Data Science

DAI-101 Spring 2025-26

Dr. Devesh Bhimsaria
Office: F9, Old Building
Department of Biosciences and Bioengineering
Indian Institute of Technology–Roorkee
devesh.bhimsaria@bt.iitr.ac.in

Python review

What is the Python Shell?

- The **Python Shell** is an interactive environment where you can directly execute Python code and see immediate results. It is also known as the **Python interactive interpreter**.

Python Installations

- Conda: Anaconda/Miniconda (Pycharm, etc.)
- Jupyter notebook
- Python IDLE (Integrated Development and Learning Environment)
- VSCode etc.
- Note there is python 2 and python 3 versions

Interactive python

- Terminal- Unix/Linux & Mac OS
  - Type “python” enter- you’ll get prompt with >>>
- Python IDLE
- Jupyter notebook: Web based interactive platform

Running python code

- Terminal- Unix/Linux & Mac OS
  - python filename
  - chmod 0755 filename
  - ./filename
- VS Code and others

Basic Data Types

Boolean type

```
>>> a = True
>>> a
True
>>> type(a)
<type 'bool'>
>>> b = bool(0)
>>> b
False
```

Numerical Types

- Integer – int

```
>>> x = 5
>>> x == 5
True
>>> y = x
```

- Floating point – float

```
>>> a = 0.1
>>> b = 0.2
>>> c = a + b
>>> a
0.1
>>> c
0.30000000000000004 # Binary Floating-Point Representation & Accumulated Rounding
Errors. use the decimal module
>>> float(1)
1.0
>>> c == 0.3
False
```

is and *is not*

- *is and is not are used to compare*

```
>>> y = x
>>> x is y
True
>>> y += 1
>>> x is y
False
>>> x is not y
True
```

Comparison Operators

same values	`==`
not same values	`!=`
same object	`is`
not same object	`is not`
less than	`<`
less than or equal to	`<=`
greater than	`>`
greater than or equal to	`>=`

- All comparison operations yield a Boolean value
- Use `is/is not` with None, `True`, and `False`
- Can chain inequalities: `1 < x <= 4`

None

- *`None` is a special identity*
- *Like `NULL` in `C/C++`*

```
>>> p = None
>>> p is None
True
>>> x is not None
True
>>> print(type(None))
<class 'NoneType'>
```

None

None is treated as False in a boolean context.

The type of None is NoneType.

Like other singleton objects in Python (e.g., True, False), None is immutable. Its value cannot be changed.

```
print(None == False) # Output: False

print(None == 0) # Output: False

print(None == "") # Output: False
```

Python Strings – str

```
>>> s = "abc"
>>> s
'abc'
>>> type(s)
<type 'str'>
>>> str(1)
'1'
>>> len(s)
3
>>> s2 = 'ab'
>>> s2 += 'c'
>>> s2
'abc'
>>> s == s2
True
```

Integers & Strings are Immutable

This means that once they are created, their value cannot be changed. Instead, any operation that seems to modify them actually creates a new object.

Lists, dictionaries, sets, etc. are mutable.

```
>>> a = 'abc'
>>> id(a)
6776912
>>> a += 'd'
>>> id(a)
7775840
>>> i = 5
>>> j = i
>>> i += 5
>>> i is j
False
```

Lists

- Used to create arrays, stacks, FIFOs. Mutable.

```
>>> l = [ 1, 2, 3, 4 ]
>>> id(l)
4352115136
>>> l += [ 5 ]
>>> id(l)
4352115136
>>> l.append( 6 )
>>> l
[1, 2, 3, 4, 5, 6]
>>> l.pop()
6
>>> l.pop(0)
1
>>> l
[2, 3, 4, 5]
>>> l[1:3] # Start from first index to second index-1
[3, 4]
>>> len(l)
4
```

