Intermediate Python

Programming for Professionals

LOGISTICS

Class Hours:

- Start time is 9am
- End time is 4:30pm
- Class times may vary slightly for specific classes
- Breaks mid-morning and afternoon (10 minutes)

Lunch:

- Lunch is 11:45am to 1pm
- Yes, 1 hour and 15 minutes
- Extra time for email, phone calls, or simply a walk.

Telecommunication:

- Turn off or set electronic devices to vibrate
- Reading or attending to devices can be distracting to other students
- Try to delay until breaks or after class

Miscellaneous

- Courseware
- Bathroom
- Fire drills

ISHE DORO

Software Engineer

Experienced programmer in C++, Python & C#

7+ Years Designing & Developing Applications

Microsoft Certified Trainer

ishe@innovationinsoftware.com

INTERMEDIATE PYTHON

Python is a unique programming language and developed for casual, devops, and professional developers.

For a devops person, Python is a scripting language that is simple and readable. For example, declaration-less variables remove substantial clutter from a Python application.

For a senior developer, Python is much more than a scripting language and has the full capability of other modern languages, such as Java, C#, or C++. You can use Python to create sophisticated applications: financial, scientific, retail, data management, blockchain, and more.

This course presents the topics necessary to create a complete Python application from coding to testing.

INTRODUCTION

- Immersion is the best approach to learn a language. This class will immerse you in Python for three days.
- Learning by doing is important. Be prepared for a hands-on experience.
- This class is not theoretical. Yes, there is some theory. But that is secondary to learning about developing real-world application using the Python language.

TARGET ENVIRONMENT

Python tools provide a common experience (as close as possible) regardless of the environment. This includes agnostic tools, such as the Idle IDE (integrated development environment).

The target environments are:

- Windows
- Mac OS X
- Linux

For class, there is no preferred environment. Feel free to use your preferred IDE. However, it is expected that you will *know* how to use your selected IDE.

LABS

Learning is better when hands-on. Each module has a companion lab reinforcing a kinesthetic learning experience.

- Labs review and reinforce important concepts
- Don’t be surprised - many of the labs extend the topics introduced in the module
- The labs offer real-world experience
- Pair programming can be effective when working on the labs

COURSE AGENDA

1. Operating system integration
2. Classes
3. Database
4. Meta-Programming
5. Python libraries
6. Time and randomization
7. Threading
8. Regular Expressions
9. Machine Learning
10. Unit testing

YOUR CLASS!

Yes, this is your class. What does this mean? You define the value.

- What is the most important ingredient of class – your participation!
- Your feedback and questions are always welcomed
- There is no protocol in class. Speak up anytime.
- We welcome your comments during and after class. Just email ishe@innovationinsoftware.com.

Ready, Set, Go!

Python Fundamentals

MAIN

Many languages have an entry-point method, which is normally “main”.

The program starts with main and ends when main exits. This is the conventional programming paradigm. You can adopt this paradigm also.

A module can run independently or imported as a library. Main should be the entry point only if the module is started independently not imported. This can be confirmed with the module variable __*name*__. If __name__ == “__main__”, the module is running independently and the entry method (i.e., main) should be called.

Here is code to properly call the entry point method.

```
import sys

def main():

    print("in main")



if  __name__ == "__main__":

    sys.exit(main())
```

DATA TYPES

Python has an assortment of types. Here are the most common:

- int
- float
- str
- bool

DATA TYPES

Which other built-in data types do you know?

RUNTIME TYPE INFO

You can confirm the type of an object at runtime with either the type or isinstance method. Type returns the kind of type for an instance; while isinstance returns the confirmation of a type as a boolean.

```
>>> a=5
>>> type(a)
<class 'int'>
>>> isinstance(a, int)
True
>>>
```

NUMERIC TYPES AND OPERATORS

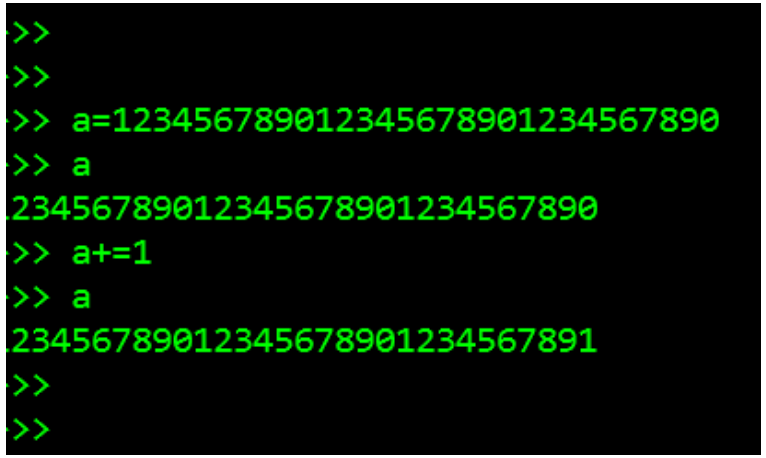
Here are the standard numeric operators.

Note: Division returns a float even if both operands are integer. If desired, cast the results to an integer.

Operation	Operator
Addition	*
Subtraction	-
Multiply	*
Division	/
Remainder	%
Integer Division	//
Exponentiation	**

INTEGER

Integral data types in Python are now just integers (i.e., not short or long). There is no concept for example of 32- or 64-bit integers. Integers in Python are signed and unlimited in size. Yes, this is true – unlimited!



FLOAT

Floats are floating point values and are usually implemented as a C double. sys.float_info provides the details about floats for a specific Python implementation. Located in the sys module.

The field sys.float_info.dig returns the number of significant digits supported for a float.

```
>>>
>>> import sys
>>> sys.float_info.dig
15
>>>
```

BOOLEAN

True and False represent the Boolean values. The names are case sensitive.

Booleans are considered a subtype of integers, for which 0 is false and non-zero is true.

In addition, anything that is *empty* is False.

```
>>>
>>> bool(0)
False
>>> bool(5)
True
>>> bool(-1)
True
>>> bool("")
False
>>> bool("hello")
True
>>> bool(list())
False
>>> bool([1,2,3])
True
```

BOOLEAN OPERATORS

Here are the boolean operators.

Boolean	Operator
Equal	==
Greater than	>
Less than	<
Greater than or equal	>=
Less than or equal	<=
Not equal	!=
And	and
Or	or
Not	not
In sequence	in

CONDITIONALS

There are several types of conditional statements. With each conditional statement, there is a colon suffix. The subsequent lines of source code are indented to indicate the block of code associated with the statement:

- if: indented statements executed if true.
- else: optional false branch of an if statement
- elif: compound if statements instead of nested if statements

```
status="executive"
if(status.lower()=="exempt"):
      print("Paid Salary + Bonus")
else:
      if(status.lower()=="hourly"):
            print("Pay Hourly + overtime")
      else:
            if(status.lower()=="executive"):
                  print("Pay Salary + Bonus + Stock")
```

SWITCH STATEMENT

There is no switch statement in Python. You must use the elif instead.

```
today="FRi"
today=today.capitalize()
print(today)
if today=="Sun":
        print("Start of the week")
elif today=="Mon":
        print("Start of work week")
elif today=="Tue":
        print("Another day")
elif today=="Wed":
        print("Hump day")
elif today=="Thu":
        print("Almost over")
elif today=="Fri":
        print("Time for celebration")
elif today=="Sat":
        print("All day / all fun")
else:
        print("This is a weird day!")
```

STRINGS

Strings are UNICODE in Python 3 with UTF-8 encoding.

You can embed strings with hidden characters using the back slash character, such as “\n” and “\t”.

Raw strings begin with the “r” prefix and treat hidden characters sequences as normal characters.

Use len method to determine the length of a string.

Description	Sequence
bell	\a
backspace	\b
form feed	\f
line feed	\n
carriage return	\r
tab	\t
16-bit Unicode point	\Uxxxx
32-bit Unicode point	\Uxxxxxxxx

STRINGS – 2

You can cast strings to bytes. Conversely, the decode method will convert bytes into a string.

