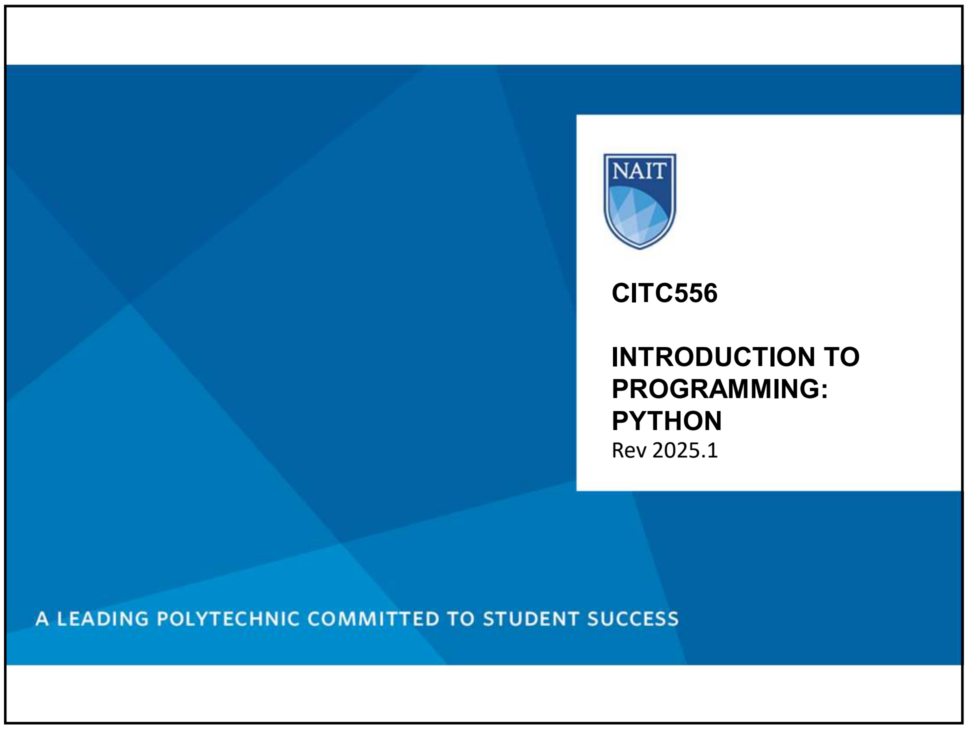
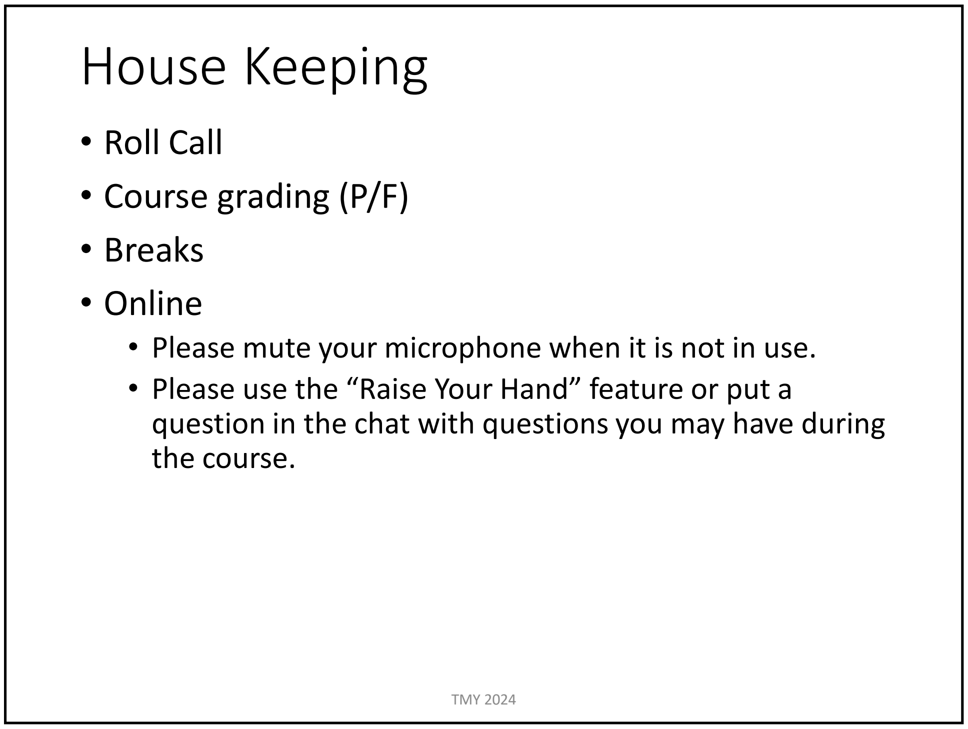


CITC556

INTRODUCTION TO PROGRAMMING: PYTHON

Rev 2025.1

A LEADING POLYTECHNIC COMMITTED TO STUDENT SUCCESS



House Keeping

- Roll Call
- Course grading (P/F)
- Breaks
- Online
  - Please mute your microphone when it is not in use.
  - Please use the “Raise Your Hand” feature or put a question in the chat with questions you may have during the course.

TMY 2024

Outline

- Introduction to Programming
- Introduction to Python
- Python Shell, IDLE and other Integrated development environments
- Variables And Data Types
- Data Structures (Lists, Tuples, Dictionaries, Sets)
- Controlling Program Flow (Conditionals)
- Iteration
- Functions
- Objects

Scope

- Create variables
    - Accept inputs and store values
- Output data to screen and files
- Conditional statements
    - Conditionally do one thing OR another based upon the evaluation of a condition
- Loops
    - Do Something multiple times
- Mathematical conditions
- Change data types
    - Alter datatypes
- Introduction to custom user objects

Learning Hurdles in Software Development: Your Challenges

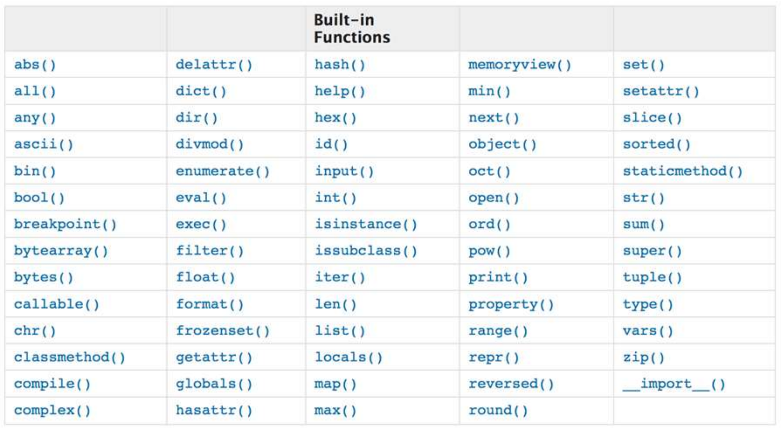
- Logic
- Syntax
- Creativity

Introduction to Python

- Installation software is available from https://www.python.org/
- Python can run on many different computer operating systems:
  - Windows
  - Apple MacOSX
  - Raspberry Pi
  - Linux
  - Unix
  - Etc…

Introduction to Python

- The base python installation has built-in functions:



		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Python Standard Libraries

- A library is a collection of functions and methods that allow you to perform lots of actions without writing your own code.
- You gain access to libraries by importing them into your code using the “import” keyword.

  example:

  import random

  The random.py library gives you the ability to generate random numbers.

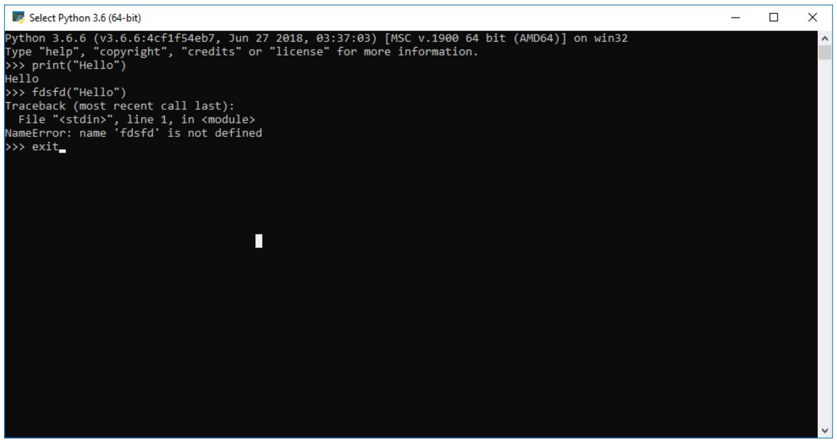
- https://docs.python.org/3/library/

Python Shell, IDLE and IDEs

- **Python Shell** is a the simplest way to use the Python language using an interactive shell.
  - Typically used as a scratch pad to enter commands.
  - Good for testing single commands
  - Commands can be entered in the shell and evaluated.
  - Does not allow for editing of Python commands as there is no way to edit content after it has been run.
    - The command(s) need to be entered again.

TMY 2024

Python Shell Example



```
Python 3.6.6 (v3.6.6:4cf1f54eb7, Jun 27 2018, 03:37:03) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello")
Hello
>>> fdsfd("Hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'fdsfd' is not defined
>>> exit
```

TMY 2024

Python IDLE

- Python IDLE is a graphical Python shell that contains a text editor and features code syntax coloring and code completion.
- It offers all the functionality of the Python Shell plus enhancements.