Lists operators

```
x = [42, 't', 1.3]
```

length	`len(x)`	`3`
concatenate	`[1, 2] + x`	`[1, 2, 42, 't', 1.3]`
membership	`42 in x`	`True`
slice	`x[0:2]`	`[42, 't']`
append	`x.append(3)`	`x: [42, 't', 1.3, 3]`
extend	`x += [3, 1]`	`x: [42, 't', 1.3, 3, 1]`
insert	`x.insert(1, 'a')`	`x: [42, 'a', 't', 1.3]`
delete	`del x[1]`	`x: [42, 1.3]`
remove	`x.pop(1)`	`'t'   x: [42, 1.3]`

Tuples

- Immutable lists. Usage e.g. days of a week, months.

```
>>> l = [ 1, 2, 3, 4 ]
>>> t = tuple(l)
>>> t
(1, 2, 3, 4)
>>> l == t
False
>>> l2 = list(t)
>>> l == l2
True
>>> len(l) == len(t)
True
>>> t.append(7)
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    t.append(7)
AttributeError: 'tuple' object has no attribute 'append’
>>> t = (1, 2)
>>> t += (3, 4) # Created new tuple & added to `t`
>>> t+=(3) # Error as 3 is now treated as integer
>>> t+=(3,) # Correct
```

Sets

- *A `set` object is an unordered collection of distinct hashable objects*
- *Mutable*

```
>>> foo = set( [1,2,3] )
>>> 2 in foo
True
>>> 5 in foo
False
>>> foo.add( 2 )
>>> foo
set([1, 2, 3])
>>> foo.add(6)
>>> foo.remove(2)
>>> foo
Set([1, 3, 6])
>>> other = set([4, 5, 6, 7])
>>> foo & other
set([6])
```

Frozen set

- *A `frozenset` is an immutable set*

```
>>> bar = frozenset( foo )
>>> bar.remove(6)
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    bar.remove(6)
AttributeError: 'frozenset' object has no attribute
'remove'
```

Dictionaries

- *A dictionary object maps hashable values to arbitrary objects*

```
>>> d = { 'a':3, 'b':2, 3:'x' }
>>> d['a']
3
>>> d.get( 5 )
>>> d.get(3) # Use 'get()' if you don’t know if the key is present
'x'
>>> d['x'] = 666
>>> d
{'a': 3, 3: 'x', 'b': 2, 'x': 666}
>>> 3 in d
True
>>> del d['b’] # removed_value = d.pop('b') # Removes 'b' and
returns its value
>>> d
{'a': 3, 'b': 2, 'x': 666}
>>> d.keys()
['a', 'b', 'x']
>>> d.items()
[('a', 3), ('b', 2), ('x', 666)]
>>> 'y' in d
False
```

Mutable vs. Immutable

- Mutable types allow changes to objects in memory
  - Examples: list, dictionary
- Immutable types do not
  - Examples: int, float, str, bool

```
>>> x = 42
>>> y = x
>>> x += 1
>>> print (x)
43
>>> print (y)  # ??
```

```
>>> x = []
>>> y = x
>>> x += [1]
>>> print (x)
[1]
>>> print (y)    # ??
```

Control Structures

if, elif, else

```
# Input: student's score
score = int(input("Enter the score: "))

# Determine the grade
if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
elif score >= 60:
    grade = "D"
else:
    grade = "F"

# Output the result
print(f"The grade is: {grade}")
```

for loop

```
# Loop through numbers 1 to 5
for num in range(1, 6): # outputs less than 2nd input
    print(f"The square of {num} is {num ** 2}")


The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
```

This uses f-strings (formatted string literals) to construct a dynamic message.

```
print("The square of " + str(num) + " is " + str(num ** 2))

print("The square of {} is {}".format(num, num ** 2))

print("The square of %d is %d" % (num, num ** 2))
```

while loop

```
# Initialize the counter
num = 1

# Loop while the condition is True
while num <= 5:
    print(f"Number: {num}")
    num += 1  # Increment the counter

Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

Sequences and Loops

```
for item in seq:   # list, tuple, str, set
    print (item)
sum = 0
for n in range(1, 11):
    sum += n
print (sum)
```

Dictionaries and Loops