Len returns the number of characters (code points in a string). This is different from the number of bytes.

```
helloinjapanese="こんにちは"

print(len("こんにちは"))

byteHello=bytes(helloinjapanese, "UTF-8")

print(len(byteHello))
```

STRINGS ARE SEQUENCES

A string is a sequence of UNICODE characters. You can access the string as a unit or inspect the string as a sequence.

You can return substrings (slices) via indexing.

```
>>> greeting="Hello!"
>>> for character in greeting:
...     print(character, end=" ")
...
H e l l o ! >>>
>>>
>>> greeting[0]
'H'
>>> greeting[-1]
'!'
>>> greeting[-2:-1]
'o'
>>> greeting[1::2]
'el!'
>>>
```

STRING TO BYTES

create a multi-culture string (1)

create bytes from Unicode string (2)

second alternative (3)

create bytes from ascii – this will fail (4)

dump bytes for Unicode string (5)

dump bytes for ASCII string (6)

```
st = "hello привет नमस्ते"                    1
test=bytearray(st, "utf-8")                   2
print(list(test))

_bytes=bytes(st, "utf-8")                     3

asciitest=bytes(st, "ascii")                  4

print(_bytes)
for x in _bytes:                              5
    print(x)

print(asciitest)
for x in _bytes:                              6
  print(x)
```

STRING FUNCTIONS

Here are common string functions:

capitalize()	Capitalize first letter of string
center(width, fill)	Center within *width* of characters
count(str, begin=0, end=len(string))	Count instances of a substring
decode(encoding='UTF-8', error='strict')	Decode string using specific codec
encode(encoding='UTF-8', error='strict')	Returns encoded version of string
endswidth(suffix, begin=0, end=len(string))	Confirm substring at end of string
find(str, begin=0, end=len(string))	Locate substring and return index to location
isalnum()	Is string alphanumeric?
isalpha()	Is string alphabetic?
isdigit()	Is string all digits?
isnumeric()	Is string is numeric?
join(sequence)	Joins items of a collection, with string as the delimiter

STRING FUNCTIONS - 2

lower()	Convert to lowercase
lstrip()	Remove leading spaces
replace(old, new [,max])	Replace substring (old) with another substring
rfind(str, begin=0, end=len(string))	Reverse find
rjust(width [,fill])	Right justify string
rstrip()	Remove trailing spaces
split(str=“ ”, num=string.count(str))	Split string based on delimiter
startswith(str, begin=0, end=len(string))	Confirm substring at beginning of string
strip([chars])	Strip leading and trailing spaces
upper()	Converts string to uppercase

ASSIGNMENT AND MUTABILITY

Variables are references to a primitive value in memory. When assigning a variable with the value of another, you are copying the reference; not the value at that location. You can confirm this with the id method, which returns the reference to a variable.

```
a = 5
b = 10

print('A =',id(a))
print('B =',id(b))
b = a

print('A =',id(a))
print('B =',id(b))
```

✓ 0.4s

```
A = 1480675098992
B = 1480675099152
A = 1480675098992
B = 1480675098992
```

CONVERSION

Python provides methods to convert from one type to another (cast):

- int(value)
- float(value)
- bool(value)
- str(value)

SEQUENCES

These sequences are predefined in Python.

- List
- Tuple
- Dictionary
- Set

LIST

A list is a sequence of potentially disparate types.

Lists are defined with [].

- list() or [] to define an empty list
- Convert to list with list()
- As with any sequence, can be nested.
- Lists are mutable. For this reason, list has methods such as append and extend.

```
names=["Carol", "Bob", "Ted", "Alice"]
actors=["Natalie Wood", "Robert Culp", \
     "Elliot Gould", "Dyan Cannon"]
role=zip(actors, names)
for person in role:
     print(person)
```

TUPLE

Tuples are immutable lists. Tuples are 3 to 7 times faster than a list! If you have a static list, such as the list of states in the United States, create a tuple. The list of states in the U. S. has not changed in a while!

- Tuples are defined with ().
- tuple() or () to define an empty list
- Convert to tuple with tuple()
- Tuple of 1: (12,)

```
>>> states=("NY", "NJ", "CA", "CO", "...")
>>> states.append("DC")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>>
```

SET

A set is a mutually exclusive list. Alternatively stated, a set is a unique list of items.

The strength of sets are the set specific methods:

- intersection (&)
- union (|)
- difference (-)
- exclusive (^)
- issubset (<=)
- issuperset (>=)

```
>>>
>>> color1={"Red", "Blue", "Green"}
>>> color2={"Red", "Yellow", "Purple"}
>>> color1 | color2
{'Red', 'Yellow', 'Purple', 'Blue', 'Green'}
>>> color1 ^  color2
{'Purple', 'Yellow', 'Blue', 'Green'}
>>> color1.intersection(color2)
{'Red'}
>>>
```

SET - 2

- Sets are defined with {}
- Set() to define an empty list; but not {}
- The {} already committed to the dictionary sequence
- Convert to tuple with tuple()

LOCAL, ENCLOSED, GLOBAL, AND BASE (LEGB)

In Python, namespaces define scope for objects. Everything in Python is an object: primitives, functions, classes, dictionaries, and more. Each object is mapped to a namespaces.

Each namespace is independent. However, namespaces can be nested.

The same object can be found in more than one namespace. When overlapping namespaces have the same object what is the persistence and scope (visibility) of the object?

Persistence is easy: the lifetime of the namespace.

Scope is more complex and defined by LEGB.

LEGB ORDER

LEGB defines the order of evaluation when the same object or variable appears in overlapping namespaces.

1. Local
2. Enclosed
3. Global
4. Built-in

The keywords, such as global and nonlocal, are used to revise visibility.

POP QUIZ: NAMESPACES

Is a *for loop* considered a separate namespace?

5 MINUTES

LEGB – SAMPLE CODE

What is the result of this code?

```
i=1


def foo():
        i=5
        print(i, "in foo()")


print(i, "global")


foo()
```

LEGB – SAMPLE CODE - 2

Here is the result of the previous code is:

```
1 global
5 in foo()
```

You can confirm namespace membership with the locals and globals functions.

```
glob = 1


def foo():
    loc = 5


print('loc in foo():', \
      'loc' in locals())
foo()
print('loc in global:', \
      'loc' in globals())


print('glob in global:', \
      'foo' in globals())
```

LEGB – SAMPLE CODE - 3

```
loc in foo(): False
loc in global: False
glob in global: True
```

```
glob = 1

def foo():
    loc = 5

print('loc in foo():', \
      'loc' in locals())
foo()
print('loc in global:', \
      'loc' in globals())

print('glob in global:', \
      'foo' in globals())
```

LEGB – SAMPLE CODE - 4

What is the result of this code?

```
a_var = 'global value'
def a_func():
     a_var = 'local value'
     print(a_var, '[ a_var inside a_func() ]')
a_func()
print(a_var, '[ a_var outside a_func() ]')
```

LEGB – SAMPLE CODE - 5

You can override the scope of a variable with the global keyword.

In this example, the local a_var is overridden with the global representation.

What is the result of this code?

```
a_var = "global value"


def a_func():
    global a_var
    a_var = "global value 2"
    print(a_var, "[ a_var inside a_func() ]")


print(a_var, "[ a_var outside a_func() ]")
a_func()
print(a_var, "[ a_var outside a_func() ]")
```

LEGB – SAMPLE CODE - 6

Here is the result of the previous code is:

```
global value [ a_var outside a_func() ]
global value 2 [ a_var inside a_func() ]
global value 2 [ a_var outside a_func() ]
```

The scope of code within a nested function is enclosed.