Microsoft Visual Studio

Visual Studio (full version)

is a "full-featured" and "convenient" development environment.

Visual Studio (free "Express" versions - only until 2017)

are feature-centered and simplified versions of the full version. Feature-centered meaning that there are different versions (Visual Studio Web Developer, Visual Studio C#, etc.) depending on your goal.

Visual Studio (free Community edition - since 2015)

is a simplified version of the full version and replaces the separated express editions used before 2015.

Visual Studio Code (VSCode)

is a cross-platform (Linux, Mac OS, Windows) editor that can be extended with plugins to your needs.

https://visualstudio.microsoft.com/vs/compare/

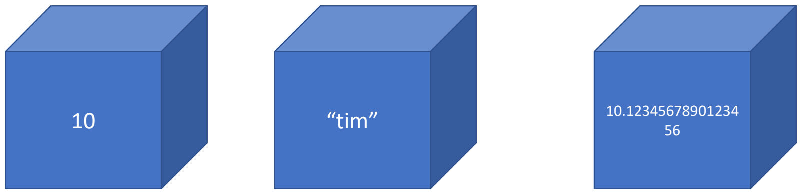
Python IDEs

- The Python languages has numerous Integrated Development Environments (IDEs), many that are free.
- A list of the editors is maintained at:
  - https://wiki.python.org/moin/PythonEditors

Variables and Data Types

Variables

- Variables are ”**containers**” to hold values.



int_one = 10
(integer – whole numbers, no decimals)

first_name = ‘tim’
Or
first_name = “tim”
(string)

float_two = 10.1234567890123456
(float decimal)

Data Types are the implicitly automatically assigned by python

TMY 2024

Variables: Simple Types

- Integer: Any whole number
- Float: Any number with a decimal place
- String: Any sequence of characters of variable length
- Boolean: A True or False value
- None: No value

- To determine the data type of a variable, Python provides a handy function called type:

```
>>> type(“Hello”)
<class ‘str’>
```

TMY 2024

Variables

- In Python, different data types are used to classify one particular type of data, determining the values that you can assign to the type and the operations you can perform on it.
- When programming, there are times we need to convert values between types in order to manipulate values in a different way.
- For example, we may need to concatenate numeric values with strings, or represent decimal places in numbers that were initialized as integer values.

TMY 2024

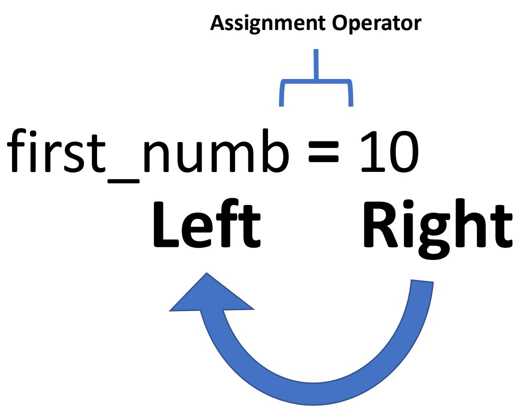
Variables and Data Types: Declaration and initialization

$$\underbrace{\text{first\_numb}}_{\text{Variable Name}} \overbrace{=}^{\substack{\text{Assignment Operator} \\ \text{Single = sign}}} \underbrace{10}_{\text{Value}}$$

Left Right

TMY 2024

Variables Assignment (=)



Variables and Data Types

- How does python store variables
- Python stores a variable value by using a concept called “by reference”.
- To speed up the performance of python the actual variable is stored in a memory location and we create a “pointer” to that memory location.
- This is much faster than referring to a variable “by value”
  - When a programming language uses “by value” to store and obtain the value of variables, a copy of the value is made, which is slower than using a pointer.

Variable and Data Types: Naming Rules

- Variables names must start with a letter or an underscore (single underscore for user created), such as:
  - _first_name
  - first_
- The remainder of your variable name may consist of letters, numbers and underscores.
  - password1
  - n00b
  - un_der_scores
- Names are case sensitive.
  - case_sensitive, CASE_SENSITIVE, and Case_Sensitive are each a different variable

TMY 2024

Variables and Data Types: Naming Conventions

- Readability is very important. Which of the following is easiest to read? I'm hoping you'll say the first example.
  - **first_name**
  - firstname
  - First_Name
- Descriptive names are very useful. Which do you think is the better variable name?
  - subtot
  - monthly_subtotal
- Avoid using the lowercase letter 'l', uppercase 'O', and uppercase 'I'.
  - Why?
  - Because the l and the I look a lot like each other and the number 1. And O looks a lot like 0.

TMY 2024

Variables and Data Types: Declaring

- //C# or Java
  - int testScore = 42;
  - string firstName = “Tim”;

The data type needs to be declared in many languages (explicit variable declaration)

- //Python
  - test_score = 42
  - first_name = ”Tim”

Python automatically creates the variable when the variable is assigned.
(implicit variable declaration)

The variable type is assigned when a value to a variable is set.

TMY 2024

Variables and Data Types: Integers and Floats

- Integers are whole numbers
  - 10
  - 234345
  - 54
- Floats are numbers with a decimal point
  - 10.0
  - 23445.1234
  - 54.1

TMY 2024

Variables and Data Types: Strings

- Text data is represented by the String data type
- Strings are defined by using either single, double or three double quotes:
  - “Hello” == ‘Hello’ == “””Hello”””

Variables: Numbers and Strings

- When performing a operation such as addition on strings, the result is called concatenation.

  - my_string_one = “10”
  - my_string_two = “20”

Both of the variables are strings, not numbers.

Code:

```
my_result = my_string_one + my_string_two
//the result for using the addition operator on two strings is concatenation
print(my_result)
```

Result:

```
1020
```

Variables: Numbers and Strings

- When performing a operation such as addition on numbers, the result is the addition of the numbers.
  - my_number_one = 15
  - my_number_two = 25

Both of the variables are integers

Code:

```
my_result = my_number_one + my_number_two
//the result for using the plus operator for two numbers is addition
//of the two numbers
print(my_result)
```

```
Result:
40
```

Variables: Numbers and Strings

- Consider the following
  - my_string_one = “10”
  - my_string_two = “20”
  - my_number_one = 15

String data type

Number data type

Code:

```
my_result = my_string_one + my_number_one
print(my_result)
```

Result:

```
 mystring_number_plus_string = my_string_one + my_number_one
TypeError: must be str, not int
```

Python is telling you that you have tried to perform an operation that it cannot complete.

Variables: Numbers and Strings

- So:
  - string + string – Concatenation
  - number + number – Addition
  - string + number – BAD, Does not work

Note: It is up to the programmer to check the values prior to performing operations to check the data type and perhaps the value too

Variables: Numbers and Strings

- Suppose your application was retrieving data from the web and it received a string value of 90, which represented a students test score.
- As noted in the earlier slides, you cannot perform mathematical operations on string/string or string/number combinations.
- If the programmer wanted to perform math and represent a string as number what could you do?
- Python and many other languages allows for a concept called casting.

Variables: Casting

Function	Description
int(x)	Converts x to an integer
long(x)	Converts x to a long integer
float(x)	Converts x to a floating point number
str(x)	Converts x to an string. x can be of the type float. integer or long.
hex(x)	Converts x integer to a hexadecimal string
chr(x)	Converts x integer to a character
ord(x)	Converts character x to an integer

Variables: Casting

- Casting allows the programmer to represent a data type as another data type without changing the value and the original variable type:
  - Syntax:
    - Result = casting_data_type(original Variable)

  - str_number_var =“20”
  - $\text{my\_result} = \underbrace{\text{int}(\underbrace{\text{string\_number\_var}}_{\text{string}})}_{\text{int}} + 100$

  - Result: 120

Variables and Data Types: Boolean

- Boolean values are used in programming for decision making.
- They are commonly used in all forms of programming.
- They have either a **True** or **False** value
- For instance, suppose you are required to evaluate test scores and need to assign a pass or fail based on the score. For our example we can assign a passing grade of 60.0.
  - We would ask the question ”if the students grade was 60 or higher than they pass the course”
    - The question would evaluate to True or False based on the grade.

Variables and Data Types: None

- The variable type none is variable that has been declared but does not have a value yet.
- Note: If you evaluate its value it will be “False”

- Students_With_No_Score = None

Input & Output

input()

The input() function allows user input and assignment to a variable:

Syntax: input(prompt)

Example: x = input('Enter your value:')

note: What type of data type is returned?

print()

The print() function outputs specified message to the screen, or other standard output device.

Syntax: print()

Example: print("Hello, how are you?")

DEMO Lab 1

Create a program to ask the user their name. Store the user name in a variable and output a greeting with the users name.

Arithmetic Operators

Operator	Description	Example a = 10, b = 20
Addition +	Adds values on either side of the operator	a + b = 30
Subtraction -	Subtracts right hand operand from left hand operand	a – b = -10
Multiplication *	Multiplies values on either side of the operator	a * b = 200
Division /	Divides left hand operand by right hand operand	b / a = 2
Modulus %	Divides left hand operand by right hand operand and returns remainder	b % a = 0
Exponent **	Performs exponential (power) calculation on operators	a**b= 10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e. rounded away from zero (towards negative infinity)	9//2 = 4 and 9.0//2.0 = 4.0 -11//3= -4 and -11.0//3 = -4.0

TMY 2024

Order of Operations

- Computer programming uses the same order of operations as in mathematics. Which is (PEMDAS)
  1) **P**arentheses
  2) **E**xponents and Roots
  3) **M**ultiplication and Division
  4) **A**ddition and **S**ubtraction

Example: Given a = 2 and b = 3

a * b – a = 4

a * (b – a) = 2

Errors Types

- **Syntax errors**

  The most basic type of error. They arise when the Python parser is unable to understand a line of code. Syntax errors are almost always fatal, i.e. there is almost never a way to successfully execute a piece of code containing syntax errors.
- **Exception errors**

  Exceptions arise when the python parser knows what to do with a piece of code but is unable to perform the action. An example would be trying to access the internet with python without an internet connection; the python interpreter knows what to do with that command but is unable to perform it.
- **Logical errors**

  The most difficult to fix. They occur when the program runs without crashing, but produces an incorrect result. The error is caused by a mistake in the program's logic. You won't get an error message, because no syntax or runtime error has occurred. These type of errors have to be hunted down by tracing code.

