PROGRESS TRACKING

CHAPTER 1 : PYTHON BASICS

1. When using python in interactive mode : '_' variable holds the result of the last operation.
2. To end program use quit() or EOF, in macos/UNIX ctrl+D is EOF

> **It is common to use #! to specify the interpreter on the first line of a program**

```
#!/Users/ayushdutta/.pyenv/shims/python
```

- We can use a fstring inside a print() :

```
print(f'{year:>3d} {principal:0.2f}')
```

Here the :>3d means a 3 decimal number, and 0.2f means round of to 2 deciman accuracy.

> **NOTE : The round() function inplements "baker's rounding" .If the value being rounded is equally close to two multiples, it is rounded to the nearest even.**

> **NOTE : Python does not have ++ or -- operator.**

Walrus Operator : Sometimes, you may see the assignment of a variable and a conditional combined together using the := operator.

```
x = 0
while((x := x+1) < 10):
    print(x)                # prints 1, 2, 3, 4, 5, ... , 9
```

- The `break` statement just breaks the innermost loop.
- The `continue` statement skips the rest of the loop body.
- `''' xyz '''` is multiline comment, which captures all string in between

1.1 Text Strings :

- `fstrings**` :

```
base_year = 2020,
print(f'{base_year + year:>4d} {principal:0.2f}')
```

- `format()` and `%` operators are also used alternative to fstring

```
print('{0:>3d} {1:0.2f}'.format(year, principal))
print('%3d %0.2f' % (year, principal))
```

- for strings, `string[-1]` refers to the last index
- **String Slicing** : `string[1:5]` from 1st index to 4th.
- `.replace('hello', 'hello2')` function replaces all the occurance.
- **Common String Methods :**

Method	Description
`s.endswith(prefix[,start [,end]])`	string ends with prefix
`s.find(sub [, start[,end]])`	Finds the first occurrence of the specified substring sub or -1 if not found.
`s.replace(old, new[,maxreplace])`	Replaces a substring, n times
`s.split([sep[,maxsplit]])`	splits string into maxsplit number of splits
`s.strip([chrs])`	Removes whitespaces but removes chars if chrs is given

- **Non-string values can be converted into a string representation by using the str(), repr(), or format() functions.**

```
s = 'hello\nworld'
str(s)
repr(s)
x = 10.235
format(x, "0.2f")
f'{x:0.2f}'
```

output :

> hello
> world
> hello\nworld
> "10.24"
> "10.24"

1.2. FILE IO

open(FILE) returns a `file oject`

Reading Files

```
with open('data.txt') as file:
  for line in file:
    print(line, end='')
file.close()
```

The `with` statement opens the object as file. file only valid in context.

```
with open('data.txt') as file:
  data = file.read()
```

Reads the whole file at once.

```
with open('data.txt', encoding='utf-8') as file:
  while (chunk := file.read(10000)):
    print(chunk, end='')
```

Reads file in chunks.

Writing Files

```
with open('data.txt', 'wt') as fp:
  while year <= numyears:
    principal = principal * (1+rate)
    print(f'{year:3d} {principal:0.2f}', file=fp)
    year += 1
```

OR

```
fp.write(f'{year:3d} {principal:0.2f}\n')  #NOTE the \n
```

Input from Console

```
name = input("What is your name")
print("hello", name)
```

1.3. Lists

```
names = ['Dave', 'Paula', 'Thomas', 'Lewis']
last_index = names[-1]

# splits the letters into a list
letters = list('Dave')

# Multi Dimentional list
a = [1, 'Dave', 3.14, ['Mark', 7, 9, [100, 101]], 10]
value_100 = a[3][3][0]
```

Sample FILE IO Code

```
# TODO
# Fileformat : Name,QTY,PRICE
# 1. read filename from argv
# 2. Find total price  accross all rows

import sys

if len(sys.argv) != 2:
    raise SystemExit(f'Usage: {sys.argv[0]} filename')

rows = []

with open(sys.argv[1], "rt") as file:
    for line in file:
        rows.append(line.split(',')[1:])

total = sum([int(row[0]) * float(row[1])  for row in rows])

print(f"SUM : {total}")
```

1.4. Tuples

`immutable`

To create simple data structures, you can pack a collection of values into an immutable object known as a tuple

Tuples can do indexing, slicing, concat, etc but they can not be modified once created.

```
# File containing lines of the form "name,shares,price"

filename = 'portfolio.py.txt'

portfolio = []

with open(filename) as file:
    for line in file:
        row = line.split(',')
        name = row[0]
        shares = int(row[1])
        price = float(row[2])

        holding = (name, shares, price)
        portfolio.append(holding)

# iterating over each holding
total = 0
for _, shares, price in portfolio:
    total += shares * price;

print(f"Total portfolio : {total}")
```

1.5. Sets

`immutable`

The elements of a set are typically restricted to **immutable** objects. For example, you can make a set of numbers, strings, or tuples.

> **Unlike lists and tuples, sets are unordered and cannot be indexed by numbers. They must have unique values**

```
print(portfolio)
s = { sets[0] for sets in portfolio }
```

```
[('GOOG', 100, 490.0), ('FB', 88, 542.0), ('ALPH', 33, 23.0), ('GOG', 18, 45.0)]
{'FB', 'ALPH', 'GOOG', 'GOG'}
```

Operations	Explanation
a = t | s	Union
b = t & s	Intersection
c = t - s	Difference
d = t ^ s	Symmetric Difference

Symmetric Difference - items that are in either s or t but not in both.

Operation	Explanation
a.add('Hello')	Single item add
a.update('hello', 'hi', 'yo')	Add multiple items
t.remove('IBM')	Remove 'IBM' or raise KeyError if absent.
s.discard('SCOX')	Remove 'SCOX' if it exists.