What is the result of this code?.

```
a_var = "global value"


def outer():
    a_var = "enclosed value"


    def inner():
        a_var = "local value"
        print(a_var)


    inner()


outer()
```

LEGB – SAMPLE CODE - 7

Here is the result of the previous code is:

```
local value
```

You can override the enclosed scope with either the global or nonlocal keywords. The nonlocal keyword is used to override scope with a non global variable.

```
a_var = "global value"


def outer():
    a_var = "local value"
    print("outer before:", a_var)


    def inner():
        nonlocal a_var
        a_var = "local value 2"
        print("in inner():", a_var)
    inner()
    print("outer after:", a_var)


outer()
```

LEGB – SAMPLE CODE - 8

```
outer before: local value
in inner(): local value 2
outer after: local value 2
```

```
a_var = "global value"


def outer():
    a_var = "local value"
    print("outer before:", a_var)


    def inner():
        nonlocal a_var
        a_var = "local value 2"
        print("in inner():", a_var)
    inner()
    print("outer after:", a_var)


outer()
```

POP QUIZ: WHAT IS THE RESULT

```
a_var = 1

def a_func():
    a_var = a_var + 1
    print(a_var, \
      "[ a_var inside a_func() ]")

print(a_var, \
      "[ a_var outside a_func() ]")
a_func()
```

5 MINUTES

LEGB – SAMPLE CODE - 9

Overloading built-in objects is typically not advisable but remains possible. Look at the sample code presented here.

```
sequence=[1,2,3,4]


def Interesting():
    def len(sequence):
        print("wrong len")


    len(sequence)


Interesting()


print(len(sequence))
```

LOOPS

There are two common loops in Python

- for
- while

WHILE

With the while statement, the while block is iterated until the while condition is False. The while loop is iterated zero or more times. If the initial condition is False, the loop is executed zero times.

Here is the syntax:

```
while(condition):

  block
```

Sample code:

```
while(var!=0):

  print(var)

  var-=1
```

FOR LOOP

The *for* loop iterates the elements of an iterative collection in order.

The syntax is:

```
for element in sequence
```

A sequence can be a list, set, tuple, or dictionary.

The *range* statement is a common way to also create a sequence.

```
>>>
>>> for element in range(30, 20, -3):
...     print(element)
...
30
27
24
21
>>>
```

LOOP MISCELLANEOUS

- *break* statement

  Ends loop immediately

- *continue* statement

  Continue to start of next iteration

- *else* statement

  Executed if loop completed naturally (did not break)

Ready, Set, Go!

Lab 1

LAB 1: SWAP VALUES

This lab:

- Swap two integer values

SWAP FUNCTION

1. Create a function that swaps two integer values
2. In some manner, the integer values should be passed as function parameters.
3. The function swaps the values in-place. The original variables should have their values exchanged.
4. Call the swap function with test values
5. Print the results.

Operating System

Integration

PROCESS INFORMATION

The *os* module has helpful methods to access system information:

- getpid

  Returns process id of the current process

- getppid

  Returns the process id for the parent process

- getlogin

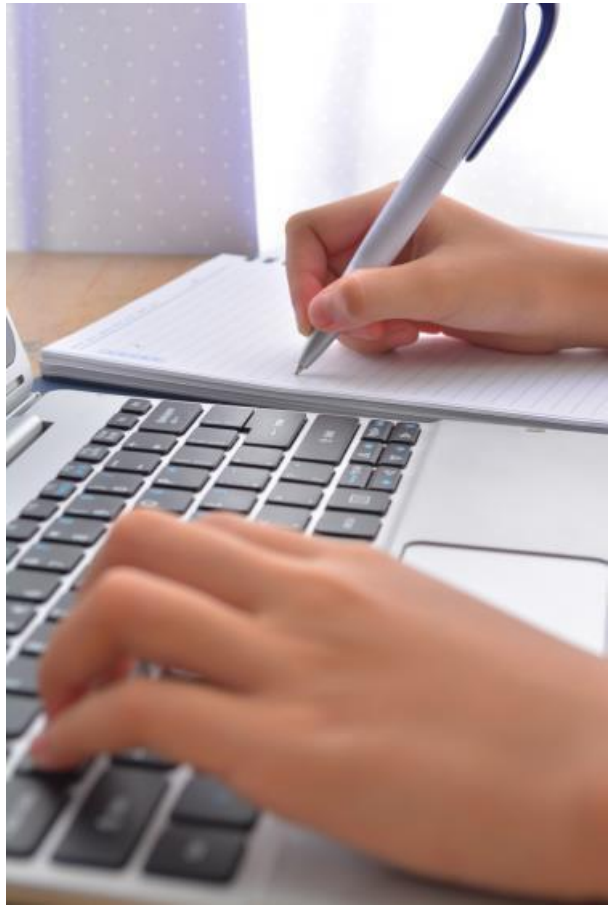
  Returns the name of the current user

- get_exec_path

  Returns the order that directories are searched to find an application when executed.

HANDS-ON EXERCISE

Process Information



PROCESS INFORMATION – HANDS ON

With this hands-on exercise, you open cmd as a parent process. Within cmd, run a python script (i.e., application). The Python script will display the process id (pid) for both the child and parent application.

Here are the steps. Open the command prompt (i.e., cmd.exe).

- Change to the appropriate directory
- From that directory, create a file with notepad (or other text editor): getinfo.py.

```
C:\>md somedir

C:\>cd somedir

C:\somedir>notepad getinfo.py

C:\somedir>
```

PROCESS INFORMATION – HANDS ON - 2

In getinfo.py:

- Import os
- Print the result of os.getpid() with formatting

```
print("Process PID %d"%os.getpid())
```

- Print the result of os.getppid() with formatting
- Pause program until some input entered; then exit.

```
input("Press any <key> to exit.")
```

Close and save the file

PROCESS INFORMATION – HANDS ON - 3

From the command prompt:

- Run getinfo.py
- View results

```
C:\>Getinfo.py
Process PID 32320
Process PID 13756
Press any <key> to exit.
```

Hand-on exercise
Done!

EXECUTE EXTERNAL APPLICATIONS

Python can execute external applications either synchronously and asynchronously. Synchronous is simpler. You launch the external application, block, and wait for the result. Asynchronous has additional features, such as polling for completion or terminating child application.

Python can capture the stdout, stderr, and stdin stream from the application. You can then read information from the application in real-time.

EXECUTE APPLICATIONS - SYNCHRONOUS

Synchronous execution of applications:

- os.popen
- os.system
- subprocess.call

OS.POPEN

Syntax:

```
file=os.popen(command,
      [,mode[,buffer]])
```

- command. command to execute
- mode. default ‘r’ for read. Also ‘w’ for write.
- buffer. 0 – no buffer. 1 – buffer enabled. > 1 set buffer size. < 0 system default.
- Returns a file object

```
>>>
>>> import os
>>> os.popen("echo Hello, World").read()
'Hello, World\n'
>>>
>>>
```

OS.SYSTEM

Syntax:

```
file=os.system(command)
```

- command. command to execute
- Returns error code, where 0 is successful.

```
>>>
>>> import os
>>> os.system("notepad.exe")
0
>>>
>>>
```

SUBPROCESS.CALL

Syntax:

```
subprocess.call(
      args, *, stdin=None,  stdout=None,
      stderr=None,  shell=False)
```

- args. list that contains command and parameters
- shell. should be considered obsolete

```
>>>
>>> import subprocess
>>> subprocess.call(["notepad.exe", "test.txt"])
0
>>>
>>>
>>>
```

EXTERNAL COMMANDS - ASYNCHRONOUS

Asynchronous execution of external commands:

- os.startfile
- subprocess.Popen

OS.STARTFILE

Syntax:

```
os.startfile(file)
```

- file. filename – opens based on application association
- Return value. error code.

```
>>>
>>> os.startfile(r"c:\code\test.txt")
>>>
```

SUBPROCESS.OPEN

Syntax:

`subprocess.Popen(…)`

- First parameter is a list: external program and parameters.
- Returns a popen object with the methods to control the child application, including *terminate*.

```
>>>
>>>
>>> p=subprocess.Popen(["notepad.exe"])
>>> p.terminate()
>>>
```

SUBPROCESS.POPEN – EXAMPLE CODE

```
import subprocess

     p = subprocess.Popen(r"python c:\code\names.py",

     shell=True, stdout=subprocess.PIPE,

     stderr=subprocess.STDOUT)


for line in p.stdout.readlines():

     print(line)


retval = p.wait()
```

ENVIRONMENT VARIABLES

The OS module also provides access to the environment variables.