Handling Exception Errors

- The following statement results in an error.

```
result = 2 / 0
  Message=ZeroDivisionError('division by zero',)
  Source=c:\users\rmcarthur.scdgroup\desktop\nait\cctp10\code examples\errorhandling\errorhandling\errorhandling.py
  StackTrace:
  File "c:\users\rmcarthur.scdgroup\desktop\nait\cctp10\code examples\errorhandling\errorhandling\errorhandling.py", line 2, in <module>    result = 2 / 0
```

- Error handling in Python is done through the use of exceptions that are caught in try blocks and handled in exception blocks.

Example:

```
try:
    result = 2 / 0
except:
    print('You cannot divide a number by zero")
```

https://docs.python.org/3/library/exceptions.html

TMY 2024

DEMO:Lab 2

Create a program that asks the user to input two numbers and divides the first number by the second. Make sure you take care of any errors.

1. Setup the try/exception block
2. Collect the inputs
3. Do the division
4. Output the result

TMY 2024

DEMO:Lab 3

Input(s): two numbers

create two inputs and capture the values

Output: the result of division numb1/numb2

do the division

print the results

1. Create two input statements and have variables to hold each input.
2. The rest of the code should be in a try and except block:
    1. Create a variable to hold the result of the division
    2. Print the result of the division.

Strings

Introduction

- We have used integers, floats, boolean and string data types.
- Strings are different than the previous data types we have looked at, they are made of smaller parts that contain an individual characters. They are known as a compound data type.

String: Hello

Index: 0 1 234 5

Introduction

- Strings are immutable (You cannot change the value of a single or multiple character of a string, once it has been initialized).
- You can however reassign the whole value of a string to another string

```
>>> str1 = "Tim"
>>> str1 = "John"
```

String functions

- Python has several built-in functions associated with the string data type.
- These functions let us easily modify and manipulate strings.
- We can think of functions as being actions that we perform on elements of our code.
- Built-in functions are those that are defined in the Python programming language and are readily available for us to use

Strings .upper() and .lower()

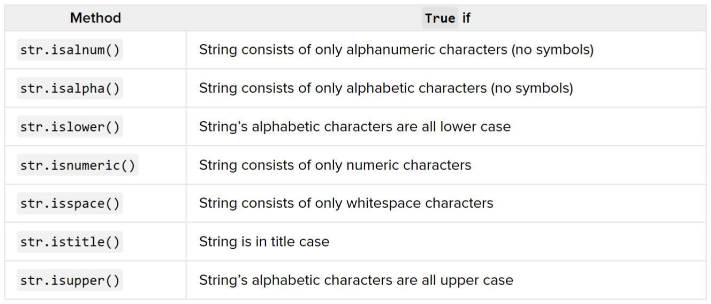
- The functions str.upper() and str.lower() will return a string with all the letters of an original string converted to upper- or lower-case letters.
- Because strings are immutable data types, the returned string will be a new string. Any characters in the string that are not letters will not be changed.
- .upper()

```
>>> strName = "Billy"
>>> print(strName.upper())
BILLY
```

- .lower()

```
>>> strName = "Billy"
>>> print(strName.lower())
billy
```

Strings Boolean Functions



Method	`True` if
`str.isalnum()`	String consists of only alphanumeric characters (no symbols)
`str.isalpha()`	String consists of only alphabetic characters (no symbols)
`str.islower()`	String's alphabetic characters are all lower case
`str.isnumeric()`	String consists of only numeric characters
`str.isspace()`	String consists of only whitespace characters
`str.istitle()`	String is in title case
`str.isupper()`	String's alphabetic characters are all upper case

```
>>> letters = "HelloWorld"
>>> numbers = "123456"
>>> print(letters.isnumeric())
False
>>> print(numbers.isnumeric())
True
```

Strings Determining Length len()

- The string method len() returns the number of characters in a string.
- This method is useful for when you need to enforce minimum or maximum lengths:
  - Password user lengths
  - Saving to a database that has a number of characters rules

```
>>> name = “Jimmy"
>>> print(len(name))
5
```

Strings .join(), split(), replace()

- .join()
  - The str.join() method will concatenate two strings, but in a way that passes one string through another:
    - >>> name_first = “John”
    - >>> print (“ “.join(name_first))

    J o h n

  - We can also use .join() to provide concatenation of a strings:
    - >>> name_last = “Appleseed”
    - >>> print(“ ”.join([name_first, name_last]))

    John Appleseed

Strings .join(), split(), replace()

- Just as we can join strings together, we can also split strings up. To do this, we will use the str.split() method. By default strings are split by white space:

  - >>>letters = “HelloWorld”
  - >>>print(letters.split())

  ['HelloWorld']

  - >>>full_name = “Mickey Mouse”
  - >>>print(full_name.split())

  ['Mickey', 'Mouse']

  - >>>colors = “red,blue,green”
  - >>>a,b,c = colors.split(“,”)
  - >>>print(a)

  red
  - >>>print(b)

  blue
  - >>>print(c)

  green

  - >>> newstring = colors.split(",")
  - >>> print(newstring)

  ['red', 'blue', 'green']

  - >>> print(colors.split())

  ['red,blue,green']

  - >>> newstring2 = "red, blue, green"
  - >>> print(newstring2.split())

  ['red,', 'blue,', 'green']

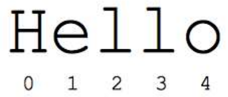
Strings .join(), split(), replace()

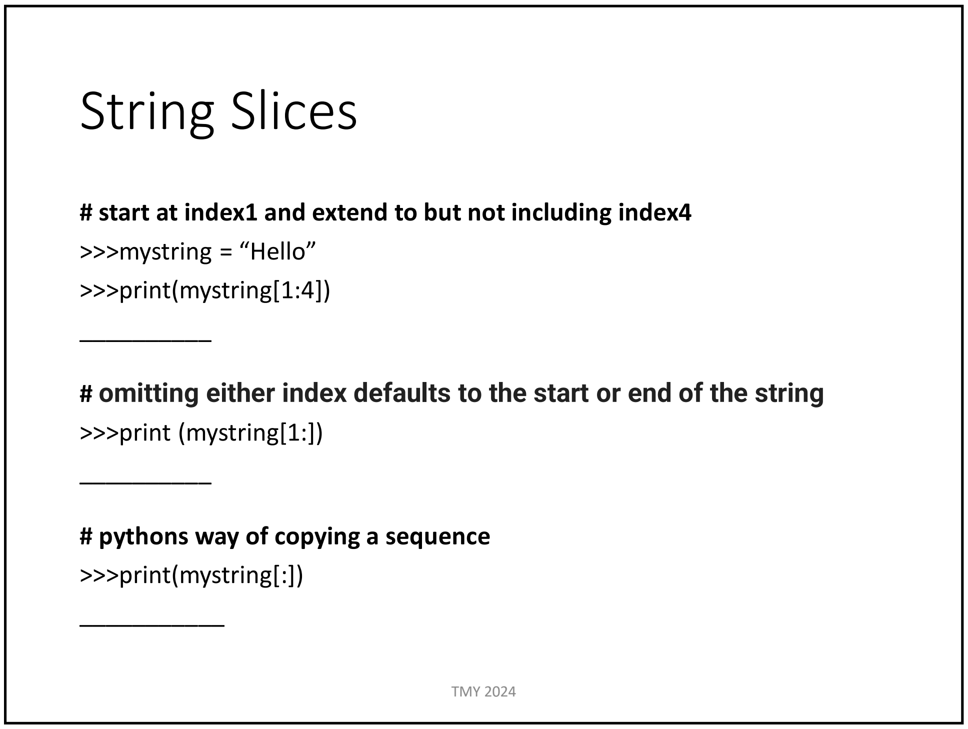
- The method **replace()** returns a copy of the string in which the occurrences of *old* have been replaced with *new*:
  - >>>full_name = “Mickey Mouse”
  - >>>print(full_name.replace(“M”, “m”))

  micky mouse

String Slices

- The python string slice syntax is a way to refer to sub-parts of a sequence (string).
- The slice syntax:
  - somestring_variable[start:end]
    - Where start is the strings index and the end is up to but not including the end index.





