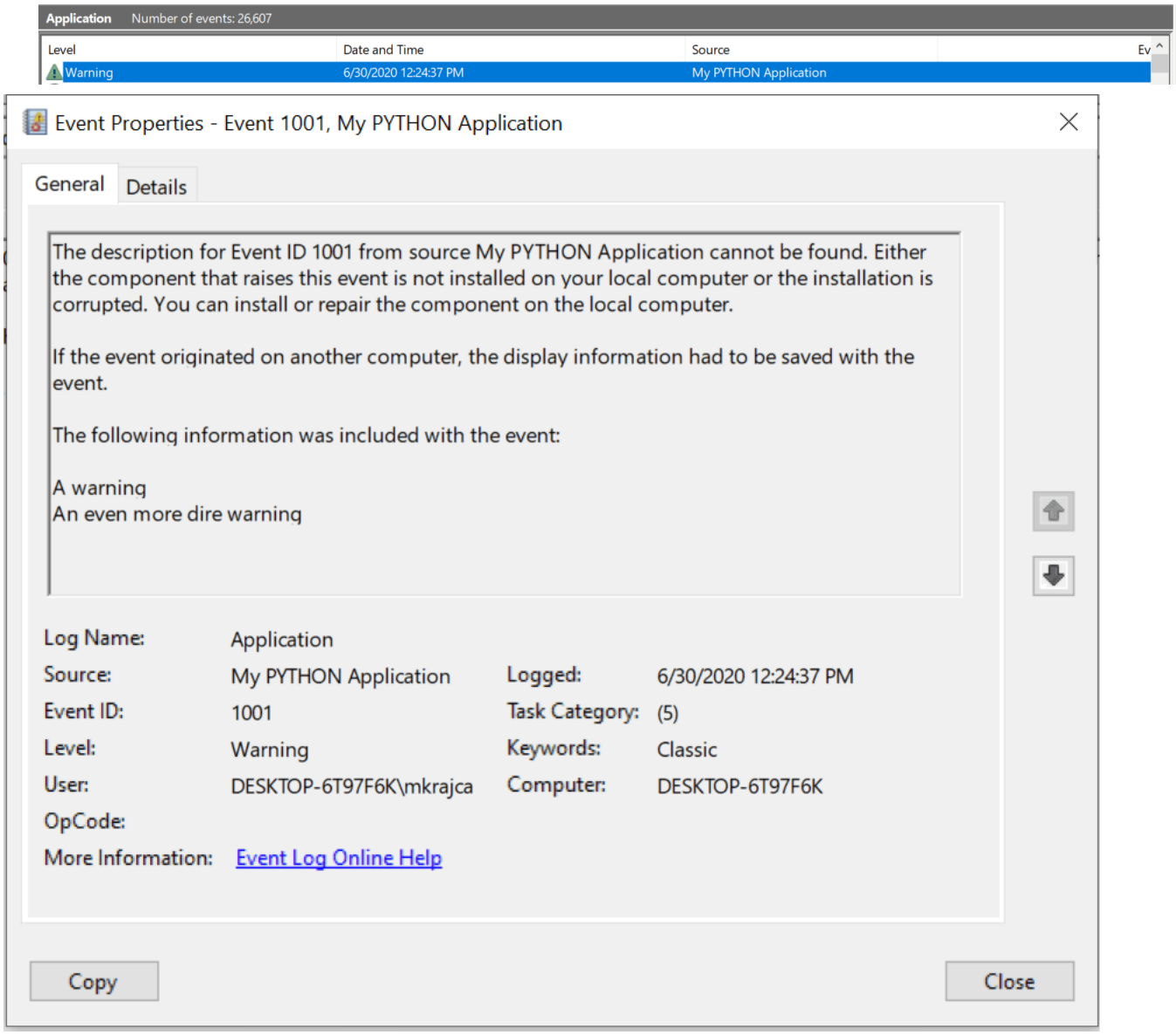
13.5 Windows Logging

install pywin32

```
(venv) E:\PycharmProjects\untitled2>pip3 install pywin32
Collecting pywin32
Downloading pywin32-228-cp37-cp37m-win32.whl (8.4 MB)
|                  | 8.4 MB 2.2 MB/s
Installing collected packages: pywin32
Successfully installed pywin32-228

(venv) E:\PycharmProjects\untitled2>
```

python windows logging

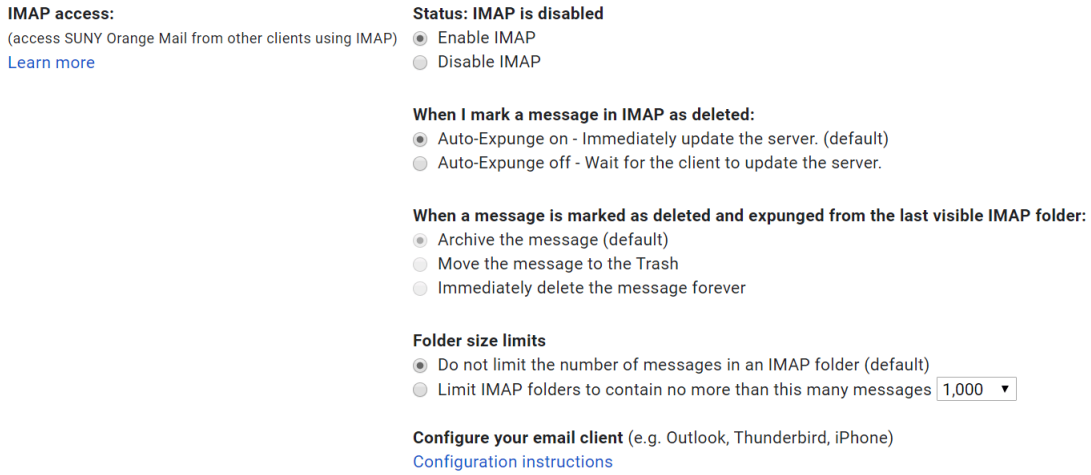
```
1
2  import win32api
3  import win32con
4  import win32evtlog
5  import win32security
6  import win32evtlogutil
7
8  ph = win32api.GetCurrentProcess()
9  th = win32security.OpenProcessToken(ph, win32con.TOKEN_READ)
10 my_sid = win32security.GetTokenInformation(th, win32security.↙
       ↳  TokenUser)[0]
11
12 applicationName = "My PYTHON Application"
13 eventID = 1001
14 category = 5  # Shell
15 myType = win32evtlog.EVENTLOG_WARNING_TYPE
16 descr = ["A warning", "An even more dire warning"]
17 data = "Application\0Data".encode("ascii")
18
19 win32evtlogutil.ReportEvent(applicationName, eventID, ↙
       ↳  eventCategory=category,eventType=myType, strings=descr, ↙
       ↳  data=data, sid=my_sid)
```



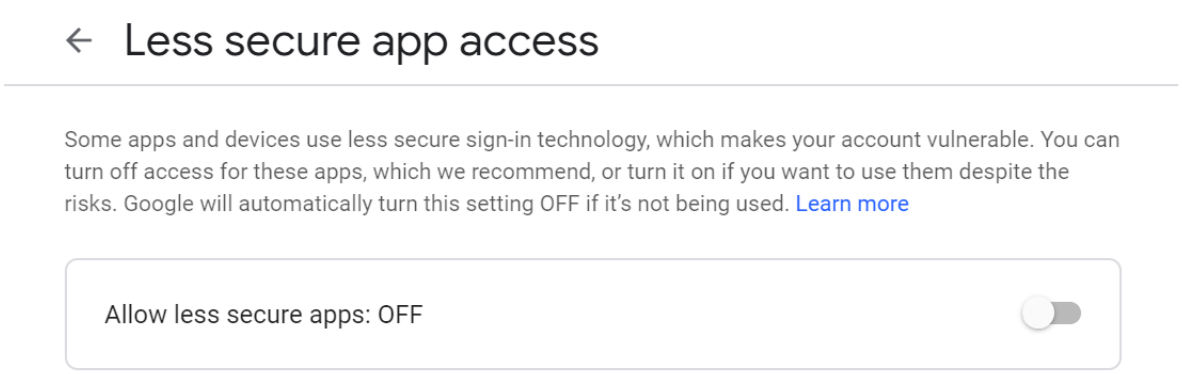
13.6 Email

Sending email from gmail account.

Config IMAP in Gmail.

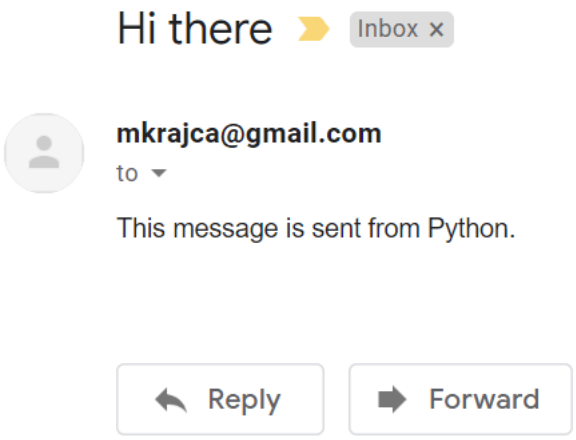


Config Less secure apps.



smtp sending emails

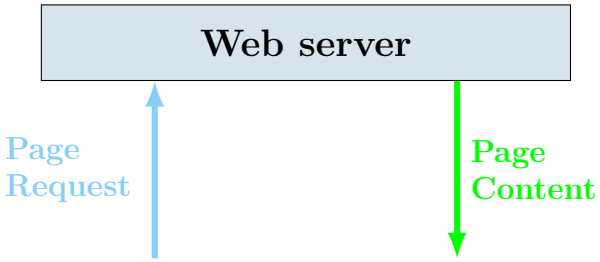
```
import smtplib, ssl

port = 587  # For starttls
smtp_server = "smtp.gmail.com"
sender_email = "mkrajca@gmail.com"
receiver_email = "miroslav.krajca@sunyorange.edu"
password = input("Type your password and press enter:")
message = """\
Subject: Hi there

This message is sent from Python."""

context = ssl.create_default_context()
with smtplib.SMTP(smtp_server, port) as server:
    server.ehlo()  # Can be omitted
    server.starttls(context=context)
    server.ehlo()  # Can be omitted
    server.login(sender_email, password)
    server.sendmail(sender_email, receiver_email, message)
```



13.7 API's

What is an API? An API, or Application Programming Interface, is a server that you can use to retrieve and send data to using code. APIs are most commonly used to retrieve data, and that will be the focus of this beginner tutorial.

When we want to receive data from an API, we need to make a request. Requests are used all over the web.



API requests work in exactly the same way – you make a request to an API server for data, and it responds to your request.

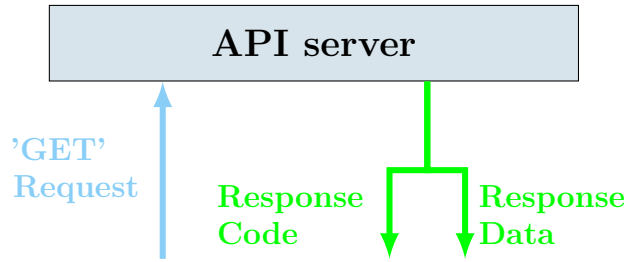
13.7.1 Making API Requests in Python

```
pip3 install requests
import requests
```

13.7.2 Making Our First API Request

There are many different types of requests. The most commonly used one, a GET request, is used to retrieve data. Because we'll just be working with retrieving data, our focus will be on making 'get' requests.