- os.environ attribute

  List all environment variables as a dictionary

- os.environ[“name”]

  Get value of a specific variable

- os.environ[“name”]=“value”

ENVIRONMENT VARIABLES - 2

- environ.get(*name*)
  Alternative to environ[“name”]

- environ.get(*name, default*)
  If environment variable not available, set with the default value.

Environ supports the standard dictionary interface, including creating and setting new dictionary items (i.e., environment variables).

TIME COMMANDS

GET THE TIME

Time is an important part of scripting and automation. Time is used to schedule events, add delays, and for notifications. Python relies on Unix epoch for time, which measures time from 12am on January 1, 1970, Coordinated Universal time (UTC).

The time module provides access to the system clock of your computer. The time function returns the number of seconds since Unix epoch.

```
>>> import time
>>> time.time()
1534006780.5738091
>>> _
```

GET THE TIME - EXAMPLE

Time is great for profiling functions. In this example, Func is profiled. Time.sleep places the current thread to sleep (yields the CPU) for the specified number of seconds.

The start and end time are calculated for Func. The difference is the elapsed time, which is displayed.

```
import time

def Func():
    time.sleep(2)

start=time.time()
Func()
end=time.time()
elapse=int(end-start)

print("Elapsed time", elapse)
```

DATETIME

The datetime component is found in the datetime module naturally. The method datetime.now() returns the current time as a datetime object.

You can compare datetime objects using Boolean operators.

```
import datetime

current=datetime.datetime.now()

print(current.month, current.day, current.year)

print(current.hour, current.minute, current.second)
```

DATETIME - 2

You can convert from epoch to datetime using the datetime.fromtimestamp method. Conversely, datetime.timestamp method converts a datetime object to epoch.

```
import datetime

import time


epoch=time.time()

current=datetime.datetime.fromtimestamp(epoch)

print(current.month, current.day, current.year)

print(current.hour, current.minute, current.second)
```

TIMEDELTA

The datetime module also offers the timedelta type, which measures duration.

```
import datetime
obj = datetime.timedelta(days=12,hours=4,minutes=11,seconds=45)
print(str(obj))

obj = datetime.timedelta(days=12,hours=4,minutes=11,seconds=45)
print(obj.total_seconds())
```

✓ 0.3s

```
12 days, 4:11:45
1051905.0
```

You can even use timedelta to add time to a datetime object.

```
delta = datetime.timedelta(days=365*2.5)
datetime.datetime.now() + delta
```

✓ 0.3s

```
datetime.datetime(2025, 2, 25, 22, 18, 43, 971292)
```

CONVERTING TIME TO STRING

Using format specifiers, the datetime.strftime method displays a datetime as a string based on format specifiers.

```
date = datetime.datetime(2022,10,12,16,25,45)
date.strftime("%B, %w %Y")
```

✓ 0.2s

```
'October, 3 2022'
```

CONVERTING TIME TO STRING - 2

Here are the format specifiers for strftime. Note they are case sensitive.

%Y	Year with century	%y	year without century
%m	month (number)	%B	month(text)
%b	month (text abbreviated)	%j	day of year
%w	day of week (number)	%A	day of week (text)
%a	day of week (text abbreviated)	%H	hour (24)
%I	hour (12)	%M	minute
%S	second	%p	AM/PM

Operating System

Lab 2

LAB: SCREEN SHOT

This lab:

- Automate the steps for taking and saving a screenshot

CREATE AND SAVE SCREENSHOT

This lab automates the steps to create and save a screenshot to a file. This includes trapping a keyboard event.

You need to download two modules from the Python Package Index

- Pyautogui: This module specializes in GUI automation. For the lab, this module is used to create screenshots.
- Keyboard: This module is for hooking and managing keyboard events.

Use pip command to install the two modules:

```
pip install pyautogui

pip install keyboard
```

If already installed, you might want to upgrade.

```
python -m pip install --upgrade pip pyautogui
```

CREATE AND SAVE SCREENSHOT - 2

Define a function to create a screenshot.

1. Create a function called screenshot
2. Display message “creating screenshot…”
3. Call pyautogui.screenshot() to create screenshot. Screenshot returns an image object.
4. Save the image, with image.save, to a temporary file in a local directory.

Start application and setup keyboard event.

1. Call keyboard.write() to simulate keyboard input – write “Screenshot application running”
2. Associate a key with the screenshot function.

```
keyboard.add_hotkey('ctrl+shift+a’,
    savescreenshot, args=())
```

3. Associate a key with the screenshot function.
4. Call keyboard.wait to suspend until “esc” entered.

CREATE AND SAVE SCREENSHOT - 3

Run and test your program:

1. Run the application
2. Switch keyboard focus away from program (i.e., click in a different application)
3. Take screenshot of something using keyboard event
4. Exit program
5. Find and open image created for screenshot

File Manipulation

Text, Bytes, and JSON

FILE FUNDAMENTALS

Python provides a robust set of commands for file input-output. Python offers a file object, which supports an assortment of file operations. Be sure to close the file object when the file is no longer required.

You can open a file for:

- Read or write
- Text or binary
- Sequential or random

READ OR WRITE

Read or write to a file with the open command. The first parameter is the file. Use the second parameter, mode, to set the context of the operation, such as read (“r”) or (“w”). Mode is case sensitive.

Be sure to close files with the *close* function on the file object.

```
myfile=open(r"c:\code\myfile10.txt", "w")
myfile.write("Bob\n")
myfile.write("Ted\n")
myfile.write("Carol\n")
myfile.write("Alice\n")
myfile.close()


myfile=open(r"c:\code\myfile10.txt", "r")
names=list(myfile.readlines())
for name in names:
   print(name)
myfile.close()
```

READ OR WRITE - 2

```
myfile=open(r"c:\code\myfile11.txt", "w")

names=["Bob\n", "Ted\n", "Carol\n", "Alice\n"]

myfile.writelines(names)

myfile.close()


myfile=open(r"c:\code\myfile11.txt", "r")

names=list(myfile.readlines())
for name in names:
  print(name)
myfile.close()
```

TEXT OR BINARY

You can read or write to a file as text or binary. Use the second parameter, mode, to set the context of the operation, such as text (“t”) or binary (“b”). Mode is case sensitive.

Text is simpler and “clear text” some has advantages. Binary is quicker and slightly more secure. As shown previously, text is the default mode.

Be sure to close the file with *close* function.

Note: Pickle shown but not discussed. More about pickle soon.

```
import pickle

myfile=open('data', 'wb')
pickle.dump(9 ** 33, myfile)
myfile.close()
myfile=open('data', 'rb')
print(pickle.load(myfile))
myfile.close()
```

SEQUENTIAL OR RANDOM

For random access, open a file with the "r" mode. Use the seek and tell functions to access a file randomly while positioning the file pointer. Before reading or writing, move the file pointer with seek.

- seek(offset [,whence])

  offset. target location within file. 0 – absolute, 1 – relative, 2 – from end

- tell()

  returns location in file

```
myfile=open("greeting.txt", "w")

myfile.write("Hello")

myfile.close()

myfile=open("greeting.txt", "r")

#myfile.write("test")

print(myfile.tell())

myfile.seek(3)

print(myfile.tell())

data=myfile.read(2)

print(data)

myfile.close()
```

WITH

Efficient management of resources, such as files, is important in all programming languages, including Python. Files should be closed when no longer needed. Using the *with* clause, you can guarantee that a file is closed when the current block ends.

1. Open a file within the *with* clause
2. In the with block, access the file
3. When the block is exited, file is automatically closed

```
>>>
>>>
>>> with open("testfile.txt", "w") as thefile:
...     thefile.write("data")
...
4
>>> thefile.closed
True
>>>
>>>
```

File Manipulation

Lab 3

LAB: WORD COUNT

This lab:

- Count the number of unique words in a file

WORD COUNT

1. Using files on your local machine, read any file name from the command line: for example *python program.py xyz.txt*
2. Confirm file extension is JSON, HTML, TXT, or XML. If not, exit program.
3. Confirm that the file exists. If not, exit program.
4. Open the file and read entire file into a text buffer.
5. Iterate through words and add unique words to a dictionary. The key should be the word with count as the value.
6. If item already exists, add 1 to count.
7. When done, display results.