1.6. Dictionaries

ref code = dicts.py

`d = dict()` #empty dict

```
s = {
    'name' : 'GOOG',
    'shares' : 100,
    'price' : 490.10,
    'S&P500' : False
}

print(s['names])
if s.get('S&P500') is False:
    s['S&P500'] = True

print(s)

if s.get('KEY_NOT_PRESENT') is None:
    print("Key Not found)

#or

print(s.get('KEY_NOT_PRESENT', "Key not found"))
# it will print "key not found" if no keys found or else it will print the value
```

```
'GOOG'
{'name' : 'GOOG', 'shares' : 100, 'price' : 490.10 'S&P200' : true}
```

Iterating over a dict:

```
dict_a = {'name':'Ayush', 'age':21}
for key, value in dict_a.items():
    print(key, value)
```

Using tuples with dicts

```
prices = { }
prices[('IBM', '2015-02-03')] = 91.23
prices['IBM', '2015-02-04'] = 91.42     # Parens omitted

> {('IBM', '2015-02-04'): 91.42}
```

> **NOTE - Any kind of object can be placed into a dictionary, including other dictionaries. However, mutable data structures such as lists, sets, and dictionaries cannot be used as keys.**

Example Dict Code

```
portfolio = [
   ('ACME', 50, 92.34),
   ('IBM', 75, 102.25),
   ('PHP', 40, 74.50),
   ('IBM', 50, 124.75)
]
total_shares = { s[0]: 0 for s in portfolio }

for name, shares, _ in portfolio:
    total_shares[name] += shares

# total_shares = {'IBM': 125, 'ACME': 50, 'PHP': 40}
```

We can do the same thing without initializing the total_share with zeros and keys. We use the `counter() function`
Basically counter generates a Counter() object with `Keys:number` items.

```
from collections import Counter()

total_shares = Counter()

for name, shares, _ in portfolio:
    total_shares[name] += shares

# total_share = Counter({'IBM': 125, 'ACME': 50, 'PHP': 40})
```

Common Functions

```
pairs = [('IBM', 125), ('ACME', 50), ('PHP', 40)]
d = dict()  # -> creats a dicts
print(list(d))  # -> creats a list of 'keys'
d.pairs() # -> creates a “keys view” object that is attached to the dict
```

1.7 Exceptions

We use `try` `except` to handle errors

```
try :
  ...
except ValueError as err:
  ...
```

also

```
raise RuntimeError('Computer says no')
```

```
raise SystemExit("EXIT") # Exit program with no error
```

1.8 Object and Classes

All values used in a program are objects. An object consists of internal data and methods that perform various kinds of operations involving that data.

The `class` statement is used to define new types of objects and for object-oriented programming.

ref : class_test.py

```python
class Stack:

    def __init__(self) -> None:
        self._items = []

    def push(self, item):
        self._items.append(item)

    def pop(self):
        self._items.pop()

    def __len__(self):
        return len(self._items)

    def __repr__(self):
        return f'<{type(self).__name__} at 0x{id(self):x}, size={len(self)}>'

    def print(self):
        print(self._items)

s = Stack()
s.push('ONE')
s.pop()
for item in ['ONE', 'TWO']:
    s.push(item)
s.push('THREE')
s.print()

print(f'len = {len(s)}')
print(s)
```

`self` -> The term “self” refers to the instance of the class that is currently being used. Its always the first argument. **Pointer to Current Object.**

`__init__(self, ...)` -> Its the class constructor. Gets called when instance of class is created. It can be used to initialize the attributes.

`_method` are meant to be private methods. Its a convention. Not to be used outside the Class.

`__method__` are special method. eg :. `__len__` gets called when `len(obj)` is called. `__repr__` changes the way how the object is displayed and printed.

Inheritance

Now we can create a Class that inherits all attributes and methods of another class.

```
class MyStack(Stack):
    def print_len(self):
        return "len : " + str(len(self._items))

s1 = MyStack()
s1.push('ONE')  # -> calling push() from Stack class
s1.push('TWO')  # -> calling push() from Stack class
print(s1.print_len())  # -> calling from MyStack class
s1.print()
```

We can also change the behaviour of an existing method in the inherited class. For ex: we can make the stack to accept only numbers.

```
class MyStack(Stack):
    def print_len(self):
        return "len : " + str(len(self._items))

    ## CHANGING METHOD behaviour of Parent class
    def push(self, item):
        if not isinstance(item, (int, float)):
            raise TypeError('Expected an integer value')
        super().push(item)

s1 = MyStack()
# s1.push('ONE') -> ERROR
s1.push(1)
print(s1.print_len())
s1.print()

#  len : 1
# [1]
```

1.9 Modules

As your programs grow in size, you will want to break them into multiple files for easier maintenance. To do this, use the import statement. To create a module, put the relevant statements and definitions into a file with a .py suffix and the same name as the module.

check : portfolio_package/portfolio_value_calculator.py

1.10 Script Writing, `__name__ == __main__`

`__name__ = name of program` : if imported as module
`__name__ = __main__` : if ran as standalone program

now we can use `if __name__ == __main__ :` to make the program run standalone in a proper way.

check : portfolio_package/readport.py

1.11 Packages

```
portfolio_package/
    __init__.py
    readport.py
    portfolio_value_calculator.py    (imports readport.py)
```

To run the portfolio_value_calculator.py, we have to go out of this package and run it :
`python -m portfolio_package.portfolio_value_calculator`

Python: It Fits Your Brain

CHAPTER 2 : OPERATORS, EXPRESSIONS, and DATA MANUPULATIONS

Number	Base
42	Decimal
0b101010	Binary Number

Number	Base
0o52	Octal Number
0x2a	Hexadecimal

> Base is not stores as part of the integer. All of the above literals will display as 42. To print number with base use `bin(x), oct(x), hex(x)` .

- `Floating Point number` - 4.2e+2 : Stored as **IEEE 754 double- precision (64-bit) values**.
- We can't include the assignment operator as part of an expression:

```
while line=file.readline(): # Syntax Error.
    print(line)

while (line:=file.readline()): # Using the wallrus operator
    print(line)
```

- Reference to lists (pointers in python). Both are refering to the same list at same memory location.

```
a = [1, 2, 3]
b = a
a += [4, 5]

print(a)    # -> [1, 2, 3, 4, 5]
print(b)    # -> [1, 2, 3, 4, 5]
```

1. Object Comparison :

`x is y` checks if `id(x) == id(y)` . Test two values to see whether they refer to literally the same object in memory. `x == y` checks if the containts of the objects are equal and not the object itself.