When we make a request, the response from the API comes with a response code which tells us whether our request was successful. Response codes are important because they immediately tell us if something went wrong.




```
response = requests.get("http://api.open-notify.org/this-api-doesnt-exist")
print(response.status_code)
```

404
The '404' status code might be familiar to you — it's the status code that a server returns if it can't find the file we requested. In this case, we asked for this-api-doesnt-exist which (surprise, surprise) didn't exist!

13.7.3 API Status Codes

Status codes are returned with every request that is made to a web server. Status codes indicate information about what happened with a request. Here are some codes that are relevant to GET requests:

200 : Everything went okay, and the result has been returned (if any).
301 : The server is redirecting you to a different endpoint. This can happen when a company switches domain names, or an endpoint name is changed.
400 : The server thinks you made a bad request. This can happen when you don't send along the right data, among other things.
401 : The server thinks you're not authenticated. Many APIs require login credentials, so this happens when you don't send the right credentials to access an API.
403 : The resource you're trying to access is forbidden: you don't have the right permissions to see it.
404 : The resource you tried to access wasn't found on the server.
503 : The server is not ready to handle the request.
You might notice that all of the status codes that begin with a '4' indicate some sort of error. The first number of status codes indicate their categorization. This is useful — you can know that if your status code starts with a '2' it was successful and if it starts with a '4' or '5' there was an error.

13.7.4 API Documentation

Often there will be multiple APIs available on a particular server. Each of these APIs are commonly called endpoints. The first endpoint we'll use is http://api.open-notify.org/astros.json, which returns data about astronauts currently in space.
response = requests.get("http://api.open-notify.org/astros.json") print(response.status_code)
200
We received a '200' code which tells us our request was successful. The documentation tells us that the API response we'll get is in JSON format. In the next section we'll learn about JSON, but first let's use the response.json() method to see the data we received back from the API:
print(response.json())
'people': ['name': 'Christina Koch', 'craft': 'ISS', 'name': 'Alexander Skvortsov', 'craft': 'ISS', 'name': 'Luca Parmitano', 'craft': 'ISS', 'name': 'Andrew Morgan', 'craft': 'ISS', 'name': 'Oleg Skripochka', 'craft': 'ISS', 'name': 'Jessica Meir', 'craft': 'ISS'], 'number': 6, 'message': 'success'

13.7.5 Using an API with Query Parameters

The http://api.open-notify.org/astros.json endpoint we used earlier does not take any parameters. We just send a GET request and the API sends back data about the number of people currently in space.

It's very common, however, to have an API endpoint that requires us to specify parameters. An example of this the http://api.open-notify.org/iss-pass.json endpoint. This endpoint tells us the next times that the international space station will pass over a given location on the earth.

If we look at the documentation, it specifies required lat (latitude) and long (longitude) parameters.

We can do this by adding an optional keyword argument, params, to our request. We can make a dictionary with these parameters, and then pass them into the requests.get function. Here's what our dictionary would look like, using coordinates for SUNY ORange:
#41.4386° N, -74.4261° W SUNY ORANGE location
parameters = {
"lat": 41.4386,
"lon": -74.4261
}

response = requests.get("http://api.open-notify.org/iss-pass.json", params=parameters)

Output

```
{
        "message": "success",
        "request": {
                "altitude": 100,
                "datetime": 1578774333,
                "latitude": 41.4386,
                "longitude": -74.4261,
```

```
                "passes": 5
        },
        "response": [
        {
                "duration": 536,
                "risetime": 1578794157
        },
        {
                "duration": 652,
                "risetime": 1578799879
        },
        {
                "duration": 602,
                "risetime": 1578805735
        },
        {
                "duration": 572,
                "risetime": 1578811613
        },
        {
                "duration": 628,
                "risetime": 1578817440
        }
        ]
}
```


Convert times from epoch to Gregorian

Output

```
2020-01-11 20:55:57
2020-01-11 22:31:19
2020-01-12 00:08:55
2020-01-12 01:46:53
2020-01-12 03:24:00
```

api calls

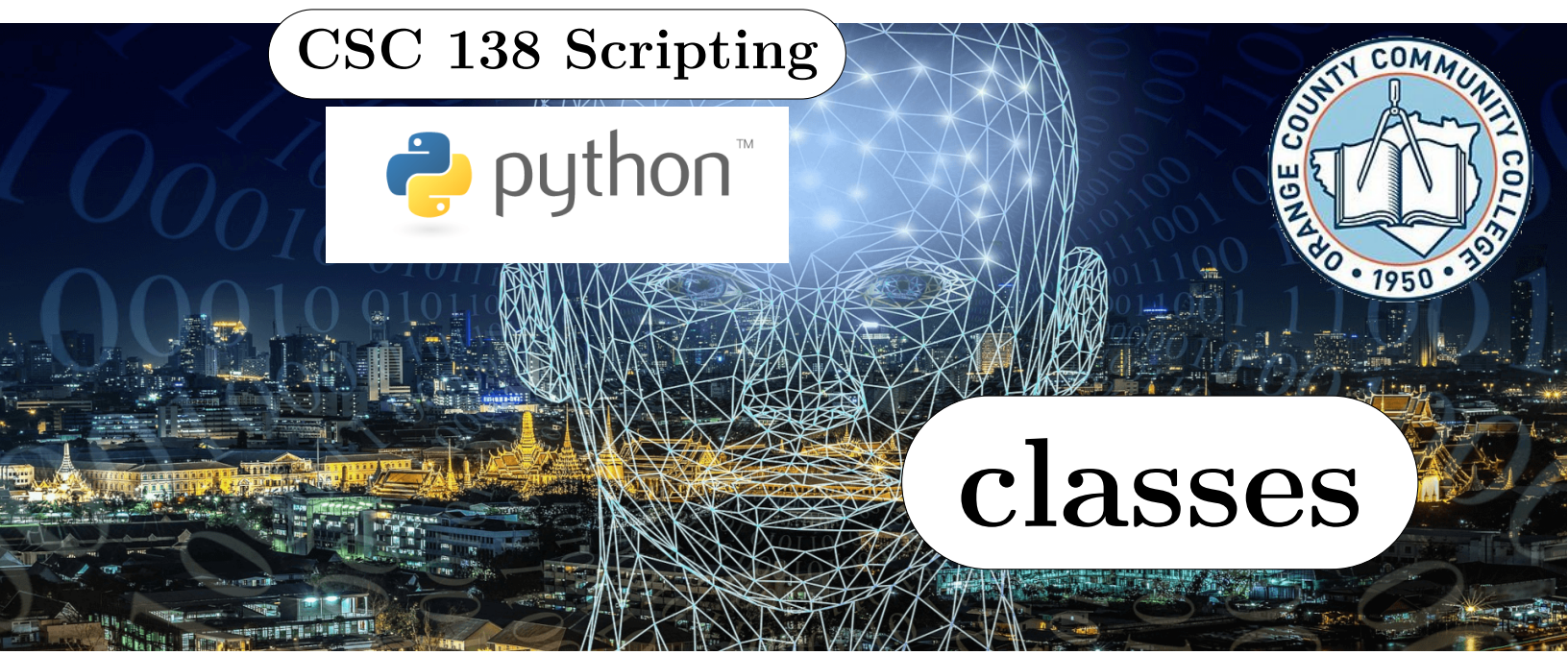
```
import requests
import json
from datetime import datetime

response = requests.get("http://api.open-notify.org/astros.json")
print(response.json())
text = json.dumps(response.json(), sort_keys=True, indent=4)
print(text)

#41.4386 N, -74.4261 W   SUNY ORANGE location

parameters = {
    "lat": 41.4386,
    "lon": -74.4261
}


response = requests.get("http://api.open-notify.org/iss-pass.json", params=parameters)
text = json.dumps(response.json(), sort_keys=True, indent=4)
print(text)
#extract times
pass_times = response.json()['response']
text = json.dumps(pass_times, sort_keys=True, indent=4)
print(text)

risetimes = []

for d in pass_times:
   time = d['risetime']
   risetimes.append(time)

print(risetimes)

times = []

for rt in risetimes:
   time = datetime.fromtimestamp(rt)
   times.append(time)
   print(time)
```

Output

```
occc@occc-VirtualBox:~$ python3 ./webbr.py
{'people': [{'name': 'Christina Koch', 'craft': 'ISS'},
            {'name': 'Alexander Skvortsov', 'craft': 'ISS'},
            {'name': 'Luca Parmitano', 'craft': 'ISS'},
            {'name': 'Andrew Morgan', 'craft': 'ISS'},
            {'name': 'Oleg Skripochka', 'craft': 'ISS'},
```

```
            {'name': 'Jessica Meir', 'craft': 'ISS'}], 'number': 6, 'message': 'succe
{
        "message": "success",
        "number": 6,
        "people": [
        {
                "craft": "ISS",
                "name": "Christina Koch"
        },
        {
                "craft": "ISS",
                "name": "Alexander Skvortsov"
        },
        {
                "craft": "ISS",
                "name": "Luca Parmitano"
        },
        {
                "craft": "ISS",
                "name": "Andrew Morgan"
        },
        {
                "craft": "ISS",
                "name": "Oleg Skripochka"
        },
        {
                "craft": "ISS",
                "name": "Jessica Meir"
        }
        ]
}
{
        "message": "success",
        "request": {
                "altitude": 100,
                "datetime": 1578774333,
                "latitude": 41.4386,
                "longitude": -74.4261,
                "passes": 5
        },
        "response": [
        {
                "duration": 536,
                "risetime": 1578794157
        },
        {
                "duration": 652,
                "risetime": 1578799879
        },
        {
                "duration": 602,
```

```
                "risetime": 1578805735
        },
        {
                "duration": 572,
                "risetime": 1578811613
        },
        {
                "duration": 628,
                "risetime": 1578817440
        }
        ]
}
[
{
        "duration": 536,
        "risetime": 1578794157
},
{
        "duration": 652,
        "risetime": 1578799879
},
{
        "duration": 602,
        "risetime": 1578805735
},
{
        "duration": 572,
        "risetime": 1578811613
},
{
        "duration": 628,
        "risetime": 1578817440
}
]
[1578794157, 1578799879, 1578805735, 1578811613, 1578817440]
2020-01-11 20:55:57
2020-01-11 22:31:19
2020-01-12 00:08:55
2020-01-12 01:46:53
2020-01-12 03:24:00
occc@occc-VirtualBox:~$
```



Python is an object oriented programming language. Unlike procedure oriented programming, where the main emphasis is on functions, object oriented programming stress on objects.

Object is simply a collection of data (variables) and methods (functions) that act on that data. A class is a blueprint for the object.

We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object.