Extra Credit

Have fun!

LINK LIST

Create a circular linked list as a class.

- You should be able to store anything in the linked list.
- Start anywhere in the list and move forwards and backwards.
- Finally be able to rotate (circular) around from any starting point in the link list.

Here are the expected methods:

- Jump to beginning
- Jump to end
- Next
- Prev
- Insert
- Delete (logical)
- Count (with and without deleted)
- Compress (remove logically deleted nodes)

NOTES

- Be prepared to present solutions midday on last day of class.
- Suggest working as a team – at least pair programming
- Work on extra credit with free time during labs and when possible

Exceptions

Error Handling

ERROR HANDLING

Proactively isolating errors is preferred to exception handling.

- Assertion allows you to isolate future software bugs during development and debugging.
- Error handling is performed at runtime. It is a preference to exception handling. Exception handling is more expensive.

Assertion is being proactive while error handling reactive.

ASSERT

Assertion allow you to test assumptions of an application during development and debugging. For example, test the acceptable values of parameters or the return value. Test the condition using the assert function.

The assert function accepts a single parameter with is a Boolean expression. If false, the statement fails and an assertion occurs.

Assertion are executed in debug mode; otherwise it is ignored. Check if in debug mode using the __debug__ variable. If True, you are debugging.

ASSERT - EXAMPLE

```
def FuncA(a, b):
    if b==0:
        return 0
    print(a/b)

FuncA(5,0)
```

```
========== RESTART: C:\Code\Python\Test.py ==========
Traceback (most recent call last):
  File "C:\Code\Python\Test.py", line 5, in <module>
    FuncA(5,0)
  File "C:\Code\Python\Test.py", line 2, in FuncA
    assert(b != 0)
AssertionError
>>>
```

ASSERT – EXAMPLE – 2

Execution of the program in debug mode.

Run the program in release mode with the –O compilation option. Notice that the exception occurs and the assertion is ignored.

```
C:\Code\Python>python test.py
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    FuncA(5,0)
  File "test.py", line 2, in FuncA
    assert(b != 0)
AssertionError
```

```
C:\Code\Python>python -O test.py
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    FuncA(5,0)
  File "test.py", line 3, in FuncA
    print(a/b)
ZeroDivisionError: division by zero
```

ERROR HANDLING

Functions in Python can return multiple values. For error handling, you can use this feature to return both a result and error code.

You can then check the return values to determine if a function failed.

```
def FuncA(a, b):
    result=0
    err=True
    try:
        #assert(b != 0)
        result=a/b
    except ZeroDivisionError:
        err=False
    return result, err


answer, ok=FuncA(10, 5)
if(ok):
    print(answer)
```

EXCEPTIONS

How many lines of code can the average developer write without a syntactical or logical errors? If your answer is “zero”, skip to the next module.

For that reason, exception handling is a necessary ingredient for most applications. It allows your application to recover from abnormal events and arguably makes your program more robust.

Even if you write perfect code, an exception remains possible. Many factors can contribute to an exception – some of them outside the control of the developer.

- Hardware error
- Latency
- User input

TRY / EXCEPT

The try clause creates a guarded block. This block is guarded from exceptions. If an exception is raised in a guarded block, control is transferred to the except clause. If the except clause catches that type of exception, execution continues in the except block. The code after the exception in the guarded block will be orphaned.

```
try:
    var1=5
    var2=0
    var1/=var2
# orphaned code
except ZeroDivisionError as e:
    print(e)
# execution continues here
```

MULTIPLE EXCEPTIONS

- You can catch multiple exceptions using tuples.

```
try:
  FuncC()
except (ZeroDivisionError, IndexError) as e:
  print("Error ", e)
```

- More than one except block is allowed. The more specific exceptions should be first.

```
try:
  FuncC()
except (ZeroDivisionError, IndexError) as e:
  print("Error ", e)
except ArithmeticError as e:
  print("Error ", e)
```

GLOBAL EXCEPT

Avoid global exceptions handlers.

```
try:
  # guarded block
except Exception:
  pass
```

Global exception handlers are treacherous! Nonetheless, they are common.

A global exception handler catches all exceptions – not a specific exception. How do you handle a non-specific exception? Global exception handlers are most often used to suppress exceptions and force the application to continue.

The hidden exception will lurk in the background and potentially crash the application later. It will be more difficult to debug at that time.

FINALLY

An exception may orphan code. That is a problem if the orphaned code must be executed, such a cleanup code. Code that must be executed should be placed in a *finally* block. This code is executed regardless of whether an exception is raised.

```
def FuncA():
  file=open(r"c:\code\data.txt")  try:
    pass # do something
  except Exception:
    pass
  finally:
    file.close()
```

ELSE

How did execution reach the statement after the *except*? Did you fall through without an exception or from the except clause? The *else* clause helps when there is different behavior based on either scenario.

```
try:
    pass
except:
    pass
else:
    print("No exception")
finally:
    pass
```

WRAP MAIN

Bootstrapping main within a try / except creates a global exception handler.

You cannot recover or continue the program from the except clause. It is solely an opportunity to perform an orderly exit or update the user if appropriate.

```
def main():
    a=4
    b=0
    a=a/b


if __name__ == '__main__':
    try:
        main()
    except:
        print("Unhandled exception.")
        print("Good bye and thanks for all of the fish")
```

GLOBAL EXCEPTION HANDLER

Global exception handling is helpful in handling unhandled exceptions.

- Wrap main within a try / except
- Change sys.excepthook to a user defined method

This is a better approach than attempting to place exception handling along every code branch, which is overly complexed and prone to error. It is a good practice only to place exception handling where exceptions are *likely* to occur.

CHANGE SYS.EXCEPTHOOK

When an exception is unhandled, the interpreter calls the sys.excepthook function with three parameters:

- Exception type
- Exception instance
- Traceback instance

A default implementation is assigned to sys.excepthook, which notifies the user of the global unhandled exception. You can update sys.excepthook to reference a different function – even a user defined method.

CHANGE SYS.EXCEPTHOOK – EXAMPLE CODE

```
import sys

def unhandled_exception(exception_type, exception_object, \

        exception_traceback):

     print("Uncaught exception")

     print("==================\n")

     print(exception_type.__name__)

     print(exception_object.args)

     print("\n++++++++++++++++++\n")

     print(exception_traceback.print_stack())


sys.excepthook = unhandled_exception


if __name__ == "__main__":

    raise RuntimeError("Test unhandled")
```

Exceptions

Lab 5

LAB: CREATE AN EXCEPTION FILE

This lab:

- Write exception records to a file.

EXCEPTION FILE

1. Define a function to handle unhandled exceptions
2. In function:
    1. Open file for extension

        open("filename","w+")
    2. Write this record into the file

        mm/dd/yy hh:mm:ss exceptionname
    3. Close file
3. Update sys.excepthook to reference the new function
4. Raise an unhandled exception in main
5. Run the application a few times.
6. Review the results in the created / appended file

Testing

AUTOMATED VS MANUAL TESTING

Exploratory testing

- Checking features and experimenting with them without a plan
- Exploratory testing is a form of manual testing
- A complete set of manual tests consists of the majority features in an application

Automated testing executes a test plan using a script

Python provides libraries for automated testing

UNIT VS INTEGRATION TESTS

Testing a single component or function is called unit testing

Testing multiple components is called integration testing

It’s hard to diagnose the cause of problems that arise during integration testing without isolating the failing component in a system.

EXAMPLE: SUM() UNIT TEST

```
# UNIT TEST - sum_test

assert sum([1,2,4]) == 7
```

✓ 0.6s

The example code on the right shows how you can use an assertion to test if the result of the sum() function is correct by comparing it to a known output.

```
# UNIT TEST - sum_test

assert sum([1,2,4]) == 5
```

⊗ 0.4s

```
---------------------------------------
AssertionError
c:\Users\ishea\Documents\Python3Lessons\
e: 3>()
      1 # UNIT TEST - sum_test
----> 3 assert sum([1,2,4]) == 5

AssertionError:
```

EXAMPLE: UNIT TEST SCRIPT

You can write a test case as a function inside a script and define an entry point as shown in the example code.

