# 1. Import libraries

In [ ]:

```
# 'generic import' of math module
import math
math.sqrt(25)
```

Out[ ]:

```
5.0
```

In [ ]:

```
# import a function
from math import sqrt
sqrt(25)
```

Out[ ]:

```
5.0
```

In [ ]:

```
# import multiple functions at once
from math import cos, floor

# import all functions in a module (generally discouraged)
from os import *
```

In [ ]:

```
# define an alias
import numpy as np
np.sqrt(36)
```

Out[ ]:

```
6.0
```

In [ ]:

```
# show all functions in math module
content = dir(math)
print(content)
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atan
h', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factoria
l', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt',
'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

# 2. Basic operations

In [ ]:

```
# Numbers
print(10 + 4)  # add (returns 14)
print(10 - 4)  # subtract (returns 6)
print(10 * 4)  # multiply (returns 40)
print(10 / 4)  # true division (returns 2.5)
print(10 // 4)  # floor division (returns 2)
print(10 ** 4)  # exponent (returns 10000)
print(10 / float(4))  # divide (returns 2.5)
print(5 % 4)  # modulo (returns 1) - also known as the remainder
```

```
14
6
40
2.5
2
10000
2.5
1
```

In [ ]:

```
# Boolean operations
# comparisons (these return True)
print(5 > 3)
print(5 >= 3)
print(5 != 3)
print(5 == 2)
```

```
True
True
True
True
```

In [ ]:

```
# boolean operations (these return True)
print(5 > 3 and 6 > 3)
print(5 > 3 or 5 < 3)
print(not False)
```

```
True
True
True
```

# 3. Data types

In [ ]:

```
# determine the type of an object
print(type(2))  # returns 'int'
print(type(2.0))  # returns 'float'
print(type('two'))  # returns 'str'
print(type(True))  # returns 'bool'
print(type(None))  # returns 'NoneType'
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
<class 'NoneType'>
```

In [ ]:

```
# check if an object is of a given type
print(isinstance(2.0, int))  # returns False
print(isinstance(2.0, (int, float)))  # returns True
```

```
False
True
```

In [ ]:

```
# convert an object to a given type
print(float(2))
print(int(2.9))
print(str(2.9))
```

```
2.0
2
2.9
```

In [ ]:

```
# zero, None, and empty containers are converted to False
print(bool(0))
print(bool(None))
print(bool(''))  # empty string
print(bool([]))  # empty list
print(bool({}))  # empty dictionary
```

```
False
False
False
```

```
False
False
```

In [ ]:

```
# non-empty containers and non-zeros are converted to True
print(bool(2))
print(bool('two'))
print(bool([2]))
```

```
True
True
True
```

## 3.1 Lists

lists are ordered, iterable, mutable (adding or removing objects changes the list size) can contain multiple data types

In [ ]:

```
# create an empty list (two ways)
empty_list = []
empty_list = list()
```

In [ ]:

```
# create a list
names = ['homer', 'marge', 'bart']

# examine a list
names[0]  # print element 0 ('homer')
len(names)  # returns the length (3)
```

Out[ ]:

```
3
```

In [ ]:

```
# modify a list (does not return the list)
names.append('lisa')  # append element to end
names.extend(['itchy', 'scratchy'])  # append multiple elements to end
# insert element at index 0 (shifts everything_ right)
names.insert(0, 'maggie')
names.remove('bart')  # searches for first instance and removes it
names.pop(0)  # removes element 0 and returns it
del names[0]  # removes element 0 (does not return it)
names[0] = 'krusty'  # replace element 0
```

In [ ]:

```python
# concatenate lists (slower than 'extend' method)
neighbors = names + ['ned', 'rod', 'todd']
```

In [ ]:

```python
# find elements in a list
names.count('lisa')  # counts the number of instances
names.index('itchy')  # returns index of first instance
```

Out[ ]:

```
2
```

In [ ]:

```python
# list slicing [start:end:stride]
weekdays = ['mon', 'tues', 'wed', 'thurs', 'fri']
weekdays[0]  # element 0
weekdays[0:3]  # elements 0, 1, 2
weekdays[:3]  # elements 0, 1, 2
weekdays[3:]  # elements 3, 4
weekdays[-1]  # last element (element 4)
weekdays[::2]  # every 2nd element (0, 2, 4)
weekdays[::-1]  # backwards (4, 3, 2, 1, 0)
```

Out[ ]:

```
['fri', 'thurs', 'wed', 'tues', 'mon']
```

In [ ]:

```python
# alternative method for returning the list backwards
list(reversed(weekdays))
```

Out[ ]:

```
['fri', 'thurs', 'wed', 'tues', 'mon']
```

In [ ]:

```python
# sort a list in place (modifies but does not return the list)
names.sort()
names.sort(reverse=True)  # sort in reverse
names.sort(key=len)  # sort by a key
```

In [ ]:

```python
# return a sorted list (but does not modify the original list)
sorted(names)
sorted(names, reverse=True)
sorted(names, key=len)
```

Out[ ]:

```
['lisa', 'itchy', 'krusty', 'scratchy']
```

In [ ]:

```
# create a second reference to the same list
num = [1, 2, 3]
same_num = num
same_num[0] = 0  # modifies both 'num' and 'same_num'
```

In [ ]:

```
# copy a list (three ways)
new_num = num.copy()
new_num = num[:]
new_num = list(num)
```

In [ ]:

```
# examine objects
id(num) == id(same_num)  # returns True
id(num) == id(new_num)  # returns False
num is same_num  # returns True
num is new_num  # returns False
num == same_num  # returns True
num == new_num  # returns True (their contents are equivalent)
```

Out[ ]:

```
True
```

In [ ]:

```
# conatenate +, replicate *
[1, 2, 3] + [4, 5, 6]
["a"] * 2 + ["b"] * 3
```

Out[ ]:

```
['a', 'a', 'b', 'b', 'b']
```

## 3.2 Tuples

Like lists, but their size cannot change: ordered, iterable, immutable, can contain multiple data types

In [ ]:

```
# create a tuple
digits = (0, 1, 'two')  # create a tuple directly
digits = tuple([0, 1, 'two'])  # create a tuple from a list
zero = (0,)  # trailing comma is required to indicate it's a tuple
```

In [ ]:

```
# examine a tuple
```

```
digits[2]  # returns 'two'
len(digits)  # returns 3
digits.count(0)  # counts the number of instances of that value (1)
digits.index(1)  # returns the index of the first instance of that value (1)
```

Out[ ]:

```
1
```

In [ ]:

```
# elements of a tuple cannot be modified
digits[2] = 2  # throws an error
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
f:\fci\Pattern Recognation\DAY 01\00-Python Basics.ipynb Cell 34 in <cell line: 2>()
      <a href='vscode-notebook-cell:/f%3A/fci/Pattern%20Recognation/DAY%2001/00-Python%20Basics.ipynb#X45sZmlsZQ%3D%3D?line=0'>1</a> # elements of a tuple cannot be modified
----> <a href='vscode-notebook-cell:/f%3A/fci/Pattern%20Recognation/DAY%2001/00-Python%20Basics.ipynb#X45sZmlsZQ%3D%3D?line=1'>2</a> digits[2] = 2

TypeError: 'tuple' object does not support item assignment
```

In [ ]:

```
# concatenate tuples
digits = digits + (3, 4)
```

In [ ]:

```
# create a single tuple with elements repeated (also works with lists)
(3, 4) * 2  # returns (3, 4, 3, 4)
```

Out[ ]:

```
(3, 4, 3, 4)
```

## 3.3 Strings

In [ ]:

```
# create a string
s = str(42)  # convert another data type into a string
s = 'I like you'
```

In [ ]:

```
# examine a string
s[0]  # returns 'I'
len(s)  # returns 10
```

Out[ ]:
```
10
```

In [ ]:
```python
# string slicing like lists
s[:6] # returns 'I like'
s[7:] # returns 'you'
s[-1] # returns 'u'
```

Out[ ]:
```
'I like'
```

In [ ]:
```python
# basic string methods (does not modify the original string)
s.lower()  # returns 'i like you'
s.upper()  # returns 'I LIKE YOU'
s.startswith('I')  # returns True
s.endswith('you')  # returns True
s.isdigit()  # returns False (returns True if every character in the string is a␣ digit)
s.find('like')  # returns index of first occurrence (2), but doesn't support regex
s.find('hate')  # returns -1 since not found
s.replace('like', 'love')  # replaces all instances of 'like' with 'love'
```

Out[ ]:
```
'I love you'
```

In [ ]:
```python
# split a string into a list of substrings separated by a delimiter
s.split(' ') # returns ['I','like','you']
s.split() # same thing
s2 = 'a, an, the'
s2.split(',') # returns ['a',' an',' the']
```

Out[ ]:
```
['a', ' an', ' the']
```

In [ ]:
```python
# join a list of strings into one string using a delimiter
stooges = ['larry', 'curly', 'moe']
' '.join(stooges)  # returns 'larry curly moe'
```

Out[ ]:
```
'larry curly moe'
```

In [ ]:
```python
# concatenate strings
s3 = 'The meaning of life is'
s4 = '42'
```

```
s3 + ' ' + s4  # returns 'The meaning of life is 42'
s3 + ' ' + str(42)  # same thing
```