As, many houses can be made from a description, we can create many objects from a class. An object is also called an instance of a class and the process of creating this object is called instantiation.

first class

```
1 #!/usr/bin/python3
2
3 class cit138:
4     name = "Python"     #member variable
5
6 x = cit138()    # x is an instance of object class type cit138
7 print(x.name)
```

Output

```
occc@occc-VirtualBox:~/lab3$ ./class.py
Python
occc@occc-VirtualBox:~/lab3$
```

Classes can have any data types that we discusses before as well as other classes(objects) as member variables. They can also have functions as members.

class member data variables

```
#!/usr/bin/python3

class cit138:
    name = "Python"
    def print_variables(self):
        print(self.name)

x = cit138()
print(x.name)
x.print_variables()
```

Output

```
occc@occc-VirtualBox:~/lab3$ ./class.py
Python
Python
occc@occc-VirtualBox:~/lab3$
```

The first parameter to a function inside a class will be the ”self” parameter.

class self

```
#!/usr/bin/python3

class cit138:
    name = "Python"
    def print_variables(self):
        print(self.name)

x = cit138()
print(x.name)
x.print_variables()
y = cit138()
y.name = "python Scripting"
print(y.name)
y.print_variables()
```

Output

```
occc@occc-VirtualBox:~/lab3$ ./class.py
Python
Python
python Scripting
python Scripting
occc@occc-VirtualBox:~/lab3$
```

The **self** variable is bound to the variable assigned to each instance of the object. When we have multiple instances we have to distinguish between them. The self variable keeps track of this. It does not have to be named self , you can call it whatever you like, but it has to be the

first parameter of any function in the class

14.1 docstring

Each class should have a docstring right after the class line declaration. this tells other programmers what your class does.

14.2 ___init___()

One of the first things you do with a class is to define the _init_() method. The ___init___() method sets the values for any parameters that need to be defined when an object is first created. When we call the class object, a new instance of the class is created, and the **___init___** method on this new object is immediately executed with all the parameters that we passed to the class object. The purpose of this method is thus to set up a new object using data that we have provided.

Note

___init___ is sometimes called the object's constructor, because it is used similarly to the way that constructors are used in other languages, but that is not technically correct – it's better to call it the initialiser. There is a different method called **___new___** which is more analogous to a constructor, but it is hardly ever used.

class init

```
1  #!/usr/bin/python3
2
3  class occc_class:
4     """ This class represents one class including the students
          ↪  and grades"""
5     name = ""
6     def __init__(self,class_name="generic class"):
7        self.name = class_name
8     def print_variables(self):
9        print(self.name)
10
11 generic = occc_class()
12 generic.print_variables()
13 cit138 = occc_class("Python scripting")
14 cit138.print_variables()
15
16 print(cit138.__doc__)
```

Output

```
occc@occc-VirtualBox:~/lab3$ ./class.py
generic class
Python scripting
This class represents one class including the students and grades
occc@occc-VirtualBox:~/lab3$
```

list of classes

```
1  #!/usr/bin/python3
2
3  class occc_class:
4     """ This class represents one class including the students
       ↪  and grades"""
5     name = ""
6     def __init__(self,class_name="generic class"):
7        self.name = class_name
8     def print_variables(self):
9        print(self.name)
10
11 all_classes = []
12 for x in range(0,4):
13    inp = input("please enter a class name   ")
14    oc = occc_class(inp)
15    all_classes.append(oc)
16
17
18
19 for cl in all_classes:
20    cl.print_variables()
```

Output

```
occc@occc-VirtualBox:~/lab3$ ./class.py
please enter a class name   cit138
please enter a class name   csc101
please enter a class name   csc102
please enter a class name   csc204
cit138
csc101
csc102
csc204
occc@occc-VirtualBox:~/lab3$
```

14.3 Adding properties to classes at run time

add items to class at runtime

```
1  #!/usr/bin/python3
2
3  class occc_class:
4     """ This class represents one class including the students
          and grades"""
5     name = ""
6     def __init__(self,class_name="generic class"):
7        self.name = class_name
8     def print_variables(self):
9        print(self.name)
10
11
12 cit138 = occc_class("Python")
13 csc101 = occc_class("comp sci 1")
14
15 cit138.extra = "scripting"
16
17 cit138.print_variables()
18 print(cit138.extra)
19 print(csc101.extra)  #this will cause error since we only
      added it to cit138
```

Output

```
occc@occc-VirtualBox:~/lab3$ ./class.py
Python
scripting
Traceback (most recent call last):
File "./class.py", line 19, in <module>
print(csc101.extra)
AttributeError: 'occc_class' object has no attribute 'extra'
occc@occc-VirtualBox:~/lab3$
```

14.4 Setters and getters

Setters is a class methods that will modify the data(variables) that are inside the class.
Getters are methods that retrieve data that is stored in the class object.
Although in most instances we can access the variables directly we should not do so. The setters methods should be used and we should have proper data validation of the input.

14.5 Inheritance

★ Python Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent / **superclass** class is the class being inherited from, also called **base** class.

Child / **subclass** class is the class that inherits from another class, also called **derived** class.

base class start

```
1  class Person:
2     def __init__(self, fname, lname):
3        self.firstname = fname         #add firstname variable to
            ↪   class
4        self.lastname = lname          #add lastname varibale to
            ↪  class
5
6     def printname(self):
7        print(self.firstname, self.lastname)
8
9  if __name__ == '__main__':
10    #Use the Person class to create an object, and then execute
         ↪   the printname method:
11    x = Person("John", "Doe")    #create an instance of object
         ↪  Person
12    x.printname()                #invoke Person object method
```

Output

```
John Doe
```

★ Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

derived class start

```
1  class Person:
2     def __init__(self, fname, lname):
3        self.firstname = fname         #add firstname variable to
            ↪   class
4        self.lastname = lname          #add lastname varibale to
            ↪  class
5
6     def printname(self):
7        print(self.firstname, self.lastname)
8
9  class Student(Person):
10    pass
11
12 if __name__ == '__main__':
13    #Use the Person class to create an object, and then execute
         ↪   the printname method:
14    x = Person("John", "Doe")    #create an instance of object
         ↪  Person
15    x.printname()                #invoke Person object method
16    y = Student("Jane","Doe")    #Create instance of student
17    y.printname()                #invoke Student object method
```

Output

```
John Doe
Jane Doe
```

Use the **pass** keyword when you do not want to add any other properties or methods to the class.
Student automatically inherited the printname() method.

Add the ___init___() Function
So far we have created a child class that inherits the properties and methods from its parent.

We want to add the ___init___() function to the child class (instead of the pass keyword).

derived class init

```
1  #!/usr/bin/python3
2  class Person:
3     def __init__(self, fname, lname):
4        self.firstname = fname         #add firstname variable to
            ↪   class
5        self.lastname = lname          #add lastname varibale to
            ↪  class
6
7     def printname(self):
8        print(self.firstname, self.lastname)
9
10 class Student(Person):
11    def __init__(self, fname, lname, student_gpa):
12       #assert isinstance(student_gpa, float)
13       self.gpa = student_gpa
14 if __name__ == '__main__':
15    #Use the Person class to create an object, and then execute
         ↪   the printname method:
16    x = Person("John", "Doe")    #create an instance of object
         ↪  Person
17    x.printname()                #invoke Person object method
18    y = Student("Jane","Doe",2.98)   #Create instance of
         ↪  student
19    y.printname()                #invoke Student object method
```

Output

```
occc@occc-VirtualBox:~/classes$ ./c.py
John Doe
Traceback (most recent call last):
  File "./c.py", line 19, in <module>
      y.printname()               #invoke Student object method
  File "./c.py", line 8, in printname
      print(self.firstname, self.lastname)
AttributeError: 'Student' object has no attribute 'firstname'
occc@occc-VirtualBox:~/classes$
```

Note that the derived class inherited the printname() method but the firstname and lastname variables were not created. This is because we created a new init() method and overrode the parents init() method.

The child's **___init___()** function overrides the inheritance of the parent's **___init___()** function.

★ Call Parent ___init___() method

derived class init of parent

```
1
2  #!/usr/bin/python3
3  class Person:
4     def __init__(self, fname, lname):
5        self.firstname = fname         #add firstname variable to
            ↪   class
6        self.lastname = lname          #add lastname varibale to
            ↪  class
7
8     def printname(self):
9        print(self.firstname, self.lastname)
10
11 class Student(Person):
12    def __init__(self, fname, lname, student_gpa):
13       #assert isinstance(student_gpa, float)
14       Person.__init__(self, fname, lname)
15       self.gpa = student_gpa
16
17 if __name__ == '__main__':
18    #Use the Person class to create an object, and then execute
         ↪   the printname method:
19    x = Person("John", "Doe")    #create an instance of object
         ↪  Person
20    x.printname()                #invoke Person object method
21    y = Student("Jane","Doe",2.98)   #Create instance of
         ↪  student
22    y.printname()                #invoke Student object method
```

Output

```
occc@occc-VirtualBox:~/classes$ ./c.py
ccc@occc-VirtualBox:~/classes$ ./c.py
John Doe
Jane Doe
occc@occc-VirtualBox:~/classes$
```

★ super() method

Use the super() Function

We can call parent methods by using the **super()** function.

derived class super()

```
#!/usr/bin/python3
class Person:
   def __init__(self, fname, lname):
      self.firstname = fname        #add firstname variable to class
      self.lastname = lname         #add lastname varibale to class

   def printname(self):
      print(self.firstname, self.lastname)

class Student(Person):
   def __init__(self, fname, lname, student_gpa):
      #assert isinstance(student_gpa, float)
      #Person.__init__(self, fname, lname)
      super().__init__(fname, lname)
      self.gpa = student_gpa

if __name__ == '__main__':
   #Use the Person class to create an object, and then execute the printname method:
   x = Person("John", "Doe")   #create an instance of object Person
   x.printname()               #invoke Person object method
   y = Student("Jane","Doe",2.98)   #Create instance of student
   y.printname()               #invoke Student object method
```

Output

```
occc@occc-VirtualBox:~/classes$ ./c.py
ccc@occc-VirtualBox:~/classes$ ./c.py
John Doe
Jane Doe
occc@occc-VirtualBox:~/classes$
```

Note: When using super() we do not pass the self as a parameter.