This approach works well for simple checks, but for complex tests, you must use test runners.

```
# UNIT TEST - sum_test

def test_sum():
    assert sum([1,4,5]) == 10

if __name__ == '__main__':
    test_sum()
    print("All tests passed")
```

TEST RUNNERS

There are several python test runners that you can use, but the most common ones are:

- unittest (built-in)
- pytest
- nose or nose2

UNITTEST

The unittest unit testing framework supports a number of testing operations. With unittest, you can:

- Automate tests
- Share setup and shutdown code for tests
- Aggregate test into collections

It supports testing concepts using an object-oriented approach.

UNITTEST CLASSES

Test fixture: Preparation needed to perform one or more tests.

Test case: An individual unit of testing. Checks the output of a given test specific to a set of inputs. The TestCase class is used to create new test cases.

Test suite: A collection of test cases, test suites or both. Used for test aggregation.

Test runner: Used to orchestrate the execution of tests and provides the outcome to the user.

CREATING A TEST WITH UNITTEST

- Tests must be inside a child class of TestCase as methods
- The unittest.TestCase class provides a series of special assertion methods that you can use to perform a test instead of the built-in assert statement.

EXAMPLE: UNIT TEST SCRIPT

The example creates a TestSum test case by inheriting unittest.TestCase, the test methods must be prefixed with the letters test.

If you are using Python 2,7 and below, you must import unittest2 instead of unittest.

```
import unittest

class TestSum(unittest.TestCase):

    def test_sum(self):
        self.assertEqual(sum(1,4,5),10,'1 + 4 + 5 must be 10')


if __name__ == '__main__':
    unittest.main()
```

EXAMPLE: UNIT TEST SCRIPT

unittest.main() provides a command line interface to the test script.

```
PS C:\> py $LessonHome\example1.py
.
----------------------------------
Ran 1 test in 0.000s

OK
```

ASSERTIONS

Method	Equivalent to
assertEqual(a,b)	a == b
assertTrue(a)	bool(a) is True
assertFalse(a)	bool(a) is False
assertIs(a,b)	a is b
assertIsNone(a)	a is None
assertIn(a,b)	a in b
assertIsInstance(a,b)	isinstance(a,b)

COMMAND LINE INTERFACE

The unittest module can be used from the command line to run tests from modules, classes or even individual test methods

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

POP QUIZ: UNITTEST

Explain the output of this test.

```
from fractions import Fraction

import unittest


class TestSum(unittest.TestCase):
    def test_list_int(self):
        """
        Test that it can sum a list of integers
        """
        data = [1, 2, 3]
        result = sum(data)
        self.assertEqual(result, 6)

    def test_list_fraction(self):
        """
        Test that it can sum a list of fractions
        """
        data = [Fraction(1, 4), Fraction(1, 4),
Fraction(2, 5)]
        result = sum(data)
        self.assertEqual(result, 1)

if __name__ == '__main__':
    unittest.main()
```

5 MINUTES

PYTEST

Pytest is another python test runner which provides a more convenient approach to testing compared to unittest.

```
pip install pytest
```

PYTEST FEATURES

- Supports built-in assert statement
- Support filtering test cases
- Rerun from the last failing test
- Plugins available to extend testing functionality
- No need for classes to create test cases
- Test cases are a series of functions in a .py file

PYTEST EXAMPLE

Pytest test cases are a series of functions prefixed by the letters *test* inside a .py file.

```
1   def func(x):
2       return x + x**2
3
4   def test_func():
5       assert func(5) == 30
```

PYTEST EXAMPLE

You run the tests by passing the test script as an argument to *pytest* in the command-line.

```
PS C:\> pytest $LessonHome\pytestexample.py
============================================================================= test session starts =====
platform win32 -- Python 3.8.8, pytest-6.2.3, py-1.10.0, pluggy-0.13.1
rootdir: C:\
plugins: anyio-2.2.0
collected 1 item

Users\ishea\Documents\Python3Lessons\Intro_To_Python\intermediate\testing\pytestexample.py .

============================================================================== 1 passed in 0.03s ======
```

LAB: CREATE UNITTESTS FOR A FLASK API

This lab:

- Create a flask API and test its operations using unittest.

THE FLASK MODULE

Flask is a web application framework that makes it simple to build Web APIs in python.

It provides classes that you can use to create an API and define HTTP operations. To use flask you need to install it first using the following command:

pip install flask

A minimal flask application is as shown below:

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"
```

THE FLASK API

1. Given the following API code, add operations for addition, subtraction, division, modulus, square and cube and return the correct result.

```
from flask import Flask

app = Flask(__name__)

operations = {
    '+':lambda a,b:a+b
}

@app.route('/<a>/<b>/<operator>')
def action(a,b,operator):
    result = None
    # Call correct operation and update result
    return str(result)
```

THE UNIT TEST

1. Expand the TestCase class in the code below and include unit tests for each arithmetic operation exposed in the action operation of the API.

```
import test_api
import unittest

class APITestCase(unittest.TestCase):

    def setUp(self):
        test_api.app.testing = True
        self.app = test_api.app.test_client()
```

You can get the result by calling self.app.get('/1/2/+') for example to get the sum of 1 and 2.

Use assertions to test the operations.

APPENDIX

DIRECTORY COMMAND

CURRENT DIRECTORY

Managing the current directory is sometimes the difference in a script / automation working or failing. Call the methods getcwd and chdir to the get and set the working directory respectively.

Both getcwd and chdir functions are found in the os module.

```
>> import os
>> os.getcwd()
c:\\code'
>> os.chdir(r"c:\\code2")
>> os.getcwd()
c:\\code2'
```

COMMON PATH COMMANDS

The os.path module has functions to manage path information. The exception type is OSError. Here are the common path methods:

- path.exists
- path.dirname
- path.basename
- path.split
- path.getsize

COMMON PATH – EXAMPLE CODE

```
from os import path
filename=r"c:\code\test.txt"
dirname=r"c:\windows"
print("Exists =  ", path.exists(filename))
print("Dirname  =  ", path.dirname(filename))
print("Basename  =  ", path.basename(filename))
print("Split  =  ", path.split(filename))
print("GetSize  =  ", path.getsize(filename), " bytes")
print("GetSize  = ", path.getsize(dirname), " files")
```

```
Exists = True
Dirname  = c:\code
Basename  = test.txt
Split  = ('c:\\code', 'test.txt')
GetSize = 3 bytes
GetSize  = 24576 files
```

POP QUIZ: DOES THE PREVIOUS CODE WORKS

What does this code do?

```
print("GetSize  = ", \
        path.getsize(dirname), " files")
```

10 MINUTES

GET DIRECTORY SIZE

This code returns the number of files in a directory and the number of bytes.

```
import os


files=len(os.listdir(r"c:\code"))

print("Files = ", files)

total=sum([os.path.getsize(f) \
           for f in os.listdir(r"c:\code") \
           if os.path.isfile(f)])

print("Bytes = ",total)
```

COMMON DIRECTORY COMMANDS

The os.path module has functions to manage directory information. The exception type is OSError. Here are common directory methods:

- os.listdir: returns the names of items in the directory
- os.path.join: combine path and entry name
- os.stat: get statistics on file
- os.mkdir: create a directory
- os.makedirs: create directory tree

OS MODULE – EXAMPLE CODE

```
import os


filename="test.txt"

extfilename=r"c:\code\test.txt"

dirname=r"c:\code"

dirname2=r"c:\code\yourname"

dirname3=r"c:\code\one\two\three"

print("Listdir  = ", os.listdir(dirname))

print("Join  = ", os.path.join(dirname, filename))

print("Stat  = ", os.stat(dirname))

print("Stat= ", os.stat(extfilename))

#print("Mkdir  = ", os.mkdir(dirname2))

print("Makedirs = ", os.makedirs(dirname3))
```

ITERATING DIRECTORIES

Use os.walk to walk (iterate) the contents (subdirectories and files) in a directory.