2. Ordered Comparison Operators :

We can use the comaprison operators on lists, tuples, and strings. It compares each index to the corresponding index. As soon as the furst one greater or smaller is encountered, it decides the total outcome.

- `[1, 2, 3] < [3, 4, 5]` also `[3, 4, 5, 6, 9, 9] < [1, 2, 4]`
  For sets, x < y tests if x is strict subset of y (i.e., has fewer elements, but is not equal to y).
- For sets it checks strict subsets as sets values are unique.

3. Conditional Expressions :

```
minvalue = a if a <= b else b
```

4. Operations on Iterables :

Description	Operations
`for vars in s:`	Iteration
`a, b, c ... in s:`	Variable unpacking
`x in s` , `x not in s`	Membership
`[a, *s, b], {a, *s, b}`	Expansion in list, tuples or sets

Example 1: Unpacking list and creating dict

```
items = [3, 4, 5]
d={ }
d['x'], d['y'], d['z'] = items

# {'x' : 3, 'y' : 4, 'z' : 5}
```

> **NOTE : When unpacking values into locations, the number of locations on the left must exactly match the number of items in the iterable on the right.**

```
datetime = ((5, 19, 2008), (10, 30, "am"))
(month, day, year), (hour, minute, am_pm) = datetime
(month, _, year), (hour, _, am_pm) = datetime    # _ is throwaway variable
```

Example 2 : If the number of items being unpacked isn't known use `*extras`

```python
items = [1, 2, 3, 4]
a, *extras, b = items    # a=1, *extras=[2, 3], b=4

datetime = ((5, 19, 2008), (10, 30, "am"))
(month, *_), (hour, *_) = daytime
```

Any iterable can be expanded when writing out list, tuple, and set literals usinf star (*) operator.

```python
items = [1, 2, 3, 4]
a = (10, 11, *items, 12)  # (10, 11, 1, 2, 3, 4, 12)
b = [a, *items, b] # [a, 1, 2, 3, 4, b] and not [a, [1, 2, 3, 4], b]
```

> NOTE : However, many iterable objects (such as files or generators) only support one-time iteration. If you use *-expansion, the contents will be consumed and the iterable won't produce any more values on subsequent iterations.

5. Operations on Sequences :

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

a[2:5]      # [2, 3, 4]
a[:3]       # [0, 1, 2]
a[-3:]      # [7, 8, 9]
a[::2]      # [0, 2, 4, 6, 8]
a[::-2]     # [9, 7, 5, 3, 1] , if stride is -ve initial is from end if i is omited
a[0:5:2]    # [0, 2, 4]
a[5:0:-2]   # [5, 3 ,1]
a[:5:1]     # [0, 1, 2, 3, 4]
a[:5:-1]    # [9, 8, 7, 6]
a[5::1]     # [5, 6, 7, 8, 9]
a[5::-1]    # [5, 4, 3, 2, 1, 0]
a[5:0:-1]   # [5, 4, 3, 2, 1]
```

Slices can be named using `slice()` . For example:

```python
firstfive = slice(0, 5)
s = 'hello world'
print(s[firstfive])     # Prints 'hello'
```

6. Operations on Mutable Sequences :

All the same as above and insertion and deletion also allowed.

```
a = [1, 2, 3, 4, 5]
a[3:4] = [-1, -2, -3]    # a = [1, 6, 10, -1, -2, -3, 5]
a[2:] = [0]     # a = [1, 6, 0]

a = [1, 2, 3, 4, 5]
a[1::2] = [10, 11]      # a = [1, 10, 3, 11, 5]
# Have to suply exact number of items in the right that matches the output on the left
del a[1:]  # [1]
```

> NOTE : The popular numpy package has different slicing semantics than Python lists. Same goes with third party packages.

7. Operations on Sets

Operations	Explaination
a = t | s	Union
b = t & s	Intersection
c = t - s	Difference
d = t ^ s	Symetric Difference

Symetric Difference - items that are in either s or t but not in both.

> Also works on keys of dicts.
> eg : `a.keys() & b.keys()`

8. Operations on Mappings

Operation	Description
`a = m['key']`	Inexing
`del m['key']`	Deleting by key

Operation	Description
'key' in m	Membership
`m.keys()`	Returns the Keys
`m.values()`	Returns the Values
`m.items()`	returns (key, value) pair

example : tuple as key (immutble)

```
m[('ayush', 'dutta', '11-07-2002')] = 'Computer Science'
m[('lugen marie','dutta', '11-28-2003')] = 'English'
```

9. List, Set & Dict Comprehensions

Find all entries that have more than 100 shares

```
# Some data (a list of dictionaries)
portfolio = [
  {'name': 'IBM', 'shares': 100, 'price': 91.1 },
  {'name': 'MSFT', 'shares': 50, 'price': 45.67 },
  {'name': 'HPE', 'shares': 75, 'price': 34.51 },
  {'name': 'CAT', 'shares': 60, 'price': 67.89 },
  {'name': 'IBM', 'shares': 200, 'price': 95.25 }
]

more100 = [s["name"] for s in portfolio if s["shares"] > 100]

# Use of walrus operator, and creating a set of tuples
more100_set = {(s["name"], v) for s in portfolio if (v := s["shares"]) >= 100}
print(more100_set)
```

> NOTE : typechecking while comprehension does not exist. So use seperate function. eg : comprehension.py

10. Generator Expressions

A generator expression is an object that carries out the same computation as a list comprehension but produces the result iteratively. It gives you a generator object that we can iterate on demand.