String Slices

# start at index1 and extend to but not including index4

```
>>>mystring = “Hello”
>>>print(mystring[1:4])
```

____

# omitting either index defaults to the start or end of the string

```
>>>print (mystring[1:])
```

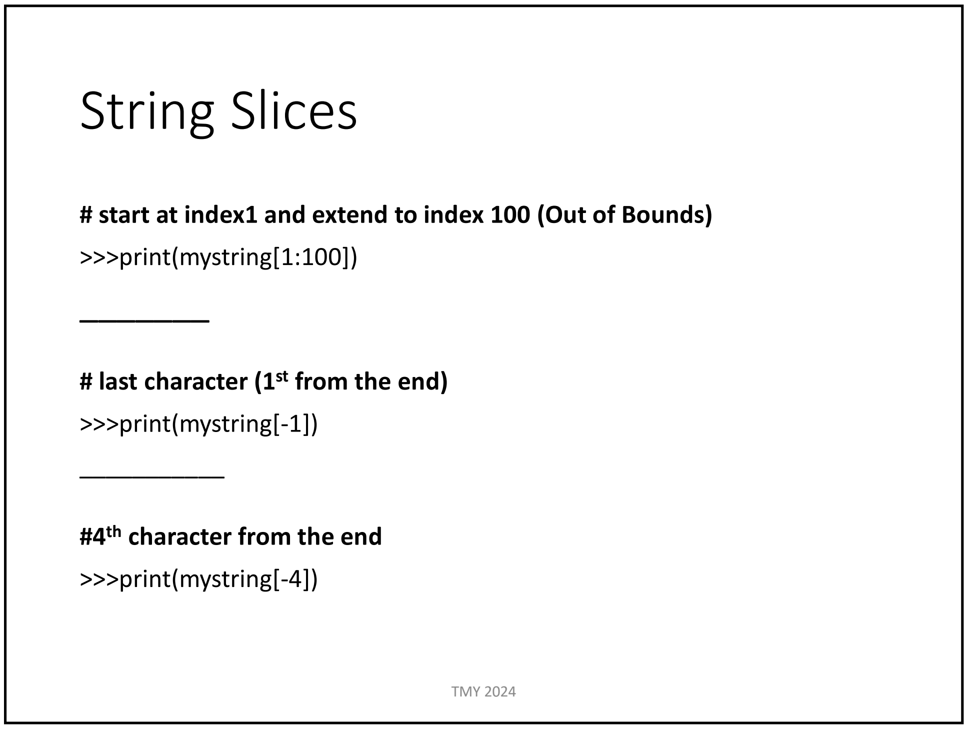
____

# pythons way of copying a sequence

```
>>>print(mystring[:])
```

____

TMY 2024



String Slices

# start at index1 and extend to index 100 (Out of Bounds)

```
>>>print(mystring[1:100])
```

____

# last character (1st from the end)

```
>>>print(mystring[-1])
```

____

#4th character from the end

```
>>>print(mystring[-4])
```

TMY 2024

DEMO: Lab 4

Write a program that outputs the following string. Try and create the output from a single print statement:

“Isn’t,” she said.

“Is too,” he said.

*Hint: You will need to investigate what an **escaped character** is.*

Data Structures

Lists

Tuples

Dictionaries

Sets

	Mutable	Ordered	Indexing/ Slicing	Duplicate Elements
List	✓	✓	✓	✓
Tuple	✗	✓	✓	✓
Set	✓	✗	✗	✗

List

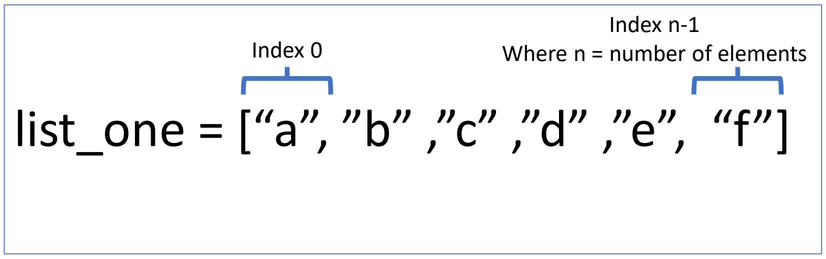
- One of the four Python collections (**List**, tuple, **dictionary**, set)
- General Purpose
- Most widely used collection
- Can grow and shrink as required
- Sortable

List: Declaring

- A list is a data type that allows the programmer to create a variable that holds multiple values.
- Unlike many other programming languages Python allows differing data types to be included in a list.
  - list_Average = [] # empty list
  - list_1 = **[1,2,3,4,"Five","Six"]**
  - list_test_scores_cctp10 = [50,45,68,90,25]

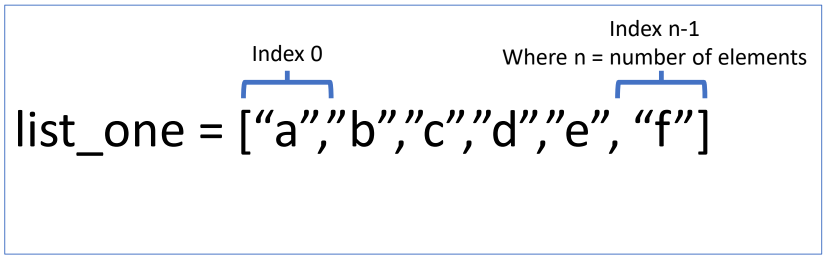
Lists Accessing Data

- Elements within a list are referred by their index value.
- Index values start from 0 to the number of elements -1.



```
list_one = ["a", "b" ,"c" ,"d" ,"e",  "f"]
```

Lists: Accessing Data


Index 0
Index n-1
Where n = number of elements
list_one = [“a”,”b”,”c”,”d”,”e”, “f”]

print(list_one) ________

print(list_one[0]) ________
print(list_one[2]) ________
print(list_one[10] ________

print(len(list_one)) ________

TMY 2024

List built-in methods

Method	Description
append()	Add Single Element to the list
extend()	Add elements of a list to another list
insert()	Insert element to the list
remove()	Removes element from the list
index()	Returns smallest index of element in list
count()	Returns occurrences of element in a list
pop()	Removes element at a given index
reverse()	Reverses a list
sort()	Sorts elements of a list
copy()	Returns shallow copy of a list
clear()	Removes all items from the list

TMY 2024

Lists: Example 1

# Create a list of numbers
# Syntax: arrayname = [element1,element2, etc…]

```
>>> numbers = [1,2,3,4,5]
>>> print(numbers)
```

Answer:_______________

# Find the number of elements within an array
# len(arrayname) returns the number of elements

```
>>> print(len(numbers))
```

Answer: ________________

TMY 2024

Lists: Example 2
>>> numbers = [1,2,3,4,5]

```
>>> print(numbers[0])
```

____

```
>>> print(numbers[1])
```

____

```
>>> print(numbers[20])
```

____

TMY 2024

Related Lists

- Suppose you had two lists:
  - One that holds names:
    - names = [“Sara,”Jimmy”,”Tom”]
  - Second array that holds ages:
    - ages = [23,21,30]

- It is common to have several arrays that hold data that are related by index value.

  [“Sara,”Jimmy”,”Tom”]

  [23,21,30]

Lists: Example 3

```
>>> names = [“Sara”, Jimmy”, ”Tom”]
>>> ages  = [23,21,30]

>>> print(names[0], “is”, ages[0])
_________________________
>>> print(names[1], “is”, ages[1])
_________________________
>>> print(names[2], “is”, ages[2])
_________________________
```

Lists: Example 4
Adding an element to the end of an list

- To add an element to the end of the list you can use the append method:

# Recall: names = [“Sara,”Jimmy”,”Tom”, ”New Guy"]

```
>>> names.append(“New Guy”)
>>> print(names)
```

____

TMY 2024

Lists: Example 5
Adding an element to a particular index of an array

# Syntax: listname.insert(indexToInsertAt, value)

```
# Recall: names = [“Sara”,”Jimmy”,”Tom”]
>>> names.insert(1,”New guy”)
>>> print(names)
```

____

TMY 2024

Lists: Example 6
Remove an element by the element value

# Syntax listname.remove(elementValue)

Recall: names = [“Sara,”Jimmy”,”Tom”, “New guy”]

>>> names.remove(“New guy”)

>>> print(names)

____

TMY 2024

Lists: Example 7
Remove the last element

# Syntax for last item removal: *listname*.pop()

# Recall: names = [“Sara,”Jimmy”,”Tom”]

>>> names.pop()

>>> print(names)

____

TMY 2024

Lists: Example 8
Remove Element by index value

# Syntax: del listname[indexValue]
# Recall: names = [“Sara”,”Jimmy”]

```
>>> del names[0]
>>> print(names)
```
____

TMY 2024

List: Example 9
Delete the entire contents

# Syntax: listname.clear()
# Recall: names = [”Jimmy”]

```
>>> names.clear()
>>> print(names)
```
____

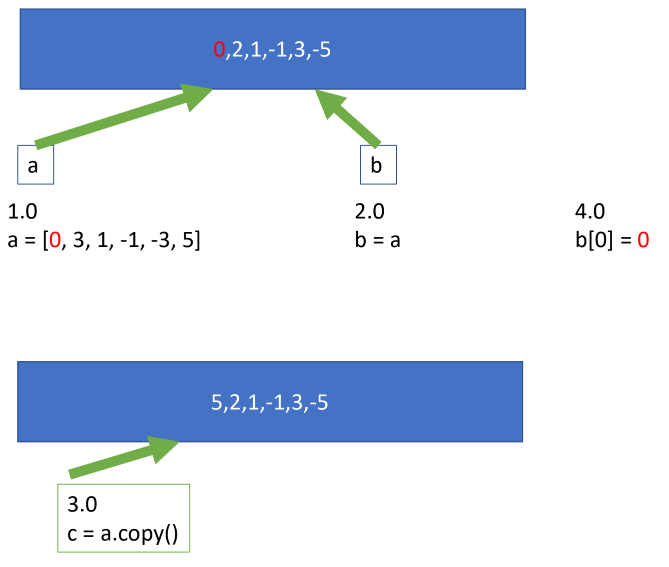
TMY 2024

DEMO: Lab 5

Given the following code:

```
# list "a"
a = [5, 3, 1, -1, -3, 5]

# make a new variable "b" and assign it as list "a"
b = a

# make a new variable "c" and assign it a copy of list "a"
c = a.copy()

# Update the first entry in b to be zero
b[0] = 0
```