```
import os

currentdir=r"c:\code"
os.chdir(currentdir)

for root, dirs, files in os.walk("."):
    for file in files:
        print(os.path.join(root, file))
    for dir in dirs:
        print(os.path.join(root, dir))
```

PICKLE

Pickle provides a binary protocol to serialize and deserialize Python objects.

- Pickling is converting an object into bytes for serialization
- Unpickling is converting bytes into an object

Pickling has advantages over marshaling or JSON:

- Does not reserialize references
- Serialize user defined types
- Efficient format
- Pickle tools

There are some disadvantages:

- Specific to Python
- Not clear text

PICKLE – SAMPLE CODE

define company class with members (1)

open pickle file for write binary (2)

Dump 1st record (3)

Dump 2nd record (4)

Release memory for company objects (5)

```
import pickle

class Company(object):                                  1
    def __init__(self, name, value):
        self.name = name
        self.value = value

with open('company_data.pkl', 'wb') as output:                  2
    company1 = Company('banana', 40)
    pickle.dump(company1, output)                       3

    company2 = Company('spam', 42)
    pickle.dump(company2, output)                       4

del company1                                            5
del company2
```

PICKLE – SAMPLE CODE - 2

open file read binary (1)

load first record from file (2)

load second record (3)

```
with open('company_data.pkl', 'rb') as input:                    1
    company1 = pickle.load(input)
    print(company1.name)  # -> banana                           2
    print(company1.value)  # -> 40

    company2 = pickle.load(input)                                3
    print(company2.name) # -> spam
    print(company2.value)  # -> 42
```

SHELF

Shelf is a dictionary persistent object. It has the dictionary interface but backed by a file store. It is located in the shelve module.

Shelf is convenient because it uses a standard dictionary interface. It combines the dictionary model with implicit data persistence. There are three simple steps to use a shelf.

1. Call shelve.open function to associate the object with a file. It returns a shelf object. Creates a .bak file.
2. Leverage shelf as a dictionary.
3. Close shelf when done – shelf.close.

Writing or reading from the shelf is the same steps.

SHELF – EXAMPLE CODE

```
# Write to shelf

import shelve

cast=shelve.open("movie")

cast["bob"]="Robert Culp"
cast["ted"]="Elliot Gould"
cast["carol"]="Natalie Wood"
cast["alice"]="Dyan Cannon"

cast.close()

#exception because shelf closed
#cast["alice"]="supported"
```

```
# Read from shelf

import shelve

cast=shelve.open("movie")
print(dict(cast))
```

JSON

JSON is a popular format for data transmission and data storage. JSON files can range from simple to complex. The adjacent JSON file is simple. Simple JSON files are represented by a dictionary in memory, which has name / value pairs.

You can iterate the items in the json file similar to dictionary entries.

```
{
  "theme" : "bluespring",
  "size": "small",
  "splashscreen": "false"
}
```

JSON – SIMPLE EXAMPLE

Open the JSON as a standard file using the open function from the OS module. Load data from the file as JSON using json.load(*file object*). You can then access the JSON as a dictionary, including iterating the JSON file.

```
import json


json_data=open(r".\my.json").read()

data = json.loads(json_data)

for item in data:

    print(data[item])
```

JSON – COMPLEX EXAMPLE

For a complex JSON file, the data is a hierarchal dictionary where nested items are sequences. Sometimes multiple levels of sequences. The adjacent JSON file is complex with multiple layers. For example, the maps element contains a sequence of two items (i.e., ids). Therefore a complex sequence is required to mirror an equally complex JSON file.

```
{
 "maps":[
         {"id":"blabla","iscategorical":"0"},
         {"id":"blabla","iscategorical":"0"}
        ],
"masks":
         {"id":"valore"},
"om_points":"value",
"parameters": [
         {"id":"valore1"},
         {"id": [
               "valore2",
               "valore3"
         ]}
    ]
}
```

POP QUIZ: COMPLEX EXAMPLE

Based on the previous examples, what will this code display:

```
import json

json_data=open(r".\file.json").read()

data = json.loads(json_data)

print(data["om_points"])
print(data["maps"][1]["id"][1])
print(data["parameters"][1]["id"][1])
```

10 MINUTES

Zen of Python

Import *this*

THIS

In this module, you will learn the Zen of Python.

Have you ever tried *this* in Python: import this?

PYTHON PROGRAMMER

Are you a Python programmer, a true practitioner, or just a Python visitor?

IMPORT THIS

Most software languages have a culture. What is the philosophy of Python? This is documented as the Zen of Python. The *import this* command provides the best practices of a true Python developer.

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Write readable code

Don't write implicit code

How complex should it be?

Simple < complex < complicated

Reduce level of arc, branches, and paths

IMPORT THIS - 2

Sparse is better than dense

Readability counts

Special cases aren't special enough to break the rules

Although practicality beats purity

Errors should never pass silently. Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

Whitespace is a great thing!

Self documenting code

Refactoring

Unless effort exceeds rewards

No global exception handlers. When necessary, explicitly silent is acceptable.

Avoid import * and *args for example

IMPORT THIS - 3

There should be one -- and preferably only one – obvious way to do it.	Defer to Guido
Although that way may not be obvious at first unless you're Dutch.	Defer to Guido
Now is better than never.	Fix problems as they occur
If the implementation is hard to explain, it's a bad idea.	Value transparency and clarity

IMPORT THIS - 4

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those

Value simplicity over opaqueness

Use namespaces

BEAUTIFUL IS BETTER THAN UGLY

- Whitespace is good
- Short lines
- Separate statements on different lines
- Separate imports on different lines
- Low complexity

EXPLICIT IS BETTER THAN IMPLICIT

This is *bad*:

```
for mod1 import *
```

This is *good*:

```
import mod1
```

This is *bad*:

```
Func(*arg)
```

This is *good*:

```
Func(a, b)
```

SIMPLE IS BETTER THAN COMPLEX

- Flat storage versus hierarchal
- Pickle for serialization versus marshaling
- JSON versus binary

SPARSE IS BETTER THAN DENSE

- Whitespace
- Whitespace
- Whitespace

READABILITY

- Self documenting code
- Short functions – 5 lines code or less
- Short classes – screen worth info (see the class at glance)
- Reasonable symbolic names
  - Functions - verbs
  - Classes - nouns

PRACTICALITY BEATS PURITY

Try not to break the rules; but nonetheless be practical.

```
def main():
    a=4
    b=0
    a=a/b


if __name__ == '__main__':
    try:
        main()
    except:
        print("Unhandled exception.")
        print("Good bye and thanks for all of the fish")
```

NAMESPACES ARE ONE HONKING GREAT IDEA

- Import a module
- Create additional namespaces using classes

ERRORS SHOULD NEVER PASS SILENTLY

```
try:
    pass
except Exception:
    pass
```

IN THE FACE OF AMBIGUITY, REFUSE THE TEMPTATION TO GUESS

```
from a import *
```

ONE OBVIOUS WAY TO DO IT

Research Guido van Rossum - the BDFL.

Guido recently resigned but pledges to remain a core programmer for Python.

https://bit.ly/2meaVb3

PYTHON IDIOMS

There are a variety of idioms from various sources:

1. Review PEP 8: Style Guide for Python Code (http://bit.ly/1ARqSBt)
2. Follow best practices on whitespace: 4 spaces for indentation, never use tabs, one blank line between functions, and two blank lines between classes.
3. Keep lines under 80 characters. Prefer implicit returns over back slashes.
4. Don't use compound statements
5. Swap with this technique: a, b=b, c
6. In the Python interpreter expression results are assigned to the underscore (_) temporary variable

PYTHON IDIOMS - 2

7. Build strings from lists of strings and *join* command.
8. Use “in” when possible
9. Check for empty list not len()=0.
10. Use list comprehensions
11. Use generators
12. Use advanced string formatting
13. Custom sorting is easy
14. EAFP versus LBYL
15. Don’t import with wildcards

CUSTOM SORTING

Here is an example of custom sorting. Sorting the tuple by the second field.

```
test=[("Bob",5), ("Ted",7), ("Carol",3)]

stest=sorted(test, key=lambda item:item[1])

print(stest)
```

ADVANCED STRING FORMATTING

Here is an example of advanced string formatting.

More soon.