```
nums = [1, 4, 9, 16]
squares = (i*i for i in values)

>>> squares
<generator object at 0x590a8>
>>> next(squares)
1
>>> next(squares)
4
...
>>> for n in squares:
...     print(n)
9
16
>>>
```

> **NOTE : A generator expression can only be used once. If you try to iterate a second time, you'll get nothing.**

Look at generators.py for implementation using FILE IO.

CHAPTER 3: Program Structure and Control Flow

3.1 Loops & Iterations

1. `enumerate` -

```python
values = [1, 2, 3, 4, 5]

for index, i in enumerate(values, start=100):
    print((index, i))

# (100, 1)
# (101, 2)
# (102, 3)
# (103, 4)
# (104, 5)
```

2. `zip` -

```python
x = [1, 2, 3, 4, 5]
y = ["a", "b", "c", "d", "e"]

for i in (zip_x_y := zip(x, y)):
    print(i)

# (1, 'a')
# (2, 'b')
# (3, 'c')
# (4, 'd')
# (5, 'e')
```

3. `for else` loop structure

The else runs only if the loop is complleted on its own without breaking at any moment.

```python
values = [1, 2, 3, 4, 6, 7, 8, 9, 6, 12, 18, 19]

for i in values:
    if i % 5 == 0:
        print("Number div by 5 found")
        break

else:
    print("No number div by 5")
```

> **NOTE : to break out of a deeply nested loop raise an exception**

3.2 Exceptions

check : exceptions.py

As a matter of programming style, you should only catch exceptions from which your code can actually recover.

Exceptions can be use with `as` var. And one can use the `a.args` which is a tuple

```
try:
    file = open("foo.txt", "r", encoding="UTF-8")

except FileNotFoundError as e:
    print(e.args)

# (2, 'No such file or directory')
```

It also supports `try else` statements. The Else executes if there is no exception raised in try block.

```
try:
    file = open("./file.txt", "r", encoding="UTF-8")

except FileNotFoundError as e:
    print(f"Unable to open file : {e}")
    DATA = ""

else:
    DATA = file.read()
    print(DATA)
```

3.2.1 Exception Hierarcy

For `IndexError` or `KeyError` we can use `LookupError` ro handle both. Both of them inherit from `LookupError`.

Exception Catagories :

Exception Class	Description
`BaseException`	The root class for all exceptions
`Exception`	Base class for all program-related errors
`ArithmeticError`	Base class for all math-related errors
`ImportError`	Base class for import-related errors
`LookupError`	Base class for all container lookup errors
`OSError`	Base class for all system-related errors
`ValueError`	Base class for value-related errors, including Unicode
`UnicodeError`	Base class for a Unicode string encoding-related errors

Other exceptions which aren't part of a larger exception group :

Exception Class	Description
`AssertionError`	Failed assert statement
`AttributeError`	Bad attribute lookup on an object
`EOFError`	End of File
`MemoryError`	Recoverable out-of-memory error
`NameError`	Name not found in the local or global namespace
`NotImplementedError`	Unimplemented feature
`RuntimeError`	A generic "something bad happened" error
`TypeError`	Operation applied to an object of the wrong type
`UnboundLocalError`	Usage of a local variable before a value is assigned

3.2.2 Exceptions and Control Flow

Few exceptions can alter the control flow.

`SystemExit` - Raised to indicate program exit. The message is printed to `sys.stderr` , and terminated with exit code 1.

```python
import sys

if len(sys.argv) != 2:
   raise SystemExit(f'Usage: {sys.argv[0]} filename')

filename = sys.argv[1]
```

3.2.3 User defined exceptions

see : custom_exceptions.py

```python
class DeviceError(Exception):
    def __init__(self, errno, msg) -> None:
        self.args = (errno, msg)
        self.errno = errno
        self.msg = msg

    # or if you dont want seperate variables for the args

    # def __init__(self, *args: object) -> None:
    #     super().__init__(*args)


try:
    raise DeviceError(1, "Not Responding")
except DeviceError as e:
    print(e.args)
    print(e.errno)
    print(e.msg)

# (1, 'Not Responding', 1)
# 1
# Not Responding
```

Note : When you create a custom exception class that redefines `__init__()` , it is important to assign a tuple containing the arguments of `__init__()` to the attribute `self.args` as shown.

Exception organised into hierarchy using inheritance:

```python
class HostnameError(NetworkError):
    pass
class TimeoutError(NetworkError):
    pass

def error1():
    raise HostnameError('Unknown host')
def error2():
    raise TimeoutError('Timed out')

try:
    error1()
except NetworkError as e:
    if type(e) is HostnameError:
        # Perform special actions for this kind of error
        ...
```

3.2.4 Chained Exception

3.2.4.1 Expected Chained Exception

see : chained_exceptions.py

```python
class ApplicationError(Exception):
    def __init__(self, msg):
        self.msg = msg


def do_something(): # This will produce ValueError exception
    x = int("N/A")
    print(x)


def spam():
    try:
        do_something()
    except Exception as e:
        # from e means chain exception raised intentionally and knowingly from inside e
        raise ApplicationError("do_something() Failed") from e


# spam() ->  #If an uncaught ApplicationError occurs, you will get a message
# that includes both exceptions.

# Lets catch it now.
# start reading here.
try:
    spam()
except ApplicationError as f:
    print(f"{f.msg}. Reason:", f.__cause__)

# __cause__ attribute contains the previous exception
# here for example - <class 'ValueError'> exception
```

NOTE : `__cause__` will only hold previous exception if the chain exception (here ApplicationError) was expected.

3.2.4.2 UnExpected Chained Exception

see : chained_exception_exp_unexp_advanced.py

```python
class ApplicationError(Exception):
    def __init__(self, msg):
        self.msg = msg


class FirstException(Exception):
    def __init__(self, msg, errno):
        self.msg = msg
        self.errno = errno
        self.args = (msg, errno)


def do_something():
    raise FirstException("First Exception", "ERROR1")
    # raise FirstException("First Exception", "1")


def spam():
    try:
        do_something()
    except FirstException as e:
        err_num = int(e.errno)  # This gives error if errno is ERROR1 and not 1
        # This error is unexpected. It will no longer raise ApplicationError. But if it
        print(err_num)
        raise ApplicationError("It Failed") from e

# START READING HERE
try:
    spam()
except Exception as f:
    # depending on errno ERROR1 or 1, it will get an ApplicationError or ValueError.
    print(type(f))
    # This will return None if no unexpected error was raised during FirstExeption,
    # else if the expected ApplicationError was raised the it will hold previous except
    # i.e FirstException
    print("Reason:", f.__cause__)
    # This will return the previous exception in case of both expected and unexpected e
    print("Reason:", f.__context__)

# __cause__ attribute contains the previous exception
# here for example - <class 'ValueError'> exception
```

3.2.5 Exception Traceback

Exceptions have an associated stack traceback that provides information about where an error occurred. The traceback is stored in the **traceback** attribute of an exception. It will print the Original traceback message.