```
for key in dict.keys():
    print ('%s => %s' % (key, dict[key]))
```

```
for pair in dict.items():
    print ('%s => %s' % pair)
```

```
for key, value in dict.items():
    print ('%s => %s' % (key, value))
```

Indentation

Indentation

- In Python, indentation is significant
- Whitespace is used to delimit program blocks
- Blocks are introduced with a colon “:”
- Each line within a basic block must be indented by the same amount

String Methods

String Methods

- String methods are built-in functions in Python that operate on string objects, allowing users to manipulate, format, or query strings easily.
- **Basic String Manipulation**

Method	Description	Example
str.upper()	Converts all characters to uppercase.	"hello".upper() → 'HELLO'
str.lower()	Converts all characters to lowercase.	"HELLO".lower() → 'hello'
str.capitalize()	Capitalizes the first character.	"python".capitalize() → 'Python'
str.title()	Capitalizes the first letter of each word.	"hello world".title() → 'Hello World'
str.strip()	Removes leading/trailing whitespace.	" hello ".strip() → 'hello'
str.lstrip()	Removes leading whitespace.	" hello".lstrip() → 'hello'
str.rstrip()	Removes trailing whitespace.	"hello ".rstrip() → 'hello'
str.replace(old, new)	Replaces all occurrences of a substring.	"banana".replace('a', 'o') → 'bonono'

String Methods

- **Querying a string**

Method	Description	Example
str.startswith(prefix)	Checks if the string starts with the given prefix.	"hello".startswith('he') → True
str.endswith(suffix)	Checks if the string ends with the given suffix.	"hello".endswith('lo') → True
str.find(sub)	Returns the index of the first occurrence of a substring, or -1.	"apple".find('p') → 1
str.index(sub)	Like find() but raises a ValueError if not found.	"apple".index('p') → 1
str.count(sub)	Counts occurrences of a substring.	"banana".count('a') → 3

- **String Splitting and Joining**

Method	Description	Example
str.split()	Splits the string into a list of words (default delimiter: space). From left.	"a,b,c".split(',') → ['a', 'b', 'c']
str.rsplit()	Splits the string from the right into a list. When no maxsplit is specified, both split() and rsplit() produce the same result. Maximum number of splits to perform. If not specified or set to -1.	"a b c".rsplit(maxsplit=1) → ['a b', 'c']
str.join(iterable)	Joins elements of an iterable with the string as a separator.	",".join(['a', 'b', 'c']) → 'a,b,c'

String Vs Integers

- '5’ isn’t equal to 5
- Conversions

```
>>> int('5')
5
>>> float('5')
5.0
>>> str(5)
'5'
>>> int(5.5)
5
>>> float(5)
5.0
```

While vs for

```
password = ""
while password != "secret":
    password = input("Enter password: ")


import random
x = 0
while x < 0.9:
    x = random.random()
    print(x)
```

Strings

- Single quotes and double quotes work same way
- Escape characters : '\n', '\t', etc.

```
>>>a='foo\tbar'
>>> a
'foo\tbar'
>>> print(a)
foo bar
```

Comments & Line Continuations

- Comments are started with “#”
- Long lines can be continued by ending with a backslash “\”
- Multiline Strings- triple quotes

```
>>> a = "Nick " + \
... "LeRoy"
>>>
>>> a
'Nick LeRoy’
>>> multiline1 = '''This is
a multiline
string.'''
>>>  multiline1
'This is\na multiline\nstring.’
>>> print(multiline1)
This is
a multiline
string.
```

```
# This is a comment
a = 5 # This is too
```

Inputs

Feature	raw_input() (Python 2)	input() (Python 2)	input() (Python 3)
Reads as String	Yes	No (evaluates input)	Yes
Evaluates Input	No	Yes	No
Explicit Conversion	Not required for strings	Required for non-strings	Required for non-strings

raw_input removed in Python 3

```
>>> 'Python' > 'C++'
True
>>> 'Python' > 'Z'
False
```

Identity: To get address