Output a, b, c. What is the difference between variables "b" and "c"



Tuple

- Immutable (cannot change/add/delete)
- Useful for fixed data
- Faster than lists
- Sequence type

Tuples: Declaring

- **A tuple consists of a number of values separated by commas, for instance:**

```
>>> car_data = ‘Ford’, ‘Focus’, 1998, 221000
```

- **Tuples may be nested**

```
>>> car_service = car_data, (1, 2, 3, 4, 5)
```

- **Types can contain mutable objects:**

```
>>> cars = ([‘Charger’, ‘Challenger’, ‘Dart’], [‘Edge’, ‘Escort’, ‘Ranger’] )
```

Tuples: Accessing

```
>>> car_data
('Ford', 'Focus', 1998, 221000)
>>> car_data[0]
'Ford'
>>> car_service
(('Ford', 'Focus', 1998, 221000), (1, 2, 3, 4, 5))
>>> car_service[1][2]
3
```

- **Remember Tuples are immutable. You will get an error if you try this:**

```
>>> car_data[3] = 221500
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    car_data[3] = 221500
TypeError: 'tuple' object does not support item assignment
```

- **However changing mutable objects within a Tuple is valid**

```
>>> cars[0][2] = 'Viper'
>>> cars
(['Charger', 'Challenger', 'Viper'], ['Edge', 'Escort', 'Ranger'])
```

Tuples: Built in Methods

Method	Description
count()	Returns occurrences of element in a tuple
index()	Returns smallest index of element in tuple

DEMO:Lab 6

- Capture from two inputs a student name and a test score and store the information in 2 lists
- Capture the information 3-times for three different students
- When done capturing the information, display a summary for the three students at the end of the application:
  - Name: John Test Score: 20
  - Name: Sue Test Score: 21
  - Name: Kate Test Score: 52
- Bonus: Try and calculate the average grade for the three students?

Dictionary

- Key/Value pairs
- Associative arrays

Dictionaries: Declaring

Key Value

```
>>> employees = {'Robert': 4098, 'Tim': 1221, 'Kelly' : 1822}
```

Element Element Element

Dictionaries: Accessing

```
>>> employees = {'Robert':4098, 'Tim': 1221, 'Kelly': 1822}
>>> employees.values()
dict_values([4098, 1221, 1822])
>>> employees.keys()
dict_keys(['Robert', 'Tim', 'Kelly'])
>>> employees['Robert']
4098
>>> list(employees)
['Robert', 'Tim', 'Kelly']
>>> list(employees)[1]
'Tim'
```

Using the list constructor to create a list from a dictionary

Dictionaries: Built in Methods

Method	Description
clear()	Removes all items
copy()	Returns shallow copy of a dictionary
fromkeys()	Creates dictionary from given sequence
get()	Returns value of the key
items()	Returns view of dictionary's (key, value) pair
keys()	Returns view object of all keys
popitem()	Returns and removes element from dictionary
setdefault()	Inserts key with a value if key is not present
pop()	Removes and returns element having given key
values()	Returns view of all values in dictionary
update()	Updates the dictionary

Set

- Store non-duplicate items
- Very fast access compared with lists
- Unordered
- Does not support indexing
- Good for mathematical operations

Sets: Declaring

```
>>> basket = {'apple', 'orange', 'pear', 'orange', 'banana'}
```

Sets: Accessing

```
>>> basket
{'orange', 'pear', 'banana', 'apple'}
```

Note that the duplicate 'Orange' has been removed

- **Remember Sets do not support indexing**

```
>>> basket[3]
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    basket[3]
TypeError: 'set' object does not support indexing
```

- **Fast membership testing**

```
>>> 'orange' in basket
True
>>> 'crabgrass' in basket
False
>>>
```

Sets: Built in Methods

Method	Description
remove()	Removes element from the set
add()	Adds element to a set
copy()	Returns shallow copy of a set
clear()	Remove all elements from a set
difference()	Returns difference of two sets
difference_update()	Updates calling set with intersection of sets
discard()	Removes an element from the set
intersection()	Returns intersection of two or more sets
intersection_update()	Updates calling set with intersection of sets
isdisjoint()	Checks disjoint sets
issubset()	Checks if a set is subset of another set
pop()	Removes an arbitrary element
symmetric_difference()	Returns symmetric difference
symmetric_difference_update()	Updates set with symmetric difference
union()	Returns union of sets
update()	Add elements to the set

Controlling Program Flow

Introduction

- Controlling program flow is typically done by decisions and loops.
  - An application can perform an action based upon a decision based upon the value of a input.
  - Optionally more decisions can be made and more actions can be performed by the application
- Suppose you wanted to make an application that would allow you to evaluate a set of test scores?

If statements

- By creating a condition that is either True or False we can use logic to control our application.
- If statements use a condition evaluated using a comparison operator to control application flow:
  - *If **grade >= 60**, then the student passed the course*
  - *If **grade < 60**, then the student failed the course*
  - *If **grade >= 90**, then the student received honors for the course*
  - *If **grade == 100**, then the student received 100% for the course*

If statements

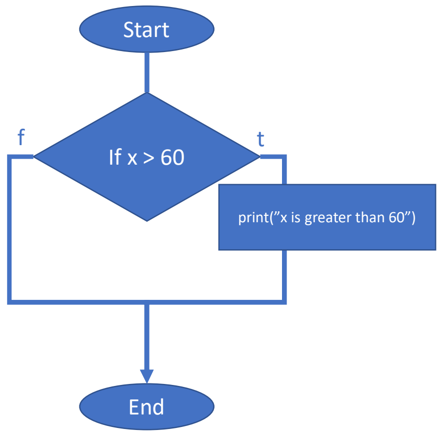
- *To evaluate an expression within a if statement comparison operators are typically used:*

```
x == y                  # Produce True if ... x is equal to y
x != y                  # ... x is not equal to y
x > y                   # ... x is greater than y
x < y                   # ... x is less than y
x >= y                  # ... x is greater than or equal to y
x <= y                  # ... x is less than or equal to y
```

Multiple conditions can be evaluated in a single comparison statement by using "and" and "or"

TMY 2024

Simple If Statement



Syntax:

if condition :

Indented Statement Block

- If the condition is true, then do the indented statements. If the condition is not true, then skip the indented statements.

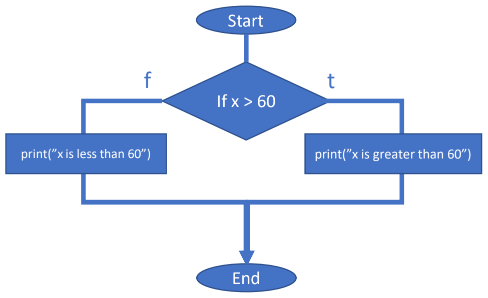
Python Code:

```
If x > 60:
   print("x is greater than 60")
```

Note: The condition must evaluate to true or false and by using the simple if statement we only do something if the statement evaluates to True.

TMY 2024

If-else Statement



Syntax:

if condition :

   Indented Statement Block for true

else:

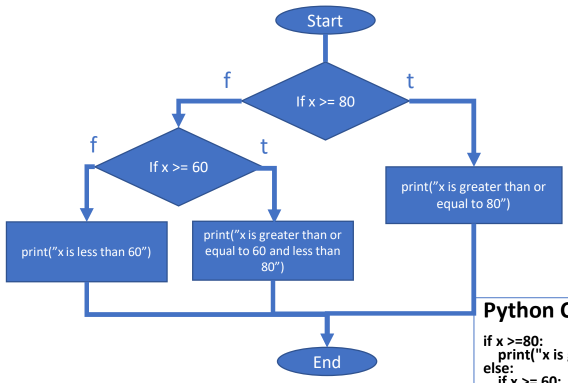
   Indented Statement Block for false

- If the condition is true, then do the indented True statements. If the condition is not True (False), then do the indented False statements.
- In this example there are only two possible outcomes, namely True or False.