```
# TRACEBACK ERROR
    tbline = traceback.format_exception(type(f), f, f.__traceback__)
    TBMSG = "".join(tbline)
    print(TBMSG)
```

3.2.6 Exception Handling Advice

1. The first rule is to not catch exceptions that can't be handled at that specific location in the code. eg : in `read_data()` function if the filename has error and file can not be opened. . It's better to let the operation fail and report an exception back to the caller. Avoiding an error check in read_data() doesn't mean that the exception would never be handled anywhere—it just means that it's not the role of read_data() to do it. Perhaps the code that prompted a user for a filename would handle this exception.
2. On the other hand, a function might be able to recover from bad data. Then use exception to deal with the bad data.
3. When catching errors, try to make your except clauses as narrow as reasonable.
4. Finally, if you're explicitly raising an exception, consider making your own exception types.

3.3 Context Managers & "with" statement

`read` : context_manager.py

in the `read` we did context manager with unexpected exception handling with **context** attribute.

> **GIST** : A raised exception can cause control flow to bypass statements responsible for releasing critical resources, such as a lock or close a file.
> The `with` statement allows a series of statements to execute inside a runtime context that is controlled by an object serving as a context manager.

Simple Context Manager :

```python
with open('debuglog', 'wt') as file:
    file.write('Debugging\n')
    ...
    file.write('Done\n')

# automatically closes file when context exists
```

1. When the with obj statement executes, it calls the method `obj.__enter__()` to signal that a new context is being entered. The `__enter__()` returns and its stores `as var`. It can return `self` like in most cases because this allows an object to be constructed and used as a context manager in the same step (like opening file) or a list in case of `ListTransaction` in context_manager.py
2. When control flow leaves the context, the method `obj.__exit__(type, value, traceback)` executes. **No error : args is `None`. If error args from the exception.** If the `__exit__()` method returns True, it indicates that the raised exception was handled and should no longer be propagated. (handled in the `__exit__()`) Returning None or False will cause the exception to propagate. (handle the exception in the try/catch)

```python
# ListTransaction where a list is given and modified. But modification only happens if
# no error encountered during each step of the modification.
# Also there will be a memory error if length exceeds MAX_LEN_LIST but recoverable and
# modified without the exceeding objects


class MyException(Exception):
    def __init__(self, msg) -> None:
        self.msg = msg
        self.errno = "LST_ERR"
        self.args = (self.msg, self.errno)

    def __str__(self) -> str:
        return self.msg + "/" + self.errno


class OutOfMemoryError(Exception):
    def __init__(self, msg) -> None:
        self.msg = msg
        self.errno = "OFM>6"
        self.args = (self.msg, self.errno)

    def __str__(self) -> str:
        return self.msg + "/" + self.errno


class ListTransaction:
    def __init__(self, thelist):
        self.thelist = thelist
        self.__workingcopy = []  # we will work on this and not the original list

    def __enter__(self):
        self.__workingcopy = list(self.thelist)
        return (
            self.__workingcopy
        )  # we return the working copy as a new list to be held in the as var

    def __exit__(self, type, value, tb):
        if type is None:  # No error occured during context)
            self.thelist[:] = self.__workingcopy
            return True

        if type is OutOfMemoryError:
```

```python
            print(value)
            # choice = input("Do you want to modify List")
            # exclude the last element that was out of memory
            self.thelist[:] = self.__workingcopy[:-1]

            # We dont need to catch the OutOfMemoryError because we already printed the
            # when the OutOfMemoryError cause the __exit__() to be called by using
            # the print(value) statement. thus we return true
            return True

        # if not OutOfMemory and Exception occured.
        # it will get invoked if `it.append(int("*"))` is uncommented
        raise MyException("Error during List Modification")


# ------------------------
# || START READING HERE ||
# ------------------------
items = [1, 2, 3]
MAX_LEN_LIST = 6

print(f"initial list : {items}")

try:
    with ListTransaction(items) as it:
        # we will keep acepting integers until user enters n or N.
        while (append_char := input("append() : ")) != "n" and (append_char != "N"):
            it.append(int(append_char))

            if len(it) > int(MAX_LEN_LIST):
                raise OutOfMemoryError("Out of memory")

except MyException as f:
    print(f, "because of : ", f.__context__)


print(f"updated list : {items}")
```

CHAPTER 4 : Objects, Types and Protocols

4.1 Essential Concepts

1. **Every piece of data stored in a program is an object**.
2. **Every object has :**
    - **Identity** : Memory Location. (pointer)
    - **Type** : Object's Class (like Integer object)
    - **Value** : The contents at the memory location (*pointer)

- The type of an object, also known as the object's class, defines the object's internal data representation as well as supported methods. When an object of a particular type is created, that object is called an instance of that type. *Attributes of an object are accessed using the dot operator.*
- **Container** : An object that holds references to other objects.
  eg : `writing a + 10 executes a method a.__add__(10).`

3. `id()` returns an integer corresponding to memory location of the object. `is` operator compares memory location. `==` operator conpares values.
4. `type()` returns type of object.
    - The type of an object is itself an object, known as object's class.