```
>>> a = 'abcdef'
>>> b = 'abc'
>>> b += 'def'
>>> a is b
False
>>> id(a), id(b)
(140239763722624, 140239763722864)
```

Operator Precedence Table

s	Operator	Description	Associativity
1 (Highest)	()	Parentheses (grouping)	N/A (evaluated first)
2	**	Exponentiation	Right-to-left
3	+x, -x, ~x	Unary operators: positive, negative, bitwise NOT	Right-to-left
4	*, /, //, %	Multiplication, division, floor division, modulus	Left-to-right
5	+, -	Addition, subtraction	Left-to-right
6	<<, >>	Bitwise shift operators	Left-to-right
7	&	Bitwise AND	Left-to-right
8	^	Bitwise XOR	Left-to-right
9	|	Bitwise OR	Bitwise OR
10	==, !=, >, <, >=, <=, is, is not, in, not in	Comparisons and membership tests	Left-to-right
11	not	Logical NOT	Right-to-left
12	and	Logical AND	Left-to-right
13 (Lowest)	or	Logical OR	Left-to-right

Examples

```
result = 2 ** 3 ** 2  # Equivalent to 2 ** (3 ** 2)
result = -3 + 5  # Unary `-` applied first
result = True or False and False
# `and` is evaluated first, so it becomes True or (False and False) → True
result = 10 - 5 - 2  # Equivalent to (10 - 5) - 2
```

Examples

Operator	Example	Explanation	Output
Parentheses	result = (2 + 3) * 4	Grouping ensures 2 + 3 is evaluated first.	20
Exponentiation	result = 2 ** 3 ** 2	Exponentiation is right-to-left, so 3 ** 2 first, then 2 ** 9.	512
Unary +, -, ~	result = -3 + ~2	Unary - makes -3, ~2 is bitwise NOT of 2. ~2 = -(2 + 1) = -3	-3 + (-3) = -6
Multiplication/Division/Floor Division/Modulus	result = 10 % 3 * 2 // 2	10 % 3 → 1, 1 * 2 → 2, then 2 // 2.	1
Addition/Subtraction	result = 5 + 3 - 2	Left-to-right: 5 + 3 = 8, then 8 - 2.	6
Bitwise Shifts	result = 4 << 1 >> 1	4 << 1 → 8, then 8 >> 1.	4
Bitwise AND	result = 5 & 3	5 in binary 101, 3 in binary 011.	1
Bitwise XOR	result = 5 ^ 3	Binary XOR: 101 ^ 011 → 110.	6
Bitwise OR	result = 5 | 3	3`	7
Comparison	result = 5 > 3 == True	5 > 3 → True	True
Membership	result = 'a' in 'abc'	'a' is present in 'abc'.	True
Identity	result = 5 is 5.0	5 and 5.0 have different types.	False
Logical NOT	result = not False	Negates False → True.	True
Logical AND	result = True and False	Both must be True → False.	False
Logical OR	result = True or False	Only one needs to be True → True.	True

break & continue

- To break out of a loop in the middle:
- *break*
- To go back to the top of a loop:
- *continue*

```
i = 0
while True :
    i += 1
    if i == 5 :
        continue
        print (i)
    if i > 10 :
        break
```

range function

- range(n) is a built-in function
- Returns a list of integers
- range(5) will return the list : (0,1,2,3,4)
- range(m,n):
- Returns a list of integers
- range(1,5) will return the list : (1,2,3,4)

```
for i in range(10) :
    if i == 5 :
        continue
    print (i)
```

```
>>> print(range(1,5) )
range(1, 5)
>>> a=range(10)
>>> a
range(0, 10)
>>> list(a)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- This happens because the range() function creates a lazy, iterable object that only generates the numbers on demand, rather than storing them explicitly in memory.