★ **Add Methods**

adding methods

```
1  #!/usr/bin/python3
2  class Person:
3     def __init__(self, fname, lname):
4        self.firstname = fname        #add firstname variable to
            ↪   class
5        self.lastname = lname         #add lastname varibale to
            ↪  class
6
7     def printname(self):
8        print(self.firstname, self.lastname)
9
10 class Student(Person):
11    def __init__(self, fname, lname, student_gpa):
12       #assert isinstance(student_gpa, float)
13       super().__init__(fname, lname)
14       self.gpa = student_gpa
15    def welcome(self):
16       print("Welcome", self.firstname, self.lastname, "your
            ↪  gpa is ", self.gpa)
17 if __name__ == '__main__':
18    #Use the Person class to create an object, and then execute
         ↪   the printname method:
19    x = Person("John", "Doe")   #create an instance of object
         ↪  Person
20    x.printname()               #invoke Person object method
21    y = Student("Jane","Doe",2.98)   #Create instance of
         ↪  student
22    y.printname()               #invoke Student object method
23    y.welcome()
```

Output

```
occc@occc-VirtualBox:~/classes$ ./c.py
John Doe
Jane Doe
Welcome Jane Doe your gpa is  2.98
occc@occc-VirtualBox:~/classes$
```

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

★ Function overriding

override parent methods

```
#!/usr/bin/python3
class Person:
   def __init__(self, fname, lname):
      self.firstname = fname        #add firstname variable to class
      self.lastname = lname         #add lastname varibale to class

   def printname(self):
      print(self.firstname, self.lastname)

class Student(Person):
   def __init__(self, fname, lname, student_gpa):
      #assert isinstance(student_gpa, float)
      super().__init__(fname, lname)
      self.gpa = student_gpa
   def welcome(self):
      print("Welcome", self.firstname, self.lastname, "your gpa is ", self.gpa)
   #override parent printname method
   def printname(self):
      print(self.firstname + " ------ " + self.lastname)
if __name__ == '__main__':
   #Use the Person class to create an object, and then execute the printname method:
   x = Person("John", "Doe")   #create an instance of object Person
   x.printname()               #invoke Person object method
   y = Student("Jane","Doe",2.98)   #Create instance of student
   y.printname()               #invoke Student object method
   y.welcome()
```

Output

```
occc@occc-VirtualBox:~/classes$ ./c.py
John Doe
Jane ------ Doe
Welcome Jane Doe your gpa is  2.98
occc@occc-VirtualBox:~/classes$
```

14.6 Advanced class topics

14.6.1 Magic functions

The following tables list important magic methods in Python 3.

Initialization and Construction

Initialization and Construction	Description
___new___(cls, other)	To get called in an object's instantiation.
___init___(self, other)	To get called by the ___new___ method.
___del___(self)	Destructor method.

Unary operators and functions

Unary operators and functions	Description
___pos___(self)	To get called for unary positive e.g. +someobject.
___neg___(self)	To get called for unary negative e.g. -someobject.
___abs___(self)	To get called by built-in abs() function.
___invert___(self)	To get called for inversion using the ~operator.
___round___(self,n)	To get called by built-in round() function.
___floor___(self)	To get called by built-in math.floor() function.
___ceil___(self)	To get called by built-in math.ceil() function.
___trunc___(self)	To get called by built-in math.trunc() function.

Augmented Assignment

Augmented Assignment	Description
___iadd___(self, other)	To get called on addition with assignment e.g. a +=b.
___isub___(self, other)	To get called on subtraction with assignment e.g. a -=b.
___imul___(self, other)	To get called on multiplication with assignment e.g. a *=b.
___ifloordiv___(self, other)	To get called on integer division with assignment e.g. a //=b.
___idiv___(self, other)	To get called on division with assignment e.g. a /=b.
___itruediv___(self, other)	To get called on true division with assignment
___imod___(self, other)	To get called on modulo with assignment e.g. a%=b.
___ipow___(self, other)	To get called on exponents with assignment e.g. a **=b.
___ilshift___(self, other)	To get called on left bitwise shift with assignment e.g. a«=b.
___irshift___(self, other)	To get called on right bitwise shift with assignment e.g. a »=b.
___iand___(self, other)	To get called on bitwise AND with assignment e.g. a&=b.
___ior___(self, other)	To get called on bitwise OR with assignment e.g. a|=b.
___ixor___(self, other)	To get called on bitwise XOR with assignment e.g. a ^=b.

Type Conversion Methods

Type Conversion	Description
___int___(self)	To get called by built-int int() method to convert a type to an int.
___float___(self)	To get called by built-int float() method to convert a type to float.
___complex___(self)	To get called by built-int complex() method to convert a type to complex.
___oct___(self)	To get called by built-int oct() method to convert a type to octal.
___hex___(self)	To get called by built-int hex() method to convert a type to hexadecimal.
___index___(self)	To get called on type conversion to an int when the object is used in a slice expression.
___trunc___(self)	To get called from math.trunc() method.

String Methods

String Methods	Description
___str___(self)	To get called by built-int str() method to return a string representation of a type.
___repr___(self)	To get called by built-int repr() method to return a machine readable representation of a type.
___unicode___(self)	To get called by built-int unicode() method to return an unicode string of a type.
___format___(self, formatstr)	To get called by built-int string.format() method to return a new style of str.
_hash___(self)	To get called by built-int hash() method to return an integer.
___nonzero___(self)	To get called by built-int bool() method to return True or False.
___dir___(self)	o get called by built-int dir() method to return a list of attributes of a class.
___sizeof___(self)	To get called by built-int sys.getsizeof() method to return the size of an object.

Attribute Methods

Attribute Methods	Description
___getattr___(self, name)	Is called when the accessing attribute of a class that does not exist.
___setattr___(self, name, value)	Is called when assigning a value to the attribute of a class.
___delattr___(self, name)	Is called when deleting an attribute of a class.

Operator Methods

Operator Methods	Description
___add___(self, other)	To get called on add operation using + operator
___sub___(self, other)	To get called on subtraction operation using - operator.
___mul___(self, other)	To get called on multiplication operation using * operator.
___floordiv___(self, other)	To get called on floor division operation using // operator.
___div___(self, other)	To get called on division operation using / operator.
___mod___(self, other)	To get called on modulo operation using % operator.
___pow___(self, other[, modulo])	To get called on calculating the power using ** operator.
___lt___(self, other)	To get called on comparison using <operator.
___le___(self, other)	To get called on comparison using <= operator.
___eq___(self, other)	To get called on comparison using == operator.
___ne___(self, other)	To get called on comparison using != operator.
___ge___(self, other)	To get called on comparison using >= operator.
___gt___(self, other)	To get called on comparison using >operator.

There other methods which are not listed here.

★ Magic methods examples

magic methods examples

```
1  #!/usr/bin/python3
2  class Person:
3     def __init__(self, fname, lname):
4        self.firstname = fname         #add firstname variable to
            ↪   class
5        self.lastname = lname          #add lastname varibale to
            ↪  class
6
7     def printname(self):
8        print(self.firstname, self.lastname)
9
10 class Student(Person):
11    def __init__(self, fname, lname, student_gpa):
12       #assert isinstance(student_gpa, float)
13       super().__init__(fname, lname)
14       self.gpa = student_gpa
15
16    def welcome(self):
17       print("Welcome", self.firstname, self.lastname, "your
            ↪  gpa is ", self.gpa)
18       #override parent printname method
19
20    def printname(self):
21       print(self.firstname + " ------ " + self.lastname)
22
23    def __eq__(self, other):
24       if(self.firstname == other.firstname) and (self.lastname
```

```
            ↪    == other.lastname):
25             return True
26          else:
27             return False
28
29  if __name__ == '__main__':
30     #Use the Person class to create an object, and then execute
          ↪    the printname method:
31     x = Person("John", "Doe")    #create an instance of object
          ↪  Person
32     x.printname()                #invoke Person object method
33     y = Student("Jane","Doe",2.98)   #Create instance of
          ↪  student
34     y.printname()                #invoke Student object method
35     y.welcome()
36     z = Student("Jane","Doe",3.00)
37     q = Student("Mary","Jane",3.00)
38     if(z == y):                  #z = "Jane","Doe", y = "Jane","Doe"
39        print("z equals y")
40     else:
41        print("z does not equal y")
42     if(z == q):                  #z = "Jane","Doe", q = "Mary","Jane"
43        print("z equals q")
44     else:
45        print("z does not equal q")
```

Output

```
occc@occc-VirtualBox:~/classes$ ./c.py
John Doe
Jane ------ Doe
Welcome Jane Doe your gpa is  2.98
z equals y
z does not equal q
occc@occc-VirtualBox:~/classes$
```

Advanced topics

regular expressions

In 1951, mathematician Stephen Cole Kleene described the concept of a regular language, a language that is recognizable by a finite automaton and formally expressible using regular expressions. In the mid-1960s, computer science pioneer Ken Thompson, one of the original designers of Unix, implemented pattern matching in the QED text editor using Kleene's notation.

↦ **Python implements regular expressions with the "re" module**

```
re.search(pattern, string, flags=0)
```

re search params

param.No.	Parameter & Description
1	pattern This is the regular expression to be matched.
2	string This is the string, which would be searched to match the pattern anywhere in the string.
3	flags You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below

re search flags

Modifier & Description
re.I **Performs case-insensitive matching.**
re.L **Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior(\b and \B).**
re.M **Makes $ match the end of a line (not just the end of the string) and makes ^match the start of any line (not just the start of the string).**
re.S **Makes a period (dot) match any character, including a newline.**
re.U **Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.**
re.X **Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker..**

↦ Regular Expression Patterns

Except for control characters, `+ ? . * ^ $ ( ) [ ] { } | \`, all characters match themselves. You can escape a control character by preceding it with a backslash.

Following table lists the regular expression syntax that is available in Python −

re search patterns 1/2

Pattern & Description
^ Matches beginning of line.
$ Matches end of line.
. Matches any single character except newline. Using m option allows it to match newline as well.
[...] Matches any single character in brackets.
[^...] Matches any single character not in brackets
re* Matches 0 or more occurrences of preceding expression.
re+ Matches 1 or more occurrence of preceding expression.
re? Matches 0 or 1 occurrence of preceding expression.
re{ n} Matches exactly n number of occurrences of preceding expression.
re{ n,} Matches n or more occurrences of preceding expression.
re{ n, m} Matches at least n and at most m occurrences of preceding expression.
a| b Matches either a or b.
(re) Groups regular expressions and remembers matched text.
(?imx) Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.
(?-imx) Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected.
(?: re) Groups regular expressions without remembering matched text.
(?imx: re) Temporarily toggles on i, m, or x options within parentheses.
(?-imx: re) Temporarily toggles off i, m, or x options within parentheses.
(?#...) Comment.
(?= re) Specifies position using a pattern. Doesn't have a range.
(?! re) Specifies position using pattern negation. Doesn't have a range.
(?>re) Matches independent pattern without backtracking.

re search patterns 2/2

Pattern & Description
\w Matches word characters.
\W Matches nonword characters.
\s Matches whitespace. Equivalent to [\t\n\r\f].
\S Matches nonwhitespace.
\d Matches digits. Equivalent to [0-9].
\D Matches nondigits.
\A Matches beginning of string.
\Z Matches end of string. If a newline exists, it matches just before newline.
\z Matches end of string.
\G Matches point where last match finished.
\b Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
\B Matches nonword boundaries.
\n, \t, etc. Matches newlines, carriage returns, tabs, etc.
\1...\9 Matches nth grouped subexpression.
\10 Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

regex

```
#!/usr/bin/python3
import re
if __name__ == "__main__":
    sentence = "This is an example sentence, it is for demonstration only"
    print(sentence)
    print(re.search('is', sentence))   #exact patern match. same as string find
    result = re.search('is', sentence) #save search result to a var
    print(result.span())     #result span() shows the string index numbers of first find
```

```
    print(result.start())    #start index in string of first find
    print(result.end())      #end index of first find

    #if we want to find all occurences of a patern match we have to
       ↪   iterate over them

    iter = re.finditer('is', sentence)  #create an iterator that
       ↪  wil go over each match
    #indices = [m.start(0) for m in iter]
    #print(iter)

    for index in iter:     #get each instance of match if there is
       ↪  no match it will retun 'None'
       print(index)
       print(index.span())

    iter = re.finditer('xyz', sentence)

    for index in iter:     #get each instance of match if there is
       ↪  no match it will retun 'None'
       print('We should not get here')
       print(index)
       print(index.span())
```