```
items = list()
if isinstance(items, list):
    items.append(item)
def removeall(items: list, item) -> list:
    return [i for i in items if i != item]
```

4. `subtypes` : A subtype is a type defined by inheritance. It carries all of the features of the original type plus additional and/or redefined methods.

```python
class MyList(list):
    def removeall(self, item) -> list:
        return [i for i in self if i != item]


my_list = MyList([1, 2, 3, 4, 5, 6, 7, 8, 2, 5, 2])
new_list = my_list.removeall(2)

# new_list = [1, 3, 4, 5, 6, 7, 8, 5]
```

4.2 Reference Counting and Garbage Collection

Python manages objects through automatic garbage collection. **All objects are reference-counted.** An object's reference count is increased whenever it's assigned to a new name or placed in a container such as a list, tuple, or dictionary.

```python
a = 37       # Creates an object with value 37
b = a        # Increases reference count on 37
c = []
c.append(b)  # Increases reference count on 37
```

Only one object - Integer object 37 is created, all rest creates references to this.

```python
del a # Decrease reference count of 37
b = 42 # Decrease reference count of 37
c[0] = 2.0 # Decrease reference count of 37
```

`sys.getrefcount(obj)` - gives the reference count of the object.

When an object's reference count reaches zero, it is garbage-collected.

Circular Dependencies :
However, in some cases a ***circular dependency*** may exist in a collection of objects that are no longer in use.

```python
a={ }
b={ }
a['b'] = b   # a contains reference to b
b['a'] = a   # b contains reference to a
del a
del b
```

In this example, the `del` statements decrease the reference count of a and b and destroy the names used to refer to the underlying objects. However, since each object contains a reference to the other, the reference count doesn't drop to zero and the objects remain allocated.

The `cycle-detection algorithm` runs periodically as the interpreter allocates more and more memory during execution. It finds and deletes these inaccessable memories. The `gc.collect()` function can be used to immediately invoke the cyclic garbage collector.

use case : During handling of large data sets

```python
def some_calculation():
    data = create_giant_data_structure()
    # Use data for some part of a calculation ...
    # Release the data
    del data

    # Calculation continues
    ...
```

4.3 References and Copies

When a program makes an assignment such as b = a, a new reference to a is created. This is not creating a copy, just assigning pointers.

`b is a` : True

Shallow Copy :
Creates a new object, but populates it with references to the items contained in the original object.

```python
a = [1, 2, [3, 4]]
b = list(a)

a[0] = -1   # wont change b[0]. new obj -1 is placed in a[0] instead of changing object
b.append(9) # wont change a because a new object 9 is assigned to b at the end

#thes will change both a and b because the object(list) [3, 4] which is shared between
a[2][0] = -5
b[2][0] = -8


print(a, b)

# [-1, 2, [-5, -8]] [1, 2, [-5, -8], 9]
```

Deep Copy :

`copy.deepcopy()` creates a completely new object using the copy module. Reserve deepcopy() for situations where you actually need a copy because you're about to mutate data and you don't want your changes to affect the original object.

> **NOTE : deepcopy() will fail with objects that involve system or runtime state (such as open files, network connections, threads, generators, and so on).**

4.4 Object Representation and Printing

```python
import datetime

d = datetime.date(2022, 11, 7)
print(d)  # This calls the `str(d)` function of the object class

# use the repr(x) function that creates a string with a representation
# of the object that you would have to type out in source code to
# create it.
repr_d = repr(d)
print(f"Date : {repr_d}") # Date : datetime.date(2022, 11, 7)
print(f"Date : {d!r}")    # Date : datetime.date(2022, 11, 7)
```

> **NOTE : `!r` suffix can be added to a value to produce its repr() value instead of the normal string conversion.**

4.5 First-Class Objects

All objects in Python are said to be first-class. This means that all objects that can be assigned to a name can also be treated as data.

Eg : The items dictionary now contains a function, a module, an exception, and a method of another object.

```
import math

items = {
    "number": 42,
    "text": "Hello World"
}

items["abs"] = abs
items["math"] = math
items["error"] = ValueError

nums = [1, 2, 3, 4]
items["append"] = nums.append

# We can now use dict lookup instead of directly using
# the functions and objects
a = items["abs"](-11)  # a = 11
b = items["math"].sqrt(4)  # b = 2

try:
    x = int("x")
except items["error"] as e:
    print("Couldn't convert")

items["append"](100)  # nums = [1, 2, 3, 4, 100]
```

Real life example :

Convert “ACME,100,490.10” into list with correct types.

```python
LINE = "ACME,100,490.10"

types = [str, int, float]
list_line = LINE.split(",")

# zip gives like (<class 'str'>, 'ACME'). We can just so ty(item)
my_list = [ty(item) for ty, item in zip(types, list_line)]

#['ACME', 100, 490.1]
```

We can rewrite : to avoid complex if/else structures

```python
if format == 'text':
    formatter = TextFormatter()
elif format == 'csv':
    formatter = CSVFormatter()
elif format == 'html':
    formatter = HTMLFormatter()
else:
    raise RuntimeError('Bad format')
```

as

```python
_formats = {
    'text' : TextFormatter,
    'csv' : CSVFormatter,
    'html' : HTMLFormatter
}
if format in _formats:
    formatter = _formats[format]()
```

4.6 Object Protocols and Data Abstraction

Most Python language features are defined by protocols.