Examples

```
>>> babylon5 = ["Sheridan","G'Kar","G'Kar","Delenn"]
>>> babylon5[1]
"G'Kar"
>>> babylon5[-1]
'Delenn’
>>> babylon5.index("G'Kar")
1
>>> babylon5[2]='Zathras'
>>> babylon5[4]='Zathras'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Slices

- sequence[start:stop:step]
- start: The index to start the slice (inclusive). Defaults to 0 if omitted.
- stop: The index to end the slice (exclusive). Defaults to the length of the sequence if omitted.
- step: The increment (step) between each index. Defaults to 1 if omitted.

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


# Basic slicing
print(lst[2:6])      # Output: [2, 3, 4, 5] (indexes 2 to 5)
print(lst[:4])       # Output: [0, 1, 2, 3] (start defaults to 0)
print(lst[5:])       # Output: [5, 6, 7, 8, 9] (stop defaults to end)


# Negative indexing
print(lst[-5:-2])    # Output: [5, 6, 7]
print(lst[-5:])      # Output: [5, 6, 7, 8, 9]


# Using steps
print(lst[::3])      # Output: [0, 3, 6, 9] (every third element)
print(lst[::-2])     # Output: [9, 7, 5, 3, 1] (reverse with step -2)
```

Functions

Functions

- Allow for code reuse
  - Same block of code can be used from several different places
- Central to procedural and object-oriented programming
- Sometimes required
  - i.e. for callbacks, etc. (In GUIs etc.)
- You've already used a lot of them:
  - int(), len(), float(), etc.

Functions

```
def process_data(data, callback):

    # Perform some processing on data

    processed_data = data.upper()  # Convert data to uppercase

    # Call the callback function with the processed data

    callback(processed_data)


# A callback function to be executed

def display_result(result):

    print(f"Processed Data: {result}")


# Example data

input_data = "hello world"


# Call process_data and pass the callback function

process_data(input_data, display_result)


Processed Data: HELLO WORLD
```

Default Values

- Parameters can have default values
- Parameters can be passed by name

```
#! /usr/bin/env python
"""Example"""
def MyFunc( a, b='xyzzy', c=None ) :
    """Takes an 1, 2 or 3 parameters"""
    s = None
    if c is None :
        s = "a=%d b=%s" % ( a, str(b) )
    else :
        s = "a=%d b=%s c=%s" % ( a, str(b), c )
    if isinstance( b, int ) :
        s += " (b is an int)"
    return s
# Call the function in various ways
print (MyFunc( 1 ))
print (MyFunc( 1, 'foo' ))
print (MyFunc( 1, 5, 'fizbin' ))
print (MyFunc( 1, c='plugh' )) # Note: "c" is specified by name!

a=1 b=xyzzy
a=1 b=foo
a=1 b=5 c=fizbin (b is an int)
a=1 b=xyzzy c=plugh
```

Returning Multiple Values

```
#! /usr/bin/env python

"""Example"""

def MyFunc( num=None ) :
       """Returns a string, and an int (or None)"""
       sval = input("Enter a string: " )
       if num is not None :
              ival = num
       else :
              try :
                            ival = int(input("Enter an integer: ") )
              except ValueError :
                            ival = None
       return sval, ival

for v in ( None, 1, 3 ) :
       s, i = MyFunc( v )
       if i is None :
              i = 999
       print "i=%d s=%s" % ( i, s )
```

```
 Enter a string: 123

 Enter an integer: asd

 i=999 s=123

 Enter a string: def

 i=1 s=def

 Enter a string: ags

 i=3 s=ags
```

Help

Built-In Help I

dir(object or type)

- Lists all operations for that object or type
- For now, ignore everything that starts with __
- Use as object.operation(…)

```
dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__',
'__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha',
'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill’]
```

Built-In Help II

help(something)

- Shows built-in documentation
- Works on objects, types, and their operations

```
help(str.lower)
Help on method_descriptor:

lower(self, /) unbound builtins.str method
    Return a copy of the string converted to lowercase.
help(lower)
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    help(lower)
NameError: name 'lower' is not defined
```