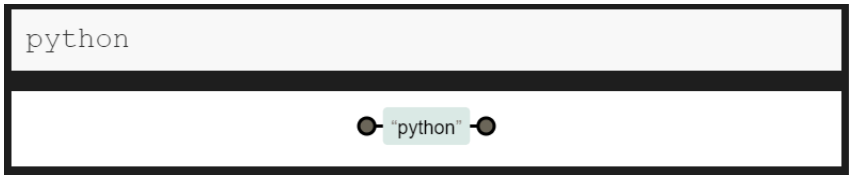
Output

```
This is an example sentence, it is for demonstration only
<re.Match object; span=(2, 4), match='is'>
(2, 4)
2
4
<re.Match object; span=(2, 4), match='is'>
(2, 4)
<re.Match object; span=(5, 7), match='is'>
(5, 7)
<re.Match object; span=(32, 34), match='is'>
(32, 34)
```

python
Match python word exactly

result = re.search(r'python',"python")
print(result.span())

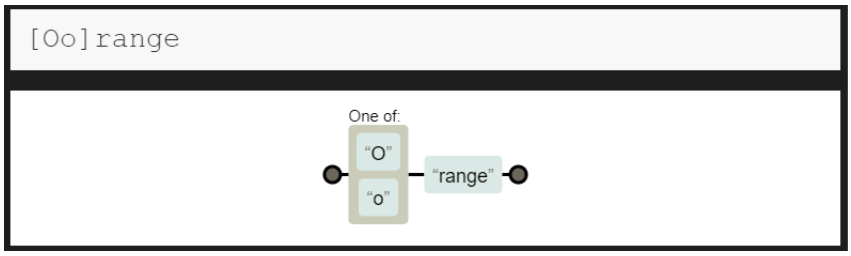
(0, 6)



```
result = re.search(r'[Oo]range', "Orange")
print(result.span())
```

```
(0,6)
```

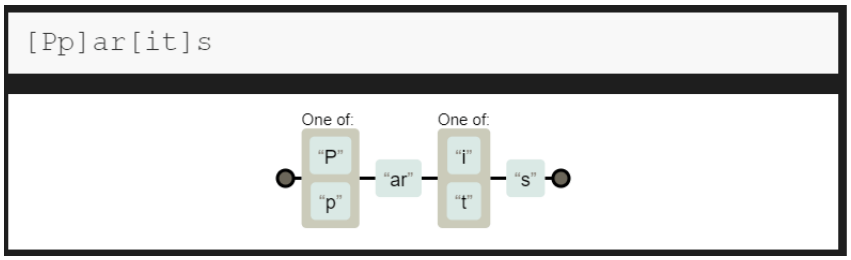
[Oo]range
Match Orange, orange




```
result = re.search(r'[Pp]ar[it]s', "Paris")
print(result.span())
```

```
(0,5)
```

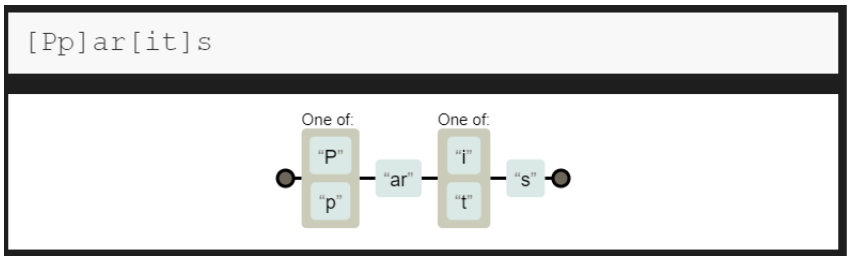
[Pp]ar[it]s
Matches parts, Paris




```
iter = re.finditer(r'[aeiou]', "Paris")
for index in iter:
   print(index)
   print(index.span())
```

```
<re.Match object; span=(1, 2), match='a'>
(1, 2)
<re.Match object; span=(3, 4), match='i'>
(3, 4)
```

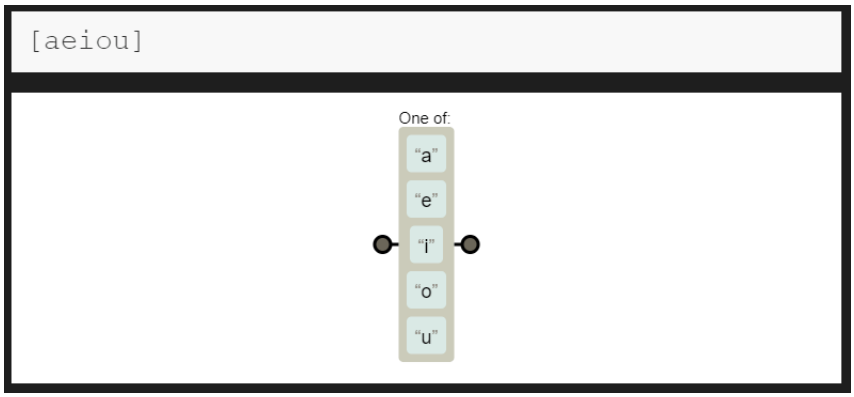
[aeiou]
Match any one lowercase vowel



```
iter = re.finditer(r'[aeiou]', "Paris")
for index in iter:
   print(index)
   print(index.span())
```

```
<re.Match object; span=(1, 2), match='a'>
(1, 2)
<re.Match object; span=(3, 4), match='i'>
(3, 4)
```

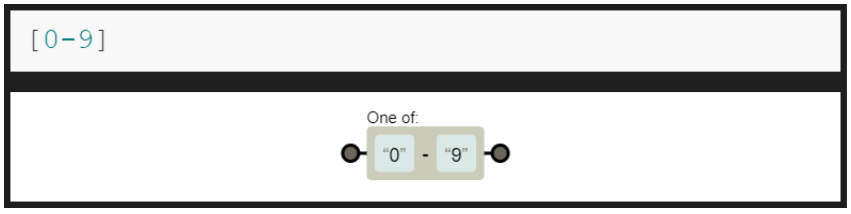
[aeiou]
Match any one lowercase vowel




```
iter = re.finditer(r'[0-9]', "occc123")
for index in iter:
   print(index)
   print(index.span())
```

```
<re.Match object; span=(4, 5), match='1'>
(4, 5)
<re.Match object; span=(5, 6), match='2'>
(5, 6)
<re.Match object; span=(6, 7), match='3'>
(6, 7)
```

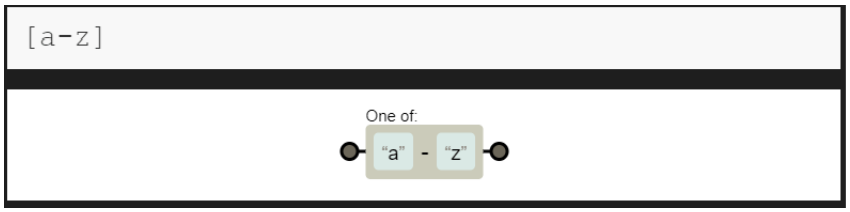
[0-9]
Match any digit; same as [0123456789]



[a-z]
Match any lowercase ASCII letter

```
iter = re.finditer(r'[a-z]', "aBc123")
for index in iter:
   print(index)
   print(index.span())
```

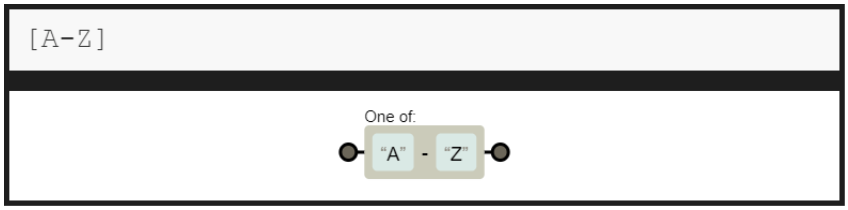
```
<re.Match object; span=(0, 1), match='a'>
(0, 1)
<re.Match object; span=(2, 3), match='c'>
(2, 3)
```



[A-Z]
Match any uppercase ASCII letter

```
iter = re.finditer(r'[A-Z]', "aBc123")
for index in iter:
   print(index)
   print(index.span())
```

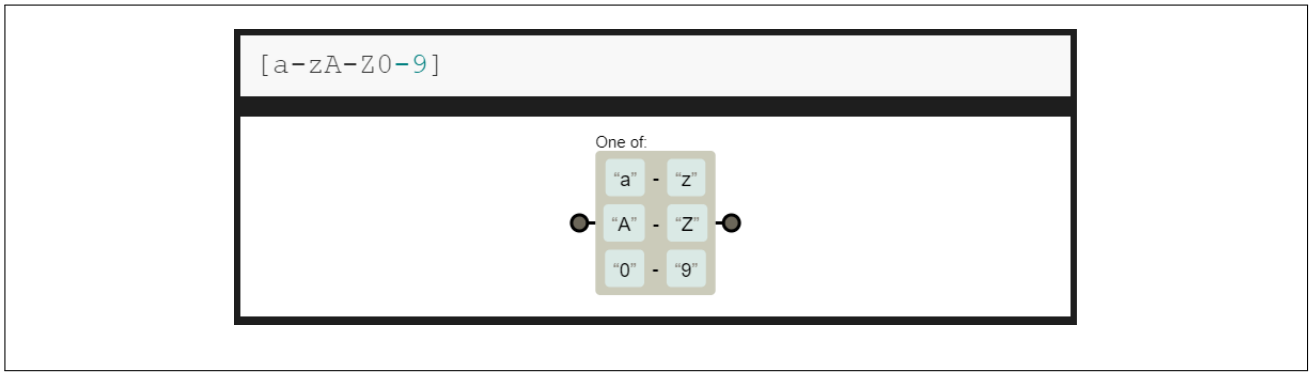
```
<re.Match object; span=(1, 2), match='B'>
(1, 2)
```



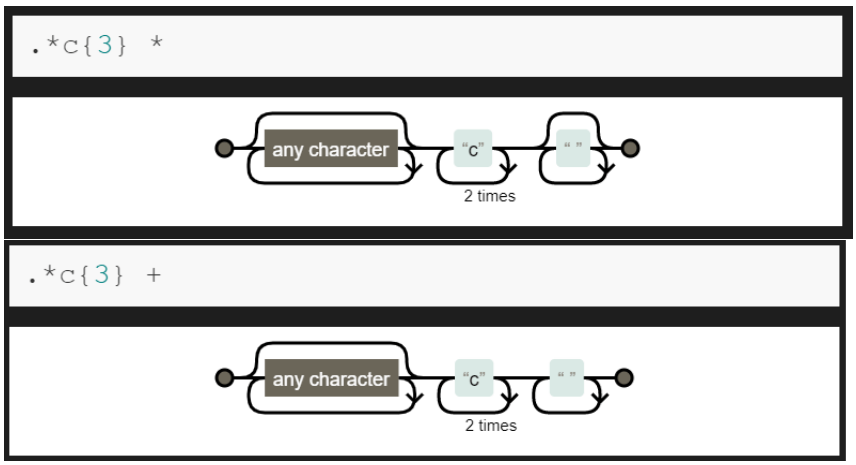
[a-zA-Z0-9]
Match any of the 3 above

```
iter = re.finditer(r'[a-zA-Z0-9]', "!#.?Ab4")
for index in iter:
   print(index)
   print(index.span())
```

```
<re.Match object; span=(4, 5), match='A'>
(4, 5)
<re.Match object; span=(5, 6), match='b'>
(5, 6)
<re.Match object; span=(6, 7), match='4'>
(6, 7)
```