```python
def compute_cost(unit_price, num_units):
    return unit_price * num_units
```

Here the function can accept a lot of data types. Some works and some fails. It depends on the implementation of the `*` operator object. ***Unlike a compiler for a static language, Python does not verify correct program behavior in advance.***

Instead, the behavior of an object is determined by a dynamic process that involves the dispatch of so-called "special" or "magic" methods. **The names of these special methods are always preceded and followed by double underscores (__) eg : `__init__()`** . The methods are automatically triggered by the interpreter as a program executes. For example, the operation `x * y` is carried out by a method `x.__mul__(y)` . These are hard wired.

These special methods are associated with different categories of core interpreter features. These catagories are called protocols.

In a user defined class we can implement these special methods to change the way the object behaves or add functionality.

4.6.1 Object Protocol

For overall management of object.

Method	Description
`__new__(cls [,*args [,**kwargs]])`	A static method called to create a new instance.
`__init__(self [,*args [,**kwargs]])`	Called to initialize a new instance after it's been created.
`__del__(self)`	Called when an instance is being destroyed.
`__repr__(self)`	Create a string representation.

[1]. The `__new__()` and `__init__()` methods are used together to create and initialize instances. When an object is created by calling SomeClass(args), it is translated into the following steps:

```
# creates the object without calling the __init__() function
my_list = MyWords.__new__(MyWords)
# then we call the __init__() function
if isinstance(my_list, MyWords):
    my_list.__init__("first_word")
```

Use of `__new__()` implementation almost always indicates the presence of advanced magic related to instance creation (for example, it is used in class methods that want to bypass `__init__()` or in certain creational design patterns such as singletons or caching). The implementation of `__new__()` doesn't necessarily need to return an instance of the class in question—if not, the subsequent call to `__init__()` on creation is skipped.

[2]. The `__del__()` method is invoked when an instance is about to be garbage-collected. This method is invoked only when an instance is no longer in use. Note that the statement `del x` only decrements the instance reference count and doesn't necessarily result in a call to this function.

[3]. The `__repr__()` method, called by the built-in `repr()` function, creates a string representation of an object that can be useful for debugging and printing. `eval()` the output of `repr` just recreates the object.

4.6.2 Number Protocol

TODO
TODO

4.6.3 Comparison Protocol

`is` checks for identity. i.e memory address. It has nothing to do with value stored.

Method	Description
`__bool__(self)`	- Returns False or True for truth-value testing. - If bool undefined `__len__()` fallback. eg : `if a:` executes `if a__bool__()` :
`__eq__(self, other)`	self == other. - It's for use with `==` and `!=` operators. First `__eq__()` checks with `is` to match `id()` . If its not same then it goes on to check the values. Because if `id()` is same then values have to be same.

Method	Description
`__ne__(self, other)`	self != other
`__lt__(self, other)`	self < other
`__le__(selfm, other)`	self <= other
`__gt__(self, other)`	self > other
`__ge__(self, other)`	self >= other
`__hash__(self)`	Computes integer hash index
Ordering is determined by (<, >, <=, >=) operators using the `__lt__()` and `__gt__()` . To evaluate `a < b` , interpreter first tries `a.__lt__(b)` except where b is subset of a (then `b.__gt__(a) is used`). If it's not implemented then interpreter tries reversed comparison using `b.__gt__(a)` . Same with >= and <=	

example :

```
a = 42
b = 52.3

a.__lt__(b)    # NotImplemented
b.__gt__(a)    # True
```

> **NOTE : It is not necessary for an ordered object to implement all of the comparison operations in Table 4.3. If you want to be able to sort objects or use functions such as `min() or max()` , then `__lt__()` must be minimally defined. If you are adding comparison operators to a user-defined class, the @total_ordering class decorator in the functools module may be of some use. It can generate all of the methods as long as you minimally implement `__eq__()` and one of the other comparisons.**

`__hash__()` : It's defined on instances that are to be placed into a set or be used as keys in a mapping (dictionary). The value returned is an integer that should be the same for two instances that compare as equal.

- `__eq__()` should always be defined together with `__hash__()` because the two methods work together.

> **NOTE : The value returned by `__hash__()` is typically used as an internal implementation detail of various data structures. However, it's possible for two different objects to have the same hash value. Therefore, `__eq__()` is necessary to resolve potential collisions.**

4.6.4 Conversion Protocol

Method	Description
`__str__()`	Conversion to string
`__bytes__(self)`	Conversion to bytes
`__format__(self, format_spec)`	Creates a formatted representation
`__bool__(self)`	bool(self)
`__int__(self)` , `__float__(self)`	int(self), float(self)
`__complex__(self)`	complex(self)
`__index__(self)`	Conversion to a integer index [self]

1. `__format__()` is called by `format()` function or `format()` method of strings.

```
x = 42
f'{x:>04d}'                 # Calls x.__format__('>04d')
format(x, '>04d')           # Calls x.__format__('>04d')
'x is {0:>04d}' .format(x)  # Calls x.__format__('>04d')
print(x.__format__('>04d'))  #0042
```

2. Python never performs implicit type conversions using these methods. Thus, even if an object x implements an `__int__()` method, the expression 3 + x will still produce a TypeError. The only way to execute `__int__()` is through an explicit use of the int() function.
3. The `__index__()` method performs an integer conversion of an object when it's used in an operation that requires an integer value. This includes indexing in sequence operations. For example, if items is a list, performing an operation such as `items[x]` will attempt to execute `items[x.__index__()]` if x is not an integer. `__index__()` is also used in various base conversions such as oct(x) and hex(x).

TODO (find examples)

4.6.5 Container Protocol

They are used by objects that want to implement containers of various kinds—lists, dicts, sets, and so on.

Method	Description
`__len__(self)`	Returns the length of `self`
`__getitem__(self, key)`	Returns `self[key]`
`__setitem__(self, key, value)`	Sets `self[key] = value`
`__delitem__(self, key)`	Deletes self[key]
`__contains__(self, obj)`	obj in self

Example :

```
a = [1, 2, 3]     # a.__len__()
len(a)            # x = a.__getitem__(2)
x = a[2]          # x = a.__getitem__(2)
a[1] = 7          # a.__setitem__(1,7)
del a[2]          # a.__delitem__(2)
5 in a            # a.__contains__(5)
```

Slicing operations such as `x = s[i:j]` are also implemented using `__getitem__()` , `__setitem__()` , and `__delitem__()` . For slices, a special `slice` instance is passed as the key.