```
a,b,c=5,10,15

s1="The story of {0}, {1}, and {2}".format(a, b, c)

s2="My name is {0:8}".format('Fred')


print(s1)

print(s2)


print(format(10.0, "7.3"))


a=5

b=20


print("{0:{1}}".format(a, b))
```

EAFP VERSUS LBYL

```
#EAFP

# Easier to ask for

# forgiveness than permission

try:

    x = my_dict["key"]

except KeyError:

    # handle missing key


#LBYL:

# Look before you leap

if "key" in my_dict:

    x = my_dict["key"]

else:

    # handle missing key
```

IDIOMS EVERYWHERE

There are several sources for Python idioms. Actually, there is no shortage of this online. The Top 10 Python Idioms I Wish I'd Learned Earlier (http://bit.ly/2ofzMO7):

POP QUIZ: BEST PRACTICES

What are some of your personal best practices for writing quality code?

10 MINUTES

COMPREHENSIONS

Comprehensions are a best practice in Python.

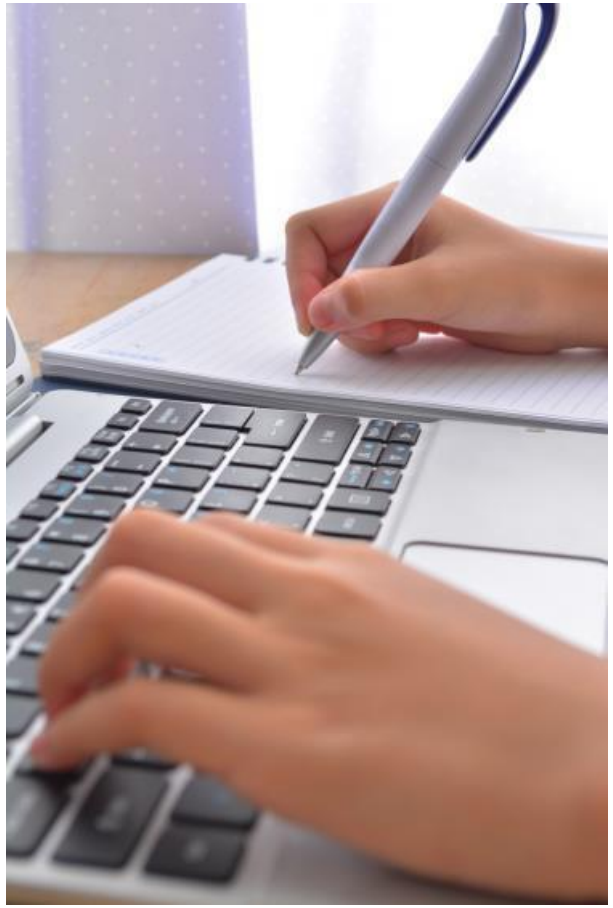
The for..in syntax is sufficient for basic iteration, which is suitable for most situations. For complex scenarios, you might consider using comprehensions.

Comprehensions perform a transformation on a collection. The result is a new sequence. A new collection is created by applying an expression to each item in a sequence.

The most common comprehensions are list and dictionaries. The type of iteration is typically identified by the enclosing characters. For example, if the enclosing characters are "[ ]", expect a list comprehension.

HANDS-ON EXERCISE

Comprehensions



COMPREHENSIONS – HANDS ON

Enter the following into the Python Command-Line Interpreter.

```
>>> mylist=[1,2,3,4,5]
>>> [item+1 for item in mylist]
[2, 3, 4, 5, 6]
```

First, you create a list. The next line is a List comprehension. Let us examine the statement from right to left.

- "item in mylist" Consider this an abbreviated for loop. At each iteration, an element from mylist is placed in item.
- "Item+1" This is the action to be applied (comprehended) on each iterated item. In our example, 1 is added to each item.
- The result of the action (and comprehension) is a new list. The original list is not changed.

Here is the syntax:

[action for target in source]

COMPREHENSIONS – HANDS ON - 2

In the same interpreter session enter.

```
>>> mylist
[1, 2, 3, 4, 5]
>>> [item+1 for item in mylist if item%2==0]
[3, 5]
```

Notice that the original list is unchanged.

For a comprehension, the *if* condition acts a filter on the iteration. When false, that particular iteration is skipped. In our example, the odd values of mylist are omitted. Therefore 1 is added to only the two even numbered items of mylist (i.e., making them odd numbers). The result is returned.

COMPREHENSIONS – HANDS ON - 3

Start an interpreter session and enter:

```
>>> employees={"Donis":"e", "Bob":"s"}
>>> employeetype={"e":"Exempt", "s":"Salary"}
>>> {empl[0]:employeetype[empl[1]] for empl in employees.items()}
{'Donis': 'Exempt', 'Bob': 'Salary'}
```

This is an example of Dictionary comprehension.

You create two dictionaries. The first is a list of employees with an wage code, such as “e” for Exempt employee. The second is a mapping of the wage code to a description.

COMPREHENSIONS – HANDS ON - 4

Dictionary comprehension create a new dictionary item

This explains the sample code.

- employee.items returns a tuple (key, value) for each element of the dictionary
- empl[0]:employeetype[empl[1]] looks complicated but is relatively straightforward. Creates a new dictionary mapping using the key of the initial dictionary (empl[0]) and mapping the code (empl[1]) to the description in the employee type dictionary.

Hand-on exercise
Done!

GENERATORS

Generators are super cool and take comprehensions to the next logical level.

Comprehensions return the results immediately. This is potentially a lot of data that which can consume a significant portion of memory. Alternatively, a generator is a runtime engine that returns each item of the result as requested. Items included in the result are delivered not at once but individually with the next function.

Generators has a similar syntax as a comprehension but bounded with parentheses. The result is a generator object. Call the next function on the generator to get the next item – starting with the first item of the collection.

```
>>> mylist=[1,2,3,4]
>>> mygenerator=(i*i for i in mylist)
>>> next(mygenerator)
1
>>> next(mygenerator)
4
>>>
```

ENUMERATE GENERATOR

create a generator (1)

enumerate generator (2)

when fully enumerated, StopIteration exception raised. (3)

Recreate generator (4)

Enumerate using for loop (5)

```
numbers=list(range(1,100))

mygen=(item+1 for item in numbers)   1
while(True):
    try:
        a=next(mygen)   2
        print(a)
    except StopIteration as error:   3
        break

numbers=list(range(1,100))

print("\n\n\n")
mygen=(item+1 for item in numbers)   4

for x in mygen:   5
    print(x)
```

ADVANCED FORMATTING

In the next lab, string formatting would be helpful. String formatting in Python is similar to C and printf. String formatting relies on % (insertion value) and format specifier, which is used as a placeholder. Here is an example of string formatting with a single placeholder

```
>>>
>>>
>>> print("\nThe answer to life is %d \n\t- Douglas Adams"%42)

The answer to life is 42
        - Douglas Adams
>>>
>>>
```

The “%d” format specifier is a placeholder for decimal numbers. After the closing quote, “%42” is the insertion for the placeholder.

FORMAT SPECIFIERS

Here are the more common specifiers:

d. Decimal format

e. Exponentiation

f. Fixed point

g. General

s. String

x. Hexadecimal

```
>>>
>>> english=True
>>> print("Donis %s"%('Hello' if english else 'Ola'))
Donis Hello
>>>
```

FORMAT SPECIFIERS - PARAMETERS

For more than one format specifier, insertions are given as a tuple. The insertion can be a calculation.

```
>>>
>>>
>>> a=1
>>> b=4
>>> print("%d + %d = %d"%(a,b,(a+b)))
1 + 4 = 5
>>>
>>>
```

Zen of Python

Lab 4

MATRIX OPERATIONS

Create an array of lambdas representing math operations that return integral types. Each takes two parameters and returns the result.

	operation	param 1	param2	answer
Addition	a	4	5	?
Division	d	0	3	?
Multiplication	m	4	8	?
Subtraction	s	1	4	?

Create a two-dimensional “something” to input the operation type, parameters, and later store the result.

Iterate each row. Calculate “operation” with parameters and store results back on the same row.

Iterate the array and display the operations, parameters, and the result. Display the result like above.