Find any word that ends with "ccc"



regex

```python
import re
if __name__ == "__main__":

    sentence_another = "Orange(occc) County"
    print(sentence)


    print(re.search(r'.*c{3} *', sentence))   #finds any string that ends with ccc
    #followed by zero or more space
    result = re.search(r'.*c{3} *', sentence) #save search result to a var
    print(result.span())     #result span() shows the string index numbers of first find
    print(result.start())    #start index in string of first find
    print(result.end())      #end index of first find

    iter = re.finditer(r'.*c{3} *', sentence)  #create an iterator that wil go over each match

    for index in iter:     #get each instance of match if there is no match it will retun 'None'
        print(index)
        print(index.span())

```

```
    print("--------------------")
    print(sentence_another)
    print("  r'.*c{3} *'  ")
    print(re.search(r'.*c{3} *', sentence_another))
    result = re.search(r'.*c{3} *', sentence_another)  # save search result to a var
    if result != None:
        print(result.span())  # result span() shows the string index numbers of first find
        print(result.start())  # start index in string of first find
        print(result.end())  # end index of first find
    print(" r'.*c{3} +'        ")
    print(re.search(r'.*c{3} +', sentence_another))
    result = re.search(r'.*c{3} +', sentence_another)  # save search result to a var
    if result != None:
        print(result.span())  # result span() shows the string index numbers of first find
        print(result.start())  # start index in string of first find
        print(result.end())  # end index of first find
```

Output

```
occc = Orange County Community College
<re.Match object; span=(0, 5), match='occc '>
(0, 5)
0
5
<re.Match object; span=(0, 5), match='occc '>
(0, 5)
--------------------
Orange(occc) County
r'.*c{3} *'
<re.Match object; span=(0, 11), match='Orange(occc'>
(0, 11)
0
11
r'.*c{3} +'
None
```

Lets examine the difference between **.*c3 *** and **.*c3 +**

.*c3 *

.* matches any character (except for line terminators)
* Quantifier — Matches between zero and unlimited times, as many times as possible, giving back as needed (greedy)
c3 matches the character c literally (case sensitive)
3 Quantifier — Matches exactly 3 times
* matches the character ' ' literally (case sensitive)
* Quantifier — Matches between zero and unlimited times, as many times as possible, giving back as needed (greedy)

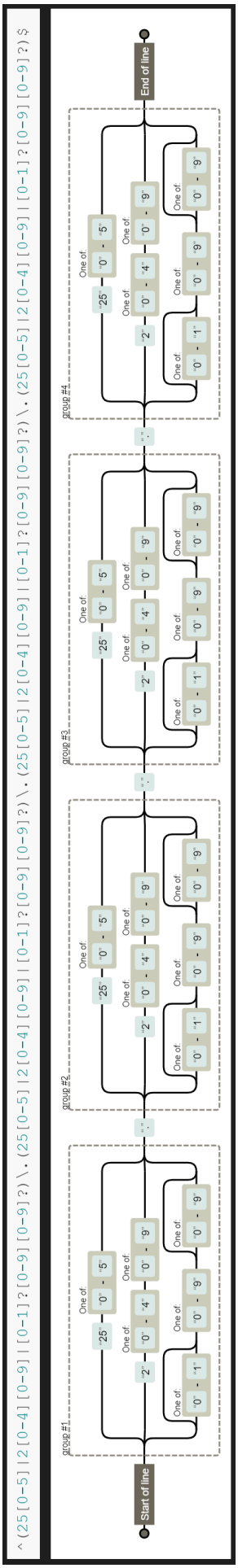
.*c3 +

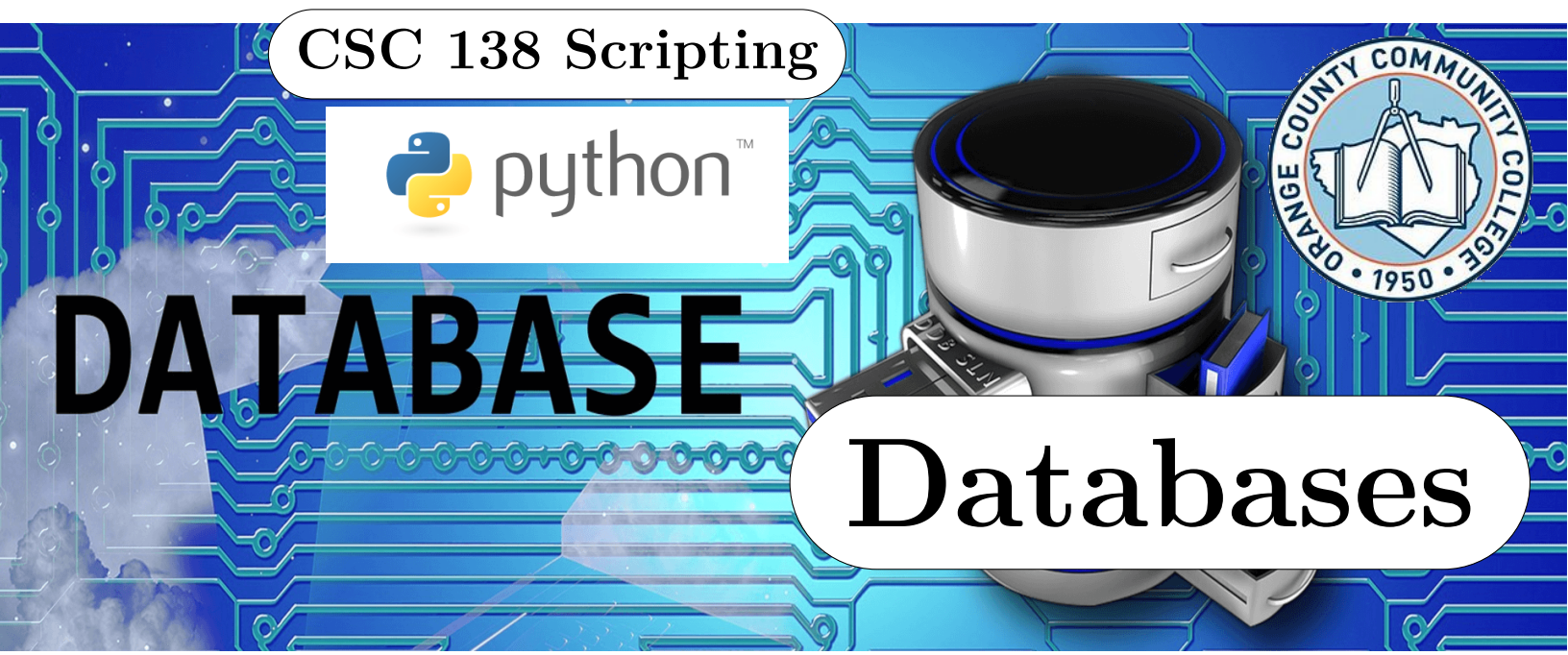
.* matches any character (except for line terminators)
* Quantifier — Matches between zero and unlimited times, as many times as possible, giving back as needed (greedy)
c3 matches the character c literally (case sensitive)
3 Quantifier — Matches exactly 3 times
+ matches the character ' ' literally (case sensitive)
+ Quantifier — Matches between one and unlimited times, as many times as possible, giving back as needed (greedy)

The regex with the "*" matched the word despite we wanted zero or more space. Since it could be zero it was able to ignore the space request as zero and still match the word.
The + one said one or more spaces so the space had to be present and thus it failed after it found ccc followed by) and not a space

More complicated regex that will capture if input is a valid IPV4 address.

```
r'^(25[0-5]|2[0-4][0-9]|[0-1]?[0-9][0-9]?)\.
(25[0-5]|2[0-4][0-9]|[0-1]?[0-9][0-9]?)\.
(25[0-5]|2[0-4][0-9]|[0-1]?[0-9][0-9]?)\.
(25[0-5]|2[0-4][0-9]|[0-1]?[0-9][0-9]?)$'
```


^(25[0-5]|2[0-4][0-9]|[0-1]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[0-1]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[0-1]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[0-1]?[0-9][0-9]?)$
Start of line
End of line
group #1
group #2
group #3
group #4
"."
One of:
"25"
"0" - "5"
"2"
"0" - "4"
"0" - "1"
"0" - "9"



Some parts of this chapter are taken from: https://books.trinket.io/pfe/ Text © Charles R. Severance. Interactive HTML © Trinket. Both provided under a CC-NC-BY-SA license.

What is a database?

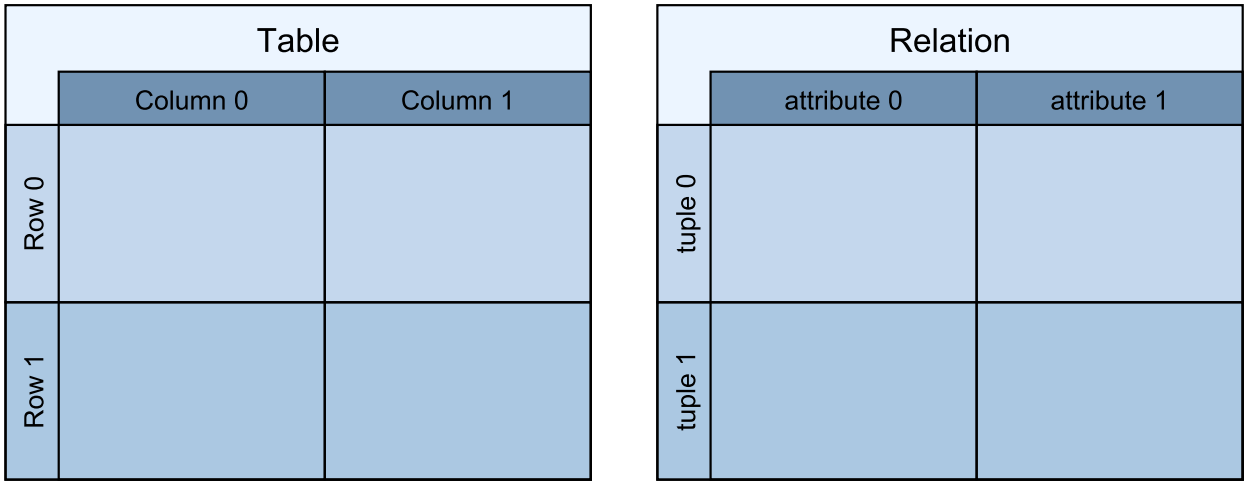
A database is an organized collection of structured information, or data, typically stored electronically in a computer system. Most databases are organized like a dictionary in the sense that they map from keys to values. The biggest difference is that the database is on disk (or other permanent storage), so it persists after the program ends. Because a database is stored on permanent storage, it can store far more data than a dictionary, which is limited to the size of the memory in the computer.