Handling Exceptions & other

Handling Exceptions

```
try:

    num = int(input("Enter a number: "))

    result = 10 / num

    print(f"The result is {result}")

#Raised when the input cannot be converted to an integer.

except ValueError:

    print("Invalid input! Please enter a valid integer.")

#Raised when attempting to divide by zero

except ZeroDivisionError:

    print("Division by zero is not allowed.")

# Executes if no exception occurs in the try block

else:

    print("No exceptions occurred. The operation was successful.")

#Executes regardless of whether an exception occurred or not. Often used for cleanup
(e.g., closing files or releasing resources).

finally:

    print("Execution completed.")
```

assert

- Used primarily as a convenient way to insert debugging assertions into a program: assert <expression>
- If <expression> evaluates to 0 (zero) or False, an AssertionError exception is raised.

```
def my_func( a, b ) :
    assert isinstance( a, int )
    assert b is not None
    ...
```

Functions: Arguments Are Local Variables

- The function can't modify the caller’s passed-in variable

```
def MyFunc( a ) :
    """Paramter a is a local variable, thus it's changes to
    the variable don't affect the caller's copy"""
    a += 10
    print ("MyFunc: a =", a)
    return a
# Call the function in various ways
b = MyFunc( 1 )
print ("main: b =", b)
b = 10
MyFunc( b )
print ("main: b =", b)
```

```
MyFunc: a = 11
main: b = 11
MyFunc: a = 20
main: b = 10
```

Function can access global variable

- The function can't modify the caller’s passed-in variable
- Assignments to globals don't work (scooping)
- Use the keyword global to modify
- A function can even have local functions

```
foo = 5
def SomeFunc( ) :
    print ("SomeFunc: foo is", foo)
    foo=10 # Scooping
SomeFunc( )
print ("main: foo is", foo)
```

```
SomeFunc: foo is 5
main: foo is 5
```

File Handling

Read a file

- Read the Entire File

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

- Read File Line by Line

```
with open("example.txt", "r") as file:
    for line in file:
        print(line.strip())  # strip() to remove extra
spaces or newline characters.
```

- Read All Lines into a List

```
with open("example.txt", "r") as file:
    lines = file.readlines()
    print(lines)
```

Read a file with exceptions

```
try:
    # Attempt to open and read the file
    with open("example.txt", "r") as file:
        content = file.read()
        print(content)
except FileNotFoundError:
    # Handle case where the file doesn't exist
    print("Error: The file 'example.txt' was not found.")
except PermissionError:
    # Handle case where the file cannot be accessed
    print("Error: You do not have permission to read the
file.")
except Exception as e:
    # Catch any other unexpected exceptions
    print(f"An unexpected error occurred: {e}")
finally:
    # Optional: Code to execute no matter what happens
    print("File read attempt completed.")
```

Writing in a file

- Writing to a File Using write()

```
with open("example.txt", "w") as file:
    file.write("This is the first line.\n")
    file.write("This is the second line.\n")
```

- Appending to a File Using write()

```
with open("example.txt", "a") as file:
    file.write("This is an appended line.\n")
```

- Writing Multiple Lines Using writelines()

```
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
with open("example.txt", "w") as file:
    file.writelines(lines)
```

Writing in a file with exceptions

```
try:
    with open("example.txt", "w") as file:
        file.write("This is safe writing.\n")
except Exception as e:
    print(f"An error occurred: {e}")
```

```
try:
    # Attempt to open and write to the file
    with open("example.txt", "w") as file:
        file.write("This is the first line.\n")
        file.write("This is the second line.\n")
    print("File written successfully.")
except FileNotFoundError:
    # Raised if the file path is invalid or inaccessible
    print("Error: The file path does not exist.")
except PermissionError:
    # Raised if the program lacks permission to write to the file
    print("Error: You do not have the necessary permissions to write to 
this file.")
except Exception as e:
    # Handles any other exceptions
    print(f"An unexpected error occurred: {e}")
finally:
    # Code in the `finally` block executes no matter what happens
    print("Write operation completed.")
```

Python Modules

Python Modules

- Python has a large collection of standard modules
  - New types, parsers, etc.
  - os, sys, copy, optparse
- There are also a lot of 3rd party modules available
  - NumPy (numerical operations)
  - Beautiful Soup (HTML parser)

Using a module

- To use a module, use the “import” keyword
- import <module>
  - Imports all symbols from the module into it's own namespace
- from <module> import a,b
  - Imports only symbols “a” and “b” from the module into the local namespace
- from <module> import *
  - Import all symbols from the module into the local namespace