Python Code:

```
if x > 60 :
   print("x is greater than 60")
else:
   print("x is less that 60")
```

Nested If else Statements


Start
If x >= 80
f
t
If x >= 60
f
t
print("x is greater than or equal to 80")
print("x is less than 60")
print("x is greater than or equal to 60 and less than 80")
End

Allows for THREE conditions to be met

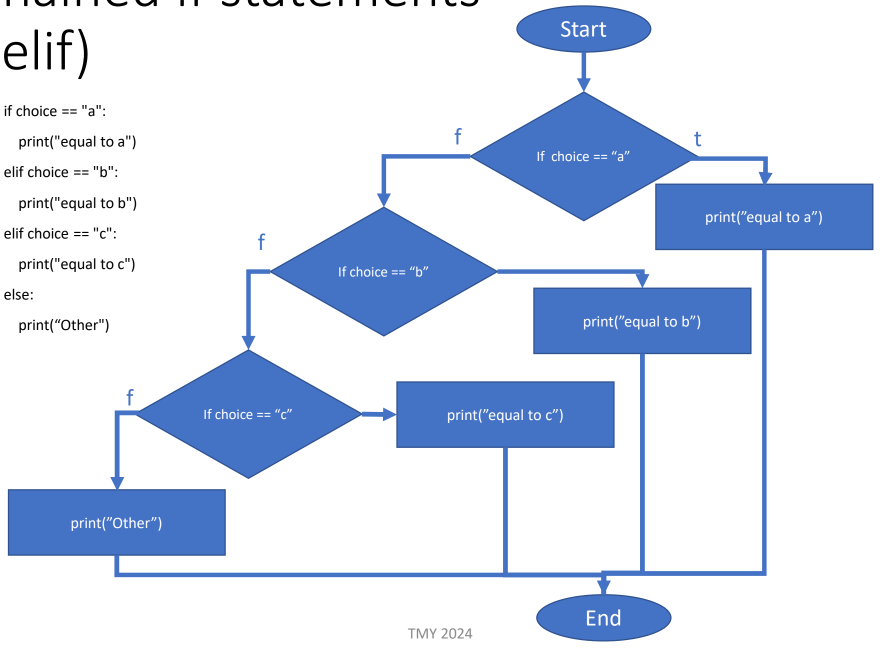
1: 80 or 80+
2: 60 or 60+
3: x < 60

Python Code:

```
if x >=80:
   print("x is greater than or equal to 80")
else:
   if x >= 60:
      print("x is greater than or equal to 60 and less than 80")
   else:
      print("x is less than 60")
```

Chained if statements (elif)

```
if choice == "a":
    print("equal to a")
elif choice == "b":
    print("equal to b")
elif choice == "c":
    print("equal to c")
else:
    print("Other")
```



DEMO: Lab 7

Create a program that uses a list of students and a separate list of their grades and outputs if they passed or failed (Assume passing grade = 50)

Test Data:

list_students = [“Tom, “Kate,”Susan”]

list_grades = [80,90,40]

Iteration

Introduction

- Python provides the programmer with several means to perform repetitive tasks.
- Loops are the primary means to perform the same action a number of times.
- The number of times the action is performed is up to the programmer to define.

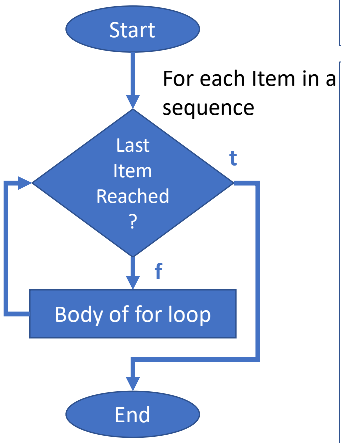
Scope

- For loops
- While loops

TMY 2024

For Loop

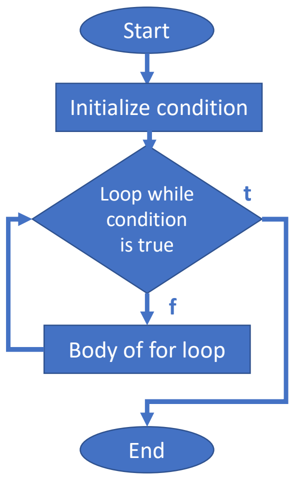
Commonly used for looping a sequence of items:
-strings
-lists



```
# Program to find the sum of all numbers stored in a list
# List of numbers (list sequence)
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
# variable to store the sum
sum = 0
# iterate over the list
for val in numbers:
    sum = sum + val

# Output: The sum is 48
print("The sum is", sum)
```

TMY 2024

While Loop



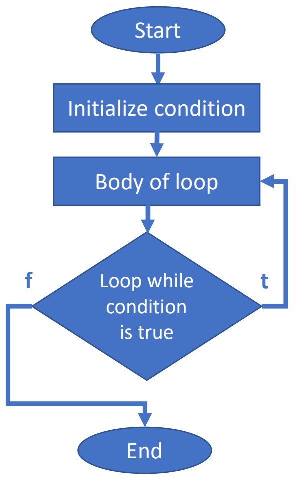
Commonly used when you DO know how many times to loop

```
# Program loop ten times
i = 0
while i < 10:
    i  = i + 1
```

```
# Program to calculate the sum of the value of i looping ten times
i = 0
mysum = 0
while i < 10:
     mysum = mysum + i
    i  = i + 1
```

TMY 2024

While Loop With Break

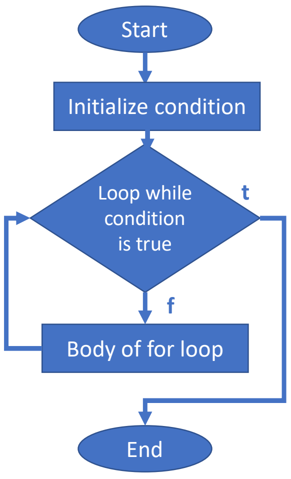


Commonly used when you want to execute the body of the loop at least **once**. The loop condition is tested at the end of the body of the loop

```
condition = True
while (condition == True):
   print("Hello")
   response = input("Do another (y/n)")
   if response == "n":
      condition = False
```

TMY 2024

While Loop With Break



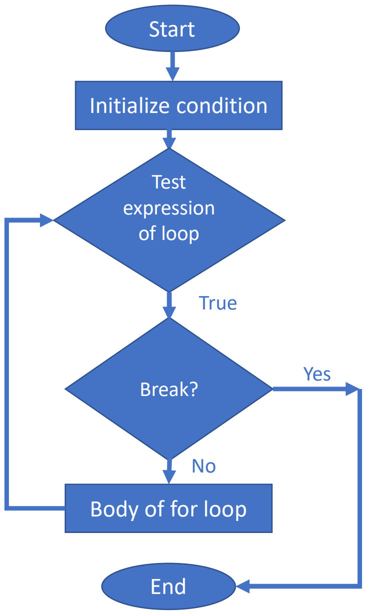
Commonly used when you DO know how many times to loop

```
# Program loop ten times
i = 0
while i <= 10:
    i  = i + 1
```

```
# Program to calculate the sum of the value of i
looping ten times
i = 0
mysum = 0
while i <= 10:
     mysum = mysum + i
    i  = i + 1
```

TMY 2024

Break Statement



A break statement can be used to end a loop when a condition is met.

```
# Initialize Variable
i = 0
# Loop forever
while True:

   #  Check to see if the condition has been  met
   if i > 9:

      # Break out of the loop
      break

   print(i)
   i = i + 1
```

TMY 2024

DEMO:Lab 8: Hints

- Create two lists.
  - One for the student names:
    - [“John”, ”Bill”, “Susan”, ”Sunny”]
  - One for the student grades
    - [50, 60, 95, 75]
  - Note the index of the two arrays relates the data.
- Loop through the lists and output the data and use a variable to hold the total averages and use that total to determine the class average.

```
John average is: 50
Bill average is: 60
Susan average is: 95
Sunny average is: 75
The class average is: 70.0
```

DEMO: Lab 9

- **Hint: check out the “Nested If else Statements” slide**
- Using the student list from Lab 8 determine the total passed, failed and received honours.
  - Failed less than 50
  - Pass greater than or equal to 50 and less than or equal to 80
  - honours greater than 80
- Output should be similar to:

```
number passed: 2
number failed: 1
number honours: 1
```

Functions

Introduction

- Functions are blocks of code that only run when required (called).
- Consider a simple calculator program:

Addition Function
Subtraction Function
Multiplication Function
Division Function

- If the user starts the calculator and performs the addition of two numbers, then the program will only call the function that performs addition.
  - The program will ignore the function code that performs:
    - Subtraction, Multiplication and division
- By placing the code to perform mathematical operations in separate functions it:
  - Allows us to reuse the function as required
  - Allows the function to become portable (You can reuse the function in another application)

Function Syntax

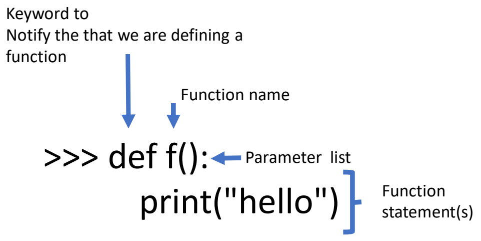
```
def functionname(arg1, arg2, ...):
        statement1
        statement
        ...
```

Note: arguments (arg(s)) are inputs to the function, not all function require them.
Note: Bold text is required when defining a function

Note: If you define a argument in this way it will be required when using the function

TMY 2024

Super Simple Function



```
>>> def f():
        print("hello")
```

- We have defined a function named “f” that prints the text “hello”, whenever we call the function.