Like a dictionary, database software is designed to keep the inserting and accessing of data very fast, even for large amounts of data. Database software maintains its performance by building indexes as data is added to the database to allow the computer to jump quickly to a particular entry.

There are many different database systems which are used for a wide variety of purposes including: Oracle, MySQL, Microsoft SQL Server, PostgreSQL, and SQLite. We focus on SQLite in this book because it is a very common database and is already built into Python. SQLite is designed to be embedded into other applications to provide database support within the application. For example, the Firefox browser also uses the SQLite database internally as do many other products.

Database concepts

When you first look at a database it looks like a spreadsheet with multiple sheets. The primary data structures in a database are: tables, rows, and columns.



In technical descriptions of relational databases the concepts of table, row, and column are more formally referred to as relation, tuple, and attribute, respectively.

Databases require more defined structure than Python lists or dictionaries.

When we create a database table we must tell the database in advance the names of each of the columns in the table and the type of data which we are planning to store in each column. When the database software knows the type of data in each column, it can choose the most efficient way to store and look up the data based on the type of data.

Various data types supported by SQLite(other databases might have other data types)

SQLLight3 data types

Example Typenames From The CREATE TABLE Statement or CAST Expression	Resulting Affinity
INT INTEGER TINYINT SMALLINT MEDIUMINT BIGINT UNSIGNED BIG INT INT2 INT8	INTEGER
CHARACTER(20) VARCHAR(255) VARYING CHARACTER(255) NCHAR(55) NATIVE CHARACTER(70) NVARCHAR(100) TEXT CLOB	TEXT
BLOB no datatype specified	BLOB
REAL DOUBLE DOUBLE PRECISION FLOAT	REAL
NUMERIC DECIMAL(10,5) BOOLEAN DATE DATETIME	NUMERIC

The code to create a database file and a table named **Students** with two columns in the database is as follows:

db connect

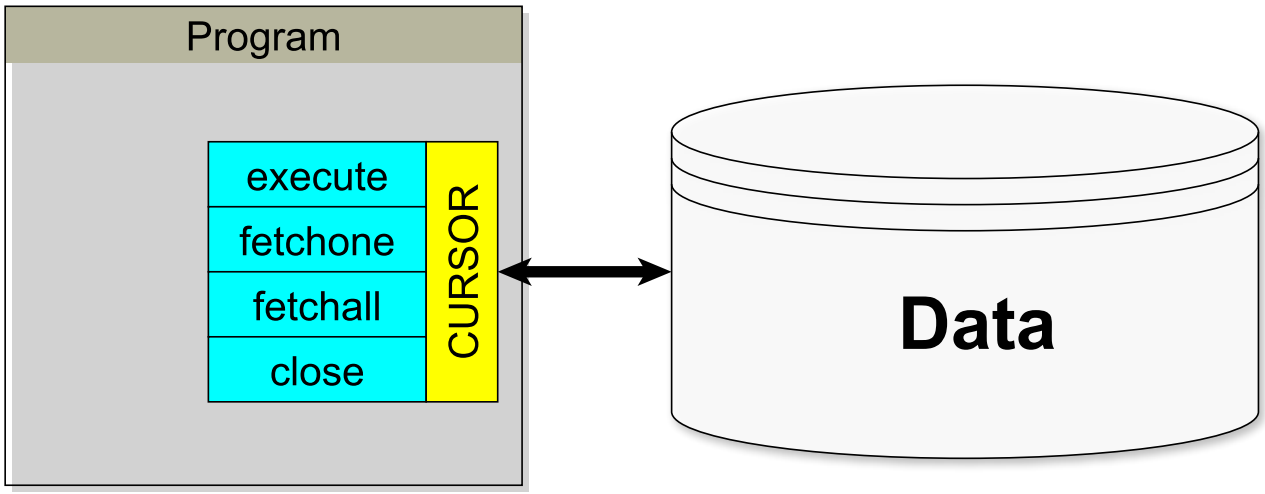
```
#!/usr/bin/python3
import sqlite3
if __name__ == '__main__':
   try:
      conn = sqlite3.connect('Students.sqlite')
      cur = conn.cursor()
   except:
      print("Error connecting to the data base! Exiting!")
      exit(0)


   cur.execute('DROP TABLE IF EXISTS cit138')
   cur.execute('CREATE TABLE cit138 (Student TEXT, grade REAL)')

   conn.close()
```

import sqlite3 imports the Sql ligbrary

conn = sqlite3.connect('Students.sqlite') Connects to the database file and reads in some basic information used by the library(version size,....)

cur = conn.cursor() actually opens the database file for our use. Think of it as the file pointer that we used to open files in previous lessons. A database cursor can be thought of as a pointer to a specific row within a query

result. The pointer can be moved from one row to the next. Depending on the type of cursor, you may be even able to move it to the previous row.



cur.execute('DROP TABLE IF EXISTS cit138') The first SQL command removes the cit138 table from the database if it exists. This pattern is simply to allow us to run the same program to create the cit138 table over and over again without causing an error. Note that the DROP TABLE command deletes the table and all of its contents from the database (i.e., there is no "undo").

cur.execute('CREATE TABLE cit138 (Student TEXT, grade REAL)') The second command creates a table named cit138 with a text column named Student and an real column named grade.

Now that we have created a table named Tracks, we can put some data into that table using the SQL INSERT operation. Again, we begin by making a connection to the database and obtaining the cursor. We can then execute SQL commands using the cursor.

The SQL **INSERT** command indicates which table we are using and then defines a new row by listing the fields we want to include **(Student, grade)** followed by the **VALUES** we want placed in the new row. We specify the values as question marks **(?, ?)** to indicate that the actual values are passed in as a tuple **('Mary', 95.7)** as the second parameter to the execute() call.

select sql

```
#!/usr/bin/python3
import sqlite3
if __name__ == '__main__':
    try:
        conn = sqlite3.connect('Students.sqlite')
        cur = conn.cursor()
    except:
        print("Error connecting to the data base! Exiting!")
        exit(0)


    cur.execute('DROP TABLE IF EXISTS cit138')
    cur.execute('CREATE TABLE cit138 (Student TEXT, grade REAL)')
    cur.execute('INSERT INTO cit138 (Student, grade) VALUES (?, ?)', ('Mary', ↙
        ↳  95.7))
    cur.execute('INSERT INTO cit138 (Student, grade) VALUES (?, ?)', ('Bob', ↙
        ↳  93.1))
    cur.execute('INSERT INTO cit138 (Student, grade) VALUES (?, ?)', ('Joe', ↙
        ↳  34.9))
    conn.commit()
```

```
18     print('Students:')
19     cur.execute('SELECT Student, grade FROM cit138')
20
21     for row in cur:
22        print(row)
23
24     cur.execute('DELETE FROM cit138 WHERE grade < 50.0')
25     conn.commit()
26     cur.execute('SELECT Student, grade FROM cit138')
27     print('Students:')
28     for row in cur:
29        print(row)
30
31     print("----------------------------------------\n\n")
32     #making it look nicer
33     widths = []
34     columns = []
35     tavnit = '|'
36     separator = '+'
37     cur.execute('SELECT Student, grade FROM cit138')
38
39
40     for cd in cur.description:
41        widths.append(len(cd[0]))
42        columns.append(cd[0])
43
44     for w in widths:
45        tavnit += " %-" + "%ss |" % (w,)
46        separator += '-' * w + '--+'
47
48     print(separator)
49     print(tavnit % tuple(columns))
50     print(separator)
51     for row in cur:
52        print(tavnit % row)
53     print(separator)
54
55     conn.close()
```

Output

```
Students:
('Mary', 95.7)
('Bob', 93.1)
('Joe', 34.9)
Students:
('Mary', 95.7)
('Bob', 93.1)
----------------------------------------


+---------+-------+
| Student | grade |
+---------+-------+
| Mary    | 95.7  |
| Bob     | 93.1  |
+---------+-------+
```

The SQL **SELECT** statement returns a result set of records from one or more tables

The **SELECT** statement has many optional clauses:

- **FROM** specifies which table to get the data.
- **WHERE** specifies which rows to retrieve.
- **GROUP** BY groups rows sharing a property so that an aggregate function can be applied to each group.
- **HAVING** selects among the groups defined by the GROUP BY clause.
- **ORDER** BY specifies an order in which to return the rows.
- **AS** provides an alias which can be used to temporarily rename tables or columns.

Given the following table:

Data

```
+---------+-------+
| Student | grade |
+---------+-------+
| Mary    | 95.7  |
| Bob     | 93.1  |
| Joe     | 34.9  |
| Julia   | 84.9  |
| Andrew  | 98.4  |
| Bill    | 79.8  |
+---------+-------+
```

cur.execute('SELECT * from cit138')

Output

```
+---------+-------+
| Student | grade |
+---------+-------+
| Mary    | 95.7  |
| Bob     | 93.1  |
| Joe     | 34.9  |
| Julia   | 84.9  |
| Andrew  | 98.4  |
| Bill    | 79.8  |
+---------+-------+
```

cur.execute('SELECT Student from cit138')

Output

```
+---------+
| Student |
+---------+
| Mary    |
| Bob     |
| Joe     |
| Julia   |
| Andrew  |
| Bill    |
+---------+
```

cur.execute('SELECT * from cit138 WHERE grade <80.0')

Output

```
+---------+-------+
| Student | grade |
+---------+-------+
| Joe     | 34.9  |
| Bill    | 79.8  |
+---------+-------+
```

cur.execute('SELECT * from cit138 WHERE grade <80.0 AND grade > 40.0')

Output

```
+---------+-------+
| Student | grade |
+---------+-------+
| Bill    | 79.8  |
+---------+-------+
```

cur.execute('SELECT * from cit138 ORDER BY grade')

Output

```
+---------+-------+
| Student | grade |
+---------+-------+
| Joe     | 34.9  |
| Bill    | 79.8  |
| Julia   | 84.9  |
| Bob     | 93.1  |
| Mary    | 95.7  |
| Andrew  | 98.4  |
+---------+-------+
```

cur.execute('SELECT Student as XYZABC123, grade as Total_grade from cit138')

Output

```
+-----------+-------------+
| XYZABC123 | Total_grade |
+-----------+-------------+
| Mary      | 95.7        |
| Bob       | 93.1        |
| Joe       | 34.9        |
| Julia     | 84.9        |
| Andrew    | 98.4        |
| Bill      | 79.8        |
+-----------+-------------+
```

One way we can look at the select statement is that it represents a loop. This loop will iterate through all the data in our table. The where portion is essentially an IF statement. if we were to represent our table as a nested dictionary and list we could rewrite sql into Python code.