```
a = [1,2,3,4,5,6]
x = a[1:5:2]          # x = a.__getitem__(slice(1, 5, 2))
a[1:3] = [10,11,12] # a.__setitem__(slice(1, 3, None), [10, 11, 12])
del a[1:4]          # a.__delitem__(slice(1, 4, None))
```

TODO - Multi-Dimensional Slices

NOTE : No part of Python or its standard library make use of multidimensional slicing or the Ellipsis. Those features are reserved purely for third-party libraries and frameworks. Perhaps the most common place you would see them used is in a library such as `numpy` .

4.6.6 Iteration Protocol

check - iterator_protocol.py

- If an instance, obj, supports iteration, it provides a method, `obj.__iter__()` , that returns an iterator.
- The iterator `_iter` , in turn, implements a single method, `_iter.__next__()` , that returns the next object or raises `StopIteration` to signal the end of iteration.

Implement `for i in my_list`

```python
my_list = [1, 2, 3, 4, 5]

_iter = my_list.__iter__()  # this is the iterator object

while True:
    try:
        i = _iter.__next__()
        print(i)
    except StopIteration:
        break

_iter_reversed = my_list.__reversed__()  # returns reversed iterator
```

`__reversed__()` is called by the built-in `reversed()` function.

[o] A common implementation technique for iteration is to use a generator function involving `yield`.

```python
class FRange:
    def __init__(self, start, stop, step) -> None:
        self.start = start
        self.stop = stop
        self.step = step

    def __iter__(self):
        x = self.start
        while x < self.stop:
            yield x  # adding items to the generator
            x += self.step


# Example use:
nums = FRange(0.0, 1.0, 0.1)

for i in nums:
    print(f"{i:0.1f}")
```

Here FRange is an iterator class which returns a iterator (generator) object. ***This works because generator functions conform to the iteration protocol themselves.*** It's a bit easier to implement an iterator in this way since you only have to worry about the `__iter__()` method. The rest of the iteration machinery is already provided by the generator.

4.6.7 Attribute Protocol

Method	Description
`__getattribute__(self, name)`	returns the attribute `self.name`
`__getattr__(self,name)`	Returns the attribute `self.name` if it's not found through `__getattribute__()`
`__setattr__(self,name, value)`	Sets the attribute `self.name = value`
`__delattr__(self,name)`	Deletes the attribute `del self.name`

Whenever an attribute is accessed, the `__getattribute__()` method is invoked. If the attribute is located, its value is returned. Otherwise, the `__getattr__()` method is invoked. The default behavior of `__getattr__()` is to raise an AttributeError exception. The `__setattr__()` method is always invoked when setting an attribute, and the `__delattr__()` method is always invoked when deleting an attribute.

4.6.8 Function Protocol

An object can emulate a function by providing the `__call__()` method. If an object, x, provides this method, it can be invoked like a function. That is, `x(arg1, arg2, ...)` invokes `x.__call__(arg1, arg2, ...)`.

eg : Replicates the behaviour of `int()` method in int object

```
class Int:
    def __call__(self, string):
        return int(string)


integer = Int()
print(integer("8"))
```

4.6.9 Context Manager Protocol

Method	Description
`__enter__(self)`	Called when entering a new context. The return value is placed in the variable listed with the as specifier to the with statement.
`__exit__(self, type, value, tb)`	Called when leaving a context. If an exception occurred, type, value, and tb have the exception type, value, and traceback information.

Summery :

The `__enter__()` method is invoked when the with statement executes. The value returned by this method is placed into the variable specified with the optional as var specifier. The `__exit__()` method is called as soon as control flow leaves the block of statements associated with the with statement. As arguments, `__exit__()` receives the current exception type, value, and a traceback if an exception has been raised. If no errors are being handled, all three values are set to None. The `__exit__()` method should return True or False to indicate if a raised exception was handled or not.

If True is returned, any pending exception is cleared and program execution continues normally with the first statement after the with block.

CHAPTER 5 : Functions

5.1 Default Arguments

```
def split(line, delimiter=','):
    statements
```

1. Once you specify a default parameter, all the following parameters must have a default value.
2. Default parameter values are evaluated once when the function is first defined, not each time the function is called. This often leads to surprising behavior if mutable objects are used as a default.

   ```
   def func(x, items=[]):
       items.append(x)
       return items
   ```

   > func(1) -> [1]
   > func(2) -> [1, 2]
   > func(3) -> [1, 2, 3]

   So better to use `items = None` .
   `NOTE` **: *As a general practice, to avoid such surprises, only use immutable objects for default argument values—numbers, strings, Booleans, None, and so on.***

5.2 Variadic Arguments

Variadic arguments are acce-pted by using `*args` as the last argument. All of the extra arguments are placed into the args variable as a tuple.

```python
def product(first, *args):
    result = first
    for x in args:
        result = result * x
    return result
```

5.3 Keyword Arguments

Function arguments can be supplied by explicitly naming each parameter and specifying a value.

```python
def func(w, x, y, z):
    statements

func(x=3, y=22, w='hello', z=[1, 2])
```

The order of arguments does not matter.

`RULE : func(positional, keyword)` all the positional arguments appear first, values are provided for all nonoptional arguments, and no argument receives more than one value.

Force the use of keyword arguments :
All args after the `*` are compulory keyword args

```python
def read_data(filename, *, debug=False):
    ...

ata = read_data('Data.csv', True)        # ERROR: TypeError
data = read_data('Data.csv', debug=True)  # Yes.
```

```python
def product(first, *values, scale=1):
    result = first * scale
    for val in values:
        result = result * val
        return result

result = product(2,3,4)             # Result = 24
result = product(2,3,4, scale=10)   # Result = 240, (3, 4) in value
```

5.4 Variadic Keyword Arguments

read : arguments.py

If the **last argument of a function** definition is prefixed with `**` , all the additional keyword arguments (those that don't match any of the other parameter names) are placed in a **dictionary** and passed to the function. **The order of items in this dictionary is guaranteed to match the order in which keyword arguments were provided**.

```
# we force engine_type and autopilot to be keyword type args and the rest goes into **k
def make_car(self, make_year, model, *, engine_type, autopilot, **kwargs):
```