Calling the function:
```
>>> f()
hello
```

TMY 2024

Function with input parameters and a return value

```
def add_two_numbers(number1, number2):
    numb1 = number1
    numb2 = number2
    total = numb1 + numb2
    return total
```

Note:
- numb1, numb2 and total are declared within the function so they only have “scope” (visibility) within the function
- The return statement allows the function to provide a value when it is called

Calling the function:

```
>>> add_two_numbers(10,3)
13
```

Function with keyword arguments (default Arguments) and a return value

Keyword Arguments

```
>>> def add_Two_Numbers(numb1=0, numb2=0):
        number1 = number1
        number2 = number2
        total = number1 + number2
        return total
```

Keyword arguments allows us to set a default value(s) for input parameters and make the use of the input parameter optional

Function with keyword arguments (default Arguments) and a return value

$$\overbrace{\texttt{>>> def add\_Two\_Numbers(numb1=0, numb2=0):}}^{\text{Keyword Arguments}}$$

```
>>> def add_Two_Numbers(numb1=0, numb2=0):
        return numb1 + numb2
```

Keyword Arguments

Shortened Syntax

TMY 2024

Function with keyword arguments (default Arguments) and a return value

```
>>> def fn(x, y = 1):
        return x + y
```

```
>>> fn(2)
________

>>> fn(y=100)
________

>>> fn(10, y= 110)
________
```

TMY 2024

Namespaces and Scope

Namespaces

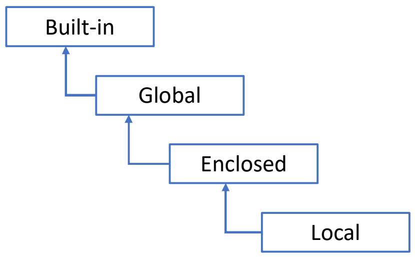
- Namespaces are just containers for mapping names to objects
- Everything in Python – Literals, lists, dictionaries, functions, classes are objects
- Such a “name-to-object” mapping allows us to access an object by a name that we’ve assigned to it.
  Eg. If we make a simple assignment via a_string = “Hello String”, we create a reference to the “Hello String” object, and henceforth we can access via its variable name a_string
- Everytime we call a for-loop or define a function, it will create its own namespace.
- Namespaces also have different levels of hierarchy called “scope”

Variable Scope

- Scope in Python defines the "hierarchy level" in which we search namespaces for certain "name-to-object" mappings
- Python uses the LEGB rule to determine the different levels of namespaces before it finds the name-to-object mapping you are referencing in your program.
- LEGB stands for:
  - **Local** can be inside a function or class method
  - **Enclosed** can be its enclosing function. If a function is wrapped inside another function.
  - **Global** refers to the uppermost level of executing code itself
  - **Built-in** are special names that Python reserves for itself.

TMY 2024

LEGB: Order of operation



1) If a particular name to object mapping cannot be found in the local namespaces
2) The namespaces of the enclosed scope will be searched next
3) If the search in the enclosed scope is unsuccessful, Python moves on to the global namespace
4) Finally Python will search in the built-in namespaces

Note: If the variable name cannot be found in any namespace, a NameError will be raised.

TMY 2024

Variable Scope: Local Variables

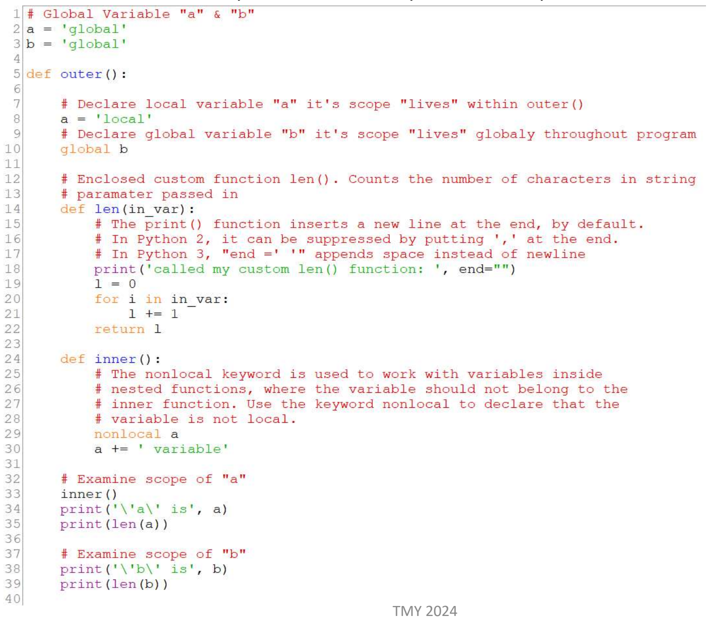
- Consider the following two function declarations:

```
def fn_one(x,y):
    const = 10
    return x * y
```

```
def fn_two(x,y):
    const = 20
    return x * y
```

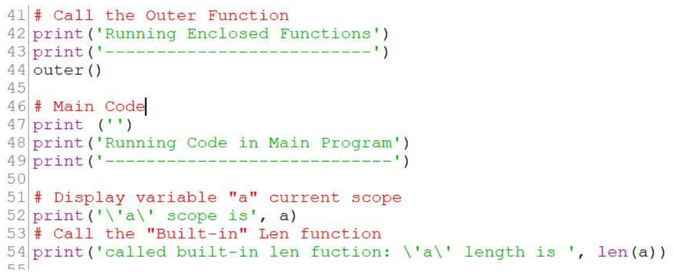
- Both functions have their own variables x,y and const variables.
- These variables are “local” to the function and are only accessible within each function.

LEGB – Local, Enclosed, Global, Built-in scope



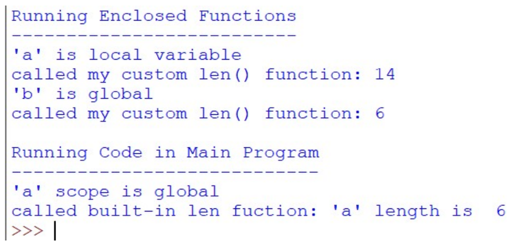
```
# Global Variable "a" & "b"
a = 'global'
b = 'global'

def outer():

    # Declare local variable "a" it's scope "lives" within outer()
    a = 'local'
    # Declare global variable "b" it's scope "lives" globaly throughout program
    global b

    # Enclosed custom function len(). Counts the number of characters in string
    # paramater passed in
    def len(in_var):
        # The print() function inserts a new line at the end, by default.
        # In Python 2, it can be suppressed by putting ',' at the end.
        # In Python 3, "end =' '" appends space instead of newline
        print('called my custom len() function: ', end="")
        l = 0
        for i in in_var:
            l += 1
        return l

    def inner():
        # The nonlocal keyword is used to work with variables inside
        # nested functions, where the variable should not belong to the
        # inner function. Use the keyword nonlocal to declare that the
        # variable is not local.
        nonlocal a
        a += ' variable'

    # Examine scope of "a"
    inner()
    print('\'a\' is', a)
    print(len(a))

    # Examine scope of "b"
    print('\'b\' is', b)
    print(len(b))

```

LEGB – Local, Enclosed, Global, Built-in scope



```
# Call the Outer Function
print('Running Enclosed Functions')
print('--------------------------')
outer()

# Main Code
print ('')
print('Running Code in Main Program')
print('----------------------------')

# Display variable "a" current scope
print('\'a\' scope is', a)
# Call the "Built-in" Len function
print('called built-in len fuction: \'a\' length is ', len(a))
```

Output:



```
Running Enclosed Functions
--------------------------
'a' is local variable
called my custom len() function: 14
'b' is global
called my custom len() function: 6

Running Code in Main Program
----------------------------
'a' scope is global
called built-in len fuction: 'a' length is  6
>>> 
```

TMY 2024

Python Directories and Folders

Scope

- What is a file?
- Typical Workflow for File Handling
- Reading from files
- Creation and writing to files
- Deleting Files

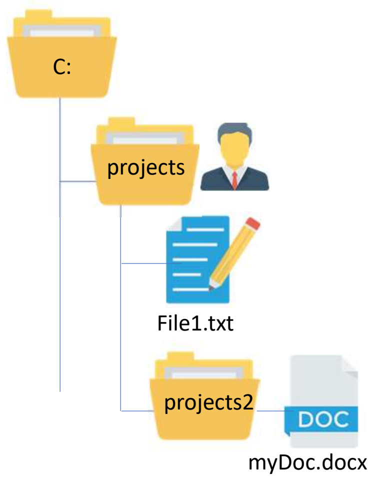
Folder and File Paths

- A location of a file within a single system can be relative or absolute.
- File paths are represented by string values and have three major parts:
    1. Folder Path - the folder path of where the file resides
    2. File Name - the name of the file
    3. Extension - the end of the file name after the period to indicate the file type

File Path – Same Directory

- Python can reference files using various methods. If the file exists in the current directory it can be accessed by just using the file name. The system will interpret the absence of a path as the current directory.

File Path – Absolute and Relative



Absolute paths always begin with the root directory of a disk partition (drive letter).
Example:
- Absolute path to the file named “File1.txt would be:
  - C:\projects\File1.txt

Relative path always begin with the disk partition (drive letter).
Example:
- Relative path to the file named “File1.txt from current directory of projects would be:
  - File1.txt

Windows uses backslashes, while Linux-based systems use forward slashes to separate directories.
Python solves this problem by using:
os.path.join(“C”, ”projects”,”projects2”)

Handy Folder Commands

- Python has a os module that can be imported into a project.
- Handy commands:
  - os.get.cwd()
    - Returns the current working directory
    - Eg: c:\\projects
  - os.chdir(*fullpath to change to*)
    - Change the working directory
    - Note Python will generate an error if the folder does not exist

Creating New Folders

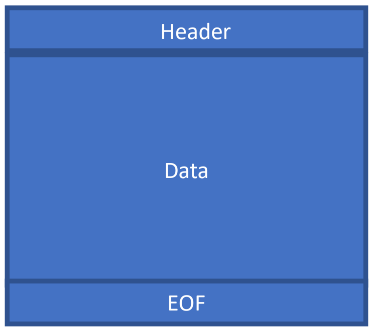
- Using the mkdir() function directories can be created:
  - Use:
    - os.mkdir(**"path and folder name"*)
    - * If no path is supplied Python will make the directory in the Working Directory
- Eg #1:
  - os.mkdir("test")
  - If the folder already exists Python will generate an error.