Copying a list

```
>>> import copy
>>> a=[1, 2, 3, 4]
>>> b = copy.copy(a)
>>> a is b
False
>>> a.append(5)
>>> a
[1, 2, 3, 4, 5]
>>> b
[1, 2, 3, 4]
>>> from copy import copy
>>> b = copy(a)
```

lambda

- lambda is a way to create anonymous functions (functions without a name) in a concise manner
- lambda arguments: expression

```
# Lambda function to add two numbers
add = lambda x, y: x + y
print(add(3, 4))  # Output: 7
```

- lambda can be used with map, filter, sorted, conditional, etc.

```
# Lambda function for a conditional operation
greater_than_five = lambda x: "Greater" if x > 5 else
"Lesser"
print(greater_than_five(7))  # Output: Greater
print(greater_than_five(3))  # Output: Lesser
```

map

- map() function applies a given function to all items in an iterable and returns a map object (which is an iterator) containing the results.
- map(function, iterable, ...)

```
numbers = [1, 2, 3, 4, 5]

# Applying a function to square each number
def square(x):
    return x ** 2

result = map(square, numbers)
print(list(result))  # Output: [1, 4, 9, 16, 25]

# Using a lambda function to double each number
result = map(lambda x: x * 2, numbers)
print(list(result))  # Output: [2, 4, 6, 8, 10]
# Adding corresponding elements from two lists
result = map(lambda x, y: x + y, a, b)
print(list(result))  # Output: [5, 7, 9]
```

map

```
a = [1, 2, 3]

# Applying two functions: square and double
def square(x):
    return x ** 2

def double(x):
    return x * 2

result = map(lambda x: double(square(x)), a)
print(list(result))  # Output: [2, 8, 18]
```

Array flattening

- Python does not flatten arrays

```
>>> x = [1,2,3]
>>> [5,x,6,x]
[5, [1, 2, 3], 6, [1, 2, 3]]
```

- To concatenate elements of a list in another list, use the + or +=

```
>>> x=[ 1, 2, 3]
>>> [ 5] + x + [ 6 ] + x
[5, 1, 2, 3, 6, 1, 2, 3]
```

Array concatenation

- join() is used to concatenate elements using delimiter.join(iterable)

```
words = ["Hello", "World", "Python"]
result = " ".join(words) # Joining with a space
print(result)  # Output: "Hello World Python"
```

- join() method does not work if the iterable contains non-string elements. Convert to string first.

```
numbers = [1, 2, 3, 4, 5]
# Convert each number to a string and join with a space
result = " ".join(map(str, numbers))
print(result)  # Output: "1 2 3 4 5"
```

Functions vs Methods

Feature	Function	Method
Definition	Defined using def or lambda	Defined inside a class
Call	Can be called directly (e.g., greet())	Called on an instance or class (e.g., obj.greet())
Binding	Not bound to any object	Always bound to an object or class
First Parameter	No special parameter (can be any argument)	Usually self (for instance) or cls (for class methods)
Access to Object	Does not have access to object attributes	Can access and modify object attributes via self

‘is’ for object identity not value

```
a='abc'
b=a
id(a) is id(b)
False
a is b
True
id(a)
4330969600
id(b)
4330969600
id(a) == id(b)
True

# As id(a) is
value and not
object
```

```
a = 'abcdef'
b='abcdef’ # same memory as a is used
(string and integer specific)
id(a)
4311225744
id(b)
4311225744
a is b # not reliable
True
b='abc'
b+='def’ # New memory created on runtime
b
'abcdef'
a is b
False
```

is

```
a = [1, 2, 3]
b = [1, 2, 3]

a == b   # True  (same value)
a is b   # False (different objects)
```

When to use ‘is’

- To check None: x is None better than x==None
- Check if True is True, etc., for singletons: True, None, and False. They have a single memory.

```
id(True)
4330737016
a=10
id(a==10)
4330737016
```

- Check object identity a=b; a is b

Thank You

-
- LeRoy & De Smet notes