select sql

```
1 #!/usr/bin/python3
2 import sqlite3
```

```
3  def pretty_print_sql(cur):
4     widths = []
5     columns = []
6     tavnit = '|'
7     separator = '+'
8
9
10    for cd in cur.description:
11       widths.append(len(cd[0]))
12       columns.append(cd[0])
13
14    for w in widths:
15       tavnit += " %-" + "%ss |" % (w,)
16       separator += '-' * w + '--+'
17
18    print(separator)
19    print(tavnit % tuple(columns))
20    print(separator)
21    for row in cur:
22       #print(row)
23       print(tavnit % row)
24    print(separator)
25
26 def pretty_print_dictionary(table,rows):
27    #get the header information. Dictionary keys
28    widths = []
29    columns = []
30    tavnit = '|'
31    separator = '+'
32    for cd in table.keys():
33       widths.append(len(cd))
34       columns.append(cd)
35
36    for w in widths:
37       tavnit += " %-" + "%ss |" % (w,)
38       separator += '-' * w + '--+'
39    print(separator)
40    print(tavnit % tuple(columns))
41    print(separator)
42    i=0
43    all_keys = list(table.keys())
44    #print(all_keys)
45
46    while(i<rows):
47       #print(i)
48       all_data_in_row =[]
49       for k in all_keys:
50          all_data_in_row.append(table[k][i])
51          #print(table[k][i])
52       #print(tuple(all_data_in_row))
53       print(tavnit % tuple(all_data_in_row))
54       i=i+1
55    print(separator)
56
57 if __name__ == '__main__':
58    '''
59    try:
60       conn = sqlite3.connect('Students.sqlite')
61       cur = conn.cursor()
62    except:
```

```
63          print("Error connecting to the data base! Exiting!")
64          exit(0)
65
66
67      cur.execute('DROP TABLE IF EXISTS cit138')
68      cur.execute('CREATE TABLE cit138 (Student TEXT, grade REAL)')
69      cur.execute('INSERT INTO cit138 (Student, grade) VALUES (?, ?)', ('Mary', ↙
            ↳  95.7))
70      cur.execute('INSERT INTO cit138 (Student, grade) VALUES (?, ?)', ('Bob', ↙
            ↳  93.1))
71      cur.execute('INSERT INTO cit138 (Student, grade) VALUES (?, ?)', ('Joe', ↙
            ↳  34.9))
72      cur.execute('INSERT INTO cit138 (Student, grade) VALUES (?, ?)', ('Julia', ↙
            ↳  84.9))
73      cur.execute('INSERT INTO cit138 (Student, grade) VALUES (?, ?)', ('Andrew', ↙
            ↳  98.4))
74      cur.execute('INSERT INTO cit138 (Student, grade) VALUES (?, ?)', ('Bill', ↙
            ↳  79.8))
75      conn.commit()
76      #making it look nicer
77      cur.execute('SELECT * from cit138')
78
79      pretty_print_sql(cur)
80      conn.close()
81      '''
82      row_counter = 0     #used to keep tract of how many items we inserted
83      cit138_table = {}
84      #create two colums Student and garade
85      cit138_table["Student"] = []
86      cit138_table["grade"] = []
87      #do an insert
88      #cur.execute('INSERT INTO cit138 (Student, grade) VALUES (?, ?)', ('Mary', ↙
            ↳  95.7))
89      cit138_table["Student"].append('Mary')
90      cit138_table["grade"].append(95.7)
91      row_counter = row_counter + 1
92      #cur.execute('INSERT INTO cit138 (Student, grade) VALUES (?, ?)', ('Bob', ↙
            ↳  93.1))
93      cit138_table["Student"].append('Bob')
94      cit138_table["grade"].append(93.1)
95      row_counter = row_counter + 1
96      #cur.execute('INSERT INTO cit138 (Student, grade) VALUES (?, ?)', ('Joe', ↙
            ↳  34.9))
97      cit138_table["Student"].append('Joe')
98      cit138_table["grade"].append(34.9)
99      row_counter = row_counter + 1
100     #cur.execute('INSERT INTO cit138 (Student, grade) VALUES (?, ?)', ('Julia', ↙
            ↳  84.9))
101     cit138_table["Student"].append('Julia')
102     cit138_table["grade"].append(84.9)
103     row_counter = row_counter + 1
104     #cur.execute('INSERT INTO cit138 (Student, grade) VALUES (?, ?)', ('Andrew', ↙
            ↳  98.4))
105     cit138_table["Student"].append('Andrew')
106     cit138_table["grade"].append(98.4)
107     row_counter = row_counter + 1
108     #cur.execute('INSERT INTO cit138 (Student, grade) VALUES (?, ?)', ('Bill', ↙
            ↳  79.8))
109     cit138_table["Student"].append('Bill')
110     cit138_table["grade"].append(79.8)
```

```
    row_counter = row_counter + 1
    print(cit138_table)
    pretty_print_dictionary(cit138_table,row_counter)
    print("_________________________________")
    print("SELECT * from cit138 WHERE grade <80.0")
    print("---------------------------------")
    #write code for cur.execute('SELECT * from cit138 WHERE grade <80.0')
    cursor={}   # create a new result set
    #select *  # get all the headers
    for header in cit138_table.keys():
       cursor[header]= []

    #print(cursor)
    #find the all the grades
    #keep track of rows
    i = 0
    total_cursor_rows = 0
    for g in cit138_table['grade']:
       if(g<80.0):
           cursor['Student'].append(cit138_table['Student'][i])
           cursor['grade'].append(cit138_table['grade'][i])
           total_cursor_rows = total_cursor_rows + 1
       i = i + 1

    #print(cursor)
    pretty_print_dictionary(cit138_table, total_cursor_rows)
```

Output

```
{'Student': ['Mary', 'Bob', 'Joe', 'Julia', 'Andrew', 'Bill'], 'grade': [95.7, 93.1, 34.9, 84.9, 9
+---------+-------+
| Student | grade |
+---------+-------+
| Mary    | 95.7  |
| Bob     | 93.1  |
| Joe     | 34.9  |
| Julia   | 84.9  |
| Andrew  | 98.4  |
| Bill    | 79.8  |
+---------+-------+

---------------------------------
SELECT * from cit138 WHERE grade <80.0
---------------------------------
+---------+-------+
| Student | grade |
+---------+-------+
| Mary    | 95.7  |
| Bob     | 93.1  |
+---------+-------+
```

From the above code you can see that we can accomplish the same result using lists and dictionaries. However, The above code is not generic enough to handle any type of data and it will become more and more complex as we try to do JOINS and other SQL data retrieval.

More complex SQl such as getting data from two tables that have data in common.
Lets create a table for two classes CIT138 and ENG111. What we want to know is all the students that are taking both classes. For this we will need to have some unique identifies in both tables. These unique identifiers of data are called KEYS. (There are many different types of KEYS in databases most will be beyond the scope of the tutorial. Here we will just care about Unique KEYS)

Each Student has a Unique identifier in the college. We call them ”A” numbers.

select sql

```
#!/usr/bin/python3
import sqlite3
def pretty_print_sql(cur):
    widths = []
    columns = []
    tavnit = '|'
    separator = '+'


    for cd in cur.description:
        widths.append(len(cd[0]))
        columns.append(cd[0])

    for w in widths:
        tavnit += " %-" + "%ss |" % (w,)
        separator += '-' * w + '--+'

    print(separator)
    print(tavnit % tuple(columns))
    print(separator)
    for row in cur:
        #print(row)
        print(tavnit % row)
    print(separator)

if __name__ == '__main__':

    try:
        conn = sqlite3.connect('Students.sqlite')
        cur = conn.cursor()
    except:
        print("Error connecting to the data base! Exiting!")
        exit(0)


    cur.execute('DROP TABLE IF EXISTS cit138')
    cur.execute('CREATE TABLE cit138 (A_no TEXT,Student TEXT, grade REAL)')
    cur.execute('INSERT INTO cit138 (A_no,Student, grade) VALUES (?, ?, ?)', ('A001','Mary', 95.7))
    cur.execute('INSERT INTO cit138 (A_no,Student, grade) VALUES (?, ?, ?)', ('A002','Bob', 93.1))
    cur.execute('INSERT INTO cit138 (A_no,Student, grade) VALUES (?, ?, ?)', ('A003','Joe', 34.9))
    cur.execute('INSERT INTO cit138 (A_no,Student, grade) VALUES (?, ?, ?)', ('A004','Julia', 84.9))
    cur.execute('INSERT INTO cit138 (A_no,Student, grade) VALUES (?, ?, ?)', ('A005','Andrew', 98.4))
    cur.execute('INSERT INTO cit138 (A_no,Student, grade) VALUES (?, ?, ?)', ('A006','Bill', 79.8))
    conn.commit()

    cur.execute('DROP TABLE IF EXISTS eng111')
    cur.execute('CREATE TABLE eng111 (A_no TEXT,Student TEXT, grade REAL)')
    cur.execute('INSERT INTO eng111 (A_no,Student, grade) VALUES (?, ?, ?)', ('A001','Mary', 95.7))
```

```
50      cur.execute('INSERT INTO eng111 (A_no,Student, grade) VALUES (?, ?, ?)', ('↙
            ↳  A030','Rose', 93.1))
51      cur.execute('INSERT INTO eng111 (A_no,Student, grade) VALUES (?, ?, ?)', ('↙
            ↳  A070','Dianne', 34.9))
52      cur.execute('INSERT INTO eng111 (A_no,Student, grade) VALUES (?, ?, ?)', ('↙
            ↳  A090','Robert', 84.9))
53      cur.execute('INSERT INTO eng111 (A_no,Student, grade) VALUES (?, ?, ?)', ('↙
            ↳  A034','Phillip', 98.4))
54      cur.execute('INSERT INTO eng111 (A_no,Student, grade) VALUES (?, ?, ?)', ('↙
            ↳  A006','Bill', 79.8))
55      conn.commit()
56
57      print("CIT138")
58      cur.execute('SELECT * from cit138')
59
60      pretty_print_sql(cur)
61      print("ENG111")
62      cur.execute('SELECT * from eng111')
63
64      pretty_print_sql(cur)
65
66      cur.execute('SELECT cit138.Student from eng111,cit138 where cit138.A_no = ↙
            ↳  eng111.A_no')
67      pretty_print_sql(cur)
68      conn.close()
```

Output

```
CIT138
+------+---------+-------+
| A_no | Student | grade |
+------+---------+-------+
| A001 | Mary    | 95.7  |
| A002 | Bob     | 93.1  |
| A003 | Joe     | 34.9  |
| A004 | Julia   | 84.9  |
| A005 | Andrew  | 98.4  |
| A006 | Bill    | 79.8  |
+------+---------+-------+
ENG111
+------+---------+-------+
| A_no | Student | grade |
+------+---------+-------+
| A001 | Mary    | 95.7  |
| A030 | Rose    | 93.1  |
| A070 | Dianne  | 34.9  |
| A090 | Robert  | 84.9  |
| A034 | Phillip | 98.4  |
| A006 | Bill    | 79.8  |
+------+---------+-------+
+---------+
| Student |
+---------+
| Mary    |
| Bill    |
+---------+
```

cur.execute('SELECT cit138.Student from eng111,cit138 where cit138.A_no = eng111.A_no')

The above select statement did all the work for us in finding students that were taking both classes. If we had to

write this out using dictionaries and Lists plus all kinds of loops keeping track of both tables the code would be very complex and possibly error prone.
ALLWAYS choose the right tool for the job. In cases where we want to store data and get information from it databases are the best tool for the job. We have barely scratched the surface of what is possible with databases and would take us a whole semester just to get the basics down.