Renaming Folders

- Python provides a simple way to rename folders via the use of the built-in function rename().
- Use:
  - os.rename(src, dest)
  - src = old name
  - dest = new name

Python File Handling

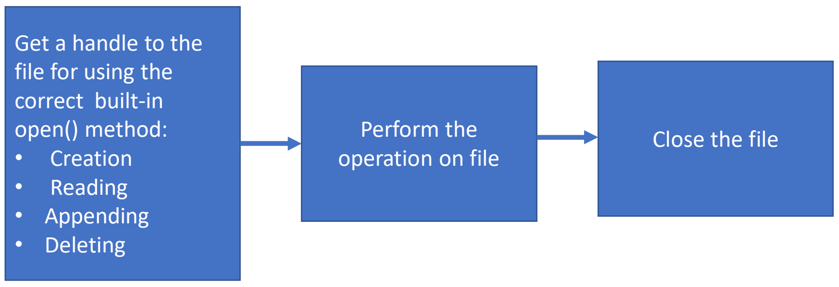
What is a file?



- A file is a contiguous set of bytes used to store data,
- Most modern file systems have three main parts to a file:
  1. Header – metadata regarding the file (name, size, type, creation date…)
  2. Data – content created by activity. The type of data is represented bu the file extension.
  3. End of File (EOF) – A special character that indicates the end of the file.

TMY 2024

Typical Workflow



Get a handle to the file for using the correct built-in open() method:
- Creation
- Reading
- Appending
- Deleting

Perform the operation on file

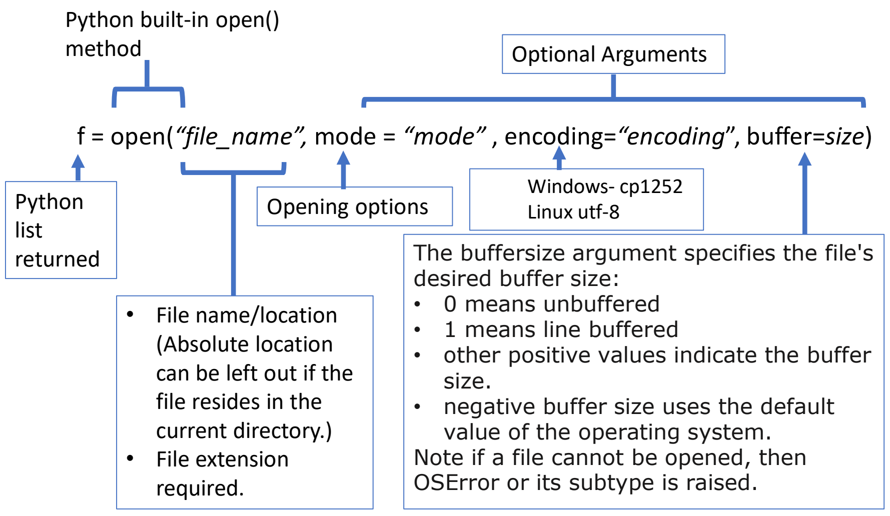
Close the file

TMY 2024

File Handling

- A common task in Python programming is to deal with tasks of file creation, adding and deleting content.
- Python has built-in methods to deal with files.
  - (https://docs.python.org/3/tutorial/inputoutput.html)
- Files are named locations on a persistent storage device (e.g. hard disk).
- Common operations include:
  - Creation of a file
  - Reading or writing to a file
  - Editing the file contents
  - Closing the file

Opening a File Syntax



Python built-in open() method

Optional Arguments

f = open(*“file_name”,* mode = *“mode”* , encoding=*“encoding”*, buffer=*size*)

Python list returned

Opening options

Windows- cp1252
Linux utf-8

- File name/location (Absolute location can be left out if the file resides in the current directory.)
- File extension required.

The buffersize argument specifies the file's desired buffer size:

- 0 means unbuffered
- 1 means line buffered
- other positive values indicate the buffer size.
- negative buffer size uses the default value of the operating system.

Note if a file cannot be opened, then OSError or its subtype is raised.

File Open and handling Modes Simple

File Open Mode	Description
"r"	Read (Default) - Opens a file for reading (will return an error if the file cannot be found).
"a"	Append – Opens a file for appending and will create the file if it does not exist at the specified location.
"w"	Write – Opens a file for writing and will create a file if it does not exist at the specified location.
"x"	Create – Creates the file and will return an error if the file name exists at the specified location

File Handling Mode	
"t"	Text – Default- Strings
"b"	Binary – Opens the file as bytes, for images, executable files (non-text files)

File Open Modes: Expanded

File Open Mode	
"rb"	Open a file for reading only in binary format
"r+"	Open a file for reading and writing
"rb+"	Open a file for reading and writing in binary format
"wb"	Open a file for reading only in binary format
"ab"	Open a file for appending in binary format
"a+"	Open a file for both appending and reading
"ab+"	Open a file for both appending and reading in binary format

Note: "+" added to a mode allows for updating a file (reading and writing)

Opening a file – Encoding

- Encoding is the process of converting data from one form to another.
- Character encoding involves taking a file and its characters and using a type of "encoding" to convert it to binary data.
- Default encoding is platform dependent.
    - Windows- cp1252
    - Linux utf-8
- If applications cannot read the specific type of encoding the results will appear with strange symbols.

Common Options for Reading from a file

File Reading function	rescription
"readline()"	Reads the character from the current position up to a newline (/n) character.
"read(chars)" -Where chars is an integer	Reads the specified number of characters, starting from the current position. Useful for large datasets.
"readlines()"	Reads all the lines until the end of the file and returns a list object

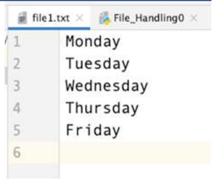
DEMO:File_Handling: Creation

```
# CREATE A FILE NAMED file1.txt
f = open("file1.txt", mode="wt” buffering=1)

# PRINT THE TYPE FOR THE OUTPUT
# USE OF THE /n adds a newline
print(type(f))

# WRITE TO THE FILE
f.write("Monday\nTuesday\nWednesday\nThursday\nFriday\n")

# What happens with this commmand?
print(f.readlines())
```

Using the write and text arguments

Newline cartridge return

Output



```
Monday
Tuesday
Wednesday
Thursday
Friday
```

1. Try running the code again and note the output in file1.txt?
2. Is this what you expected?
3. At the end of the add the following command:
   - print(f.readlines())

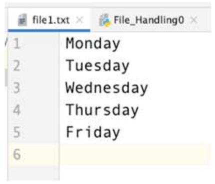
DEMO:File_Handling1: Reading

```
# CREATE A HANDLE TO THE FILE NAMED file1.txt
f = open("file1.txt", mode=”r”)

# What happens with this command?
print(f.readlines())
# Close the file
f.close
```

Using the read mode

Output

```
['Monday\n', 'Tuesday\n', 'Wednesday\n', 'Thursday\n', 'Friday\n']
```



```
Monday
Tuesday
Wednesday
Thursday
Friday
```

- File was opened via the “r” read mode

DEMO:File_Handling1: Reading

```
f = open("file1.txt")
print(f.readlines())
```

or

```
f = open("file1.txt”, mode=“r”)
print(f.readlines())
```

or

```
f = open("file1.txt”, mode=“r”, encoding =“utf-8”)
print(f.readlines())
```

file1.txt

```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

Output

```
['Monday\n', 'Tuesday\n', 'Wednesday\n', 'Thursday\n', 'Friday\n', 'Saturday\n',
'Sunday']
```

TMY 2024

DEMO:File_Handling2: Reading with a “while-loop”

```
# CREATE A HANDLE TO THE FILE NAMED file1.txt
f = open("file1.txt", mode="r")

# USE A WHILE LOOP
while True:
    try:
        single_line = next(f).splitlines()
        print(single_line)
    except StopIteration:
        break

# CHECK IF THE FILE IS CLOSED
print(f.closed)

# Close the file
f.close()

# CHECK IF CLOSED AGAIN
print(f.closed)
```

- splitline() allows for the return of a list object for every line with no blank-lines between entries.

Returns a set of lists

Output

```
['Monday']
['Tuesday']
['Wednesday']
['Thursday']
['Friday']
False
True
```

TMY 2024

DEMO:File_Handling2: Reading with a “with-loop”

```
# CREATE A HANDLE TO THE FILE NAMED file1.txt
f = open("file1.txt", mode="r")

with f as myFile:
    read_data = f.read()

print(read_data)

# CHECK IF FILE CLOSED
print(f.closed)
```

The python documentation suggests doing it this way because:

1. File is automatically closed, even if an exception is raised.
2. Shorter syntax.

Returns text and the cartridge returns are read and executed

Output

```
Monday
Tuesday
Wednesday
Thursday
Friday

True
```

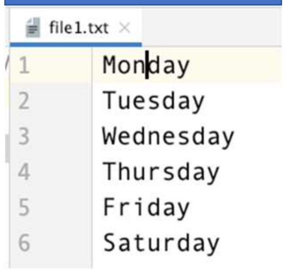
TMY 2024

DEMO:File_Handling2: Appending

```
# CREATE A HANDLE TO THE FILE NAMED file1.txt
f = open("file1.txt", mode=”a")

# Add content
f.write("Saturday")


# What happens with this command?
print(f.readlines())
```

Using the append mode

File


file1.txt

```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

- File was opened via the “a” append mode.
- Try and print the results
    - Change the mode to “a+” and try and print again.

TMY 2024