```
Out[ ]: 'The meaning of life is 42'
```

```
In [ ]:

# remove whitespace from start and end of a string
s5 = ' ham and cheese '
s5.strip()  # returns 'ham and cheese'
```

```
Out[ ]: 'ham and cheese'
```

```
In [ ]:

# string substitutions: all of these return 'raining cats and dogs'
'raining %s and %s' % ('cats', 'dogs')  # old way
'raining {} and {}'.format('cats', 'dogs')  # new way
'raining {arg1} and {arg2}'.format(arg1='cats', arg2='dogs')  # named arguments
```

```
Out[ ]: 'raining cats and dogs'
```

```
In [ ]:

# string formatting
# more examples: http://mkaz.com/2012/10/10/python-string-format/
'pi is {:.2f}'.format(3.14159)  # returns 'pi is 3.14'
```

```
Out[ ]: 'pi is 3.14'
```

## 3.5 Dictionaries

Dictionaries are structures which can contain multiple data types, and is ordered with key-value pairs: for each (unique) key, the dictionary outputs one value. Keys can be strings, numbers, or tuples, while the corresponding values can be any Python object. Dictionaries are: unordered, iterable, mutable

```
In [ ]:

# create an empty dictionary (two ways)
empty_dict = {}
empty_dict = dict()
```

```
In [ ]:

# create a dictionary (two ways)
family = {'dad': 'homer', 'mom': 'marge', 'size': 6}
family = dict(dad='homer', mom='marge', size=6)
```

In [ ]:

```
# convert a list of tuples into a dictionary
list_of_tuples = [('dad', 'homer'), ('mom', 'marge'), ('size', 6)]
family = dict(list_of_tuples)
```

In [ ]:

```
# examine a dictionary
family['dad']  # returns 'homer'
len(family)  # returns 3
family.keys()  # returns list: ['dad', 'mom', 'size']
family.values()  # returns list: ['homer', 'marge', 6]
family.items()  # returns list of tuples:
# [('dad', 'homer'), ('mom', 'marge'), ('size', 6)]
'mom' in family  # returns True
'marge' in family  # returns False (only checks keys)
```

Out[ ]:

```
False
```

In [ ]:

```
# modify a dictionary (does not return the dictionary)
family['cat'] = 'snowball'  # add a new entry
family['cat'] = 'snowball ii'  # edit an existing entry
del family['cat']  # delete an entry
family['kids'] = ['bart', 'lisa']  # value can be a list
family.pop('dad')  # removes an entry and returns the value ('homer')
family.update({'baby': 'maggie', 'grandpa': 'abe'})  # add multiple entries
```

In [ ]:

```
# accessing values more safely with 'get'
family['mom'] # returns 'marge'
family.get('mom') # same thing

try:
    family['grandma'] # throws an error
except KeyError as e:
    print("Error", e)

family.get('grandma') # returns None
family.get('grandma', 'not found') # returns 'not found' (the default)
```

```
Error 'grandma'
```

Out[ ]: 'not found'

In [ ]:

```
# accessing a list element within a dictionary
family['kids'][0]  # returns 'bart'
family['kids'].remove('lisa')  # removes 'lisa'
```

## 3.6 Sets

Like dictionaries, but with unique keys only (no corresponding values). They are: unordered, iterable, mutable, can contain multiple data types made up of unique elements (strings, numbers, or tuples)

In [ ]:

```
# create an empty set
empty_set = set()

# create a set
languages = {'python', 'r', 'java'}  # create a set directly
snakes = set(['cobra', 'viper', 'python'])  # create a set from a list
```

In [ ]:

```
# examine a set
len(languages)  # returns 3
'python' in languages  # returns True
```

Out[ ]: True

In [ ]:

```
# set operations
languages & snakes  # returns intersection: {'python'}
languages | snakes  # returns union: {'cobra', 'r', 'java', 'viper', 'python'}
languages - snakes  # returns set difference: {'r', 'java'}
snakes - languages  # returns set difference: {'cobra', 'viper'}
```

Out[ ]: {'cobra', 'viper'}

In [ ]:

```
# modify a set (does not return the set)
languages.add('sql')  # add a new element
languages.add('r')  # try to add an existing element (ignored, no error)
languages.remove('java')  # remove an element
```

```
try:
    # try to remove a non-existing element (throws an error)
    languages.remove('c')
except KeyError as e:
    print("Error", e)

languages.discard('c')  # removes an element if present, but ignored otherwise
languages.pop()  # removes and returns an arbitrary element

languages.clear()  # removes all elements
# add multiple elements (can also pass a list or set)
languages.update('go', 'spark')
```

```
Error 'c'
```

In [ ]:

```
# get a sorted list of unique elements from a list
sorted(set([9, 0, 2, 1, 0])) # returns [0, 1, 2, 9]
```

Out[ ]:

```
[0, 1, 2, 9]
```

# 4. Execution control statements

## 4.1 Conditional statements

In [ ]:

```
x = 3
# if statement
if x > 0:
    print('positive')
```

```
positive
```

In [ ]:

```
# if/else statement
if x > 0:
    print('positive')
else:
    print('zero or negative')
```

```
positive
```

In [ ]:

```
# if/elif/else statement
if x > 0:
    print('positive')
elif x == 0:
    print('zero')
else:
    print('negative')
```

```
positive
```

In [ ]:

```
# single-line if/else statement
# known as a 'ternary operator'
'positive' if x > 0 else 'zero or negative'
```

Out[ ]:

```
'positive'
```

## 4.2 Loops

In [ ]:

```
# range returns a list of integers
range(0, 3) # returns [0, 1, 2]: includes first value but excludes second value
range(3) # same thing: starting at zero is the default
range(0, 5, 2) # returns [0, 2, 4]: third argument specifies the 'stride'
```

Out[ ]:

```
range(0, 5, 2)
```

In [ ]:

```
# for loop (not recommended)
fruits = ['apple', 'banana', 'cherry']
for i in range(len(fruits)):
    print(fruits[i].upper())
```

```
APPLE
BANANA
CHERRY
```

In [ ]:

```
# alternative for loop (recommended style)
for fruit in fruits:
    print(fruit.upper())
```

```
APPLE
BANANA
```

```
CHERRY
```

In [ ]:

```python
# iterate through two things at once (using tuple unpacking)
family = {'dad':'homer', 'mom':'marge', 'size':6}
for key, value in family.items():
    print(key, value)
```

```
dad homer
mom marge
size 6
```

In [ ]:

```python
# use enumerate if you need to access the index value within the loop
for index, fruit in enumerate(fruits):
    print(index, fruit)
```

```
0 apple
1 banana
2 cherry
```

In [ ]:

```python
# for/else loop
for fruit in fruits:
    if fruit == 'banana':
        print("Found the banana!")
        break # exit the loop and skip the 'else' block
    else:
        # this block executes ONLY if the for loop completes without hitting 'break'
        print("Can't find the banana")
```

```
Can't find the banana
Found the banana!
```

In [ ]:

```python
# while loop
count = 0
while count < 5:
    print("This will print 5 times")
    count += 1 # equivalent to 'count = count + 1'
```

```
This will print 5 times
This will print 5 times
This will print 5 times
This will print 5 times
This will print 5 times
```

# 5. Functions

In [ ]:

```
# define a function with no arguments and no return values
def print_text():
    print('this is text')

# call the function
print_text()
```

```
this is text
```

In [ ]:

```
# define a function with one argument and no return values
def print_this(x):
    print(x)
```

In [ ]:

```
# call the function
print_this(3) # prints 3
n = print_this(3) # prints 3, but doesn't assign 3 to n
                  # because the function has no return statement
```

```
3
3
```

In [ ]:

```
# define a function with one argument and one return value
def square_this(x):
    return x ** 2
```

In [ ]:

```
# include an optional docstring to describe the effect of a function
def square_this(x):
    """Return the square of a number."""
    return x ** 2
```

In [ ]:

```
# call the function
square_this(3) # prints 9
var = square_this(3) # assigns 9 to var, but does not print 9
```

In [ ]:

```
# default arguments
```

```python
def power_this(x, power=2):
    return x ** power

power_this(2) # 4
power_this(2, 3) # 8
```

Out[ ]: 8

In [ ]:

```python
# use 'pass' as a placeholder if you haven't written the function body
def stub():
    pass
```

In [ ]:

```python
# return two values from a single function
def min_max(nums):
    return min(nums), max(nums)

# return values can be assigned to a single variable as a tuple
nums = [1, 2, 3]
min_max_num = min_max(nums) # min_max_num = (1, 3)
```

In [ ]:

```python
# return values can be assigned into multiple variables using tuple unpacking
min_num, max_num = min_max(nums) # min_num = 1, max_num = 3
```

# 6. List comprehensions

Process which affects whole lists without iterating through loops.

In [ ]:

```python
# for loop to create a list of cubes
nums = [1, 2, 3, 4, 5]
cubes = []
for num in nums:
    cubes.append(num**3)
```

In [ ]:

```python
# equivalent list comprehension
cubes = [num**3 for num in nums] # [1, 8, 27, 64, 125]
```

In [ ]:

```
# for loop to create a list of cubes of even numbers
cubes_of_even = []
for num in nums:
    if num % 2 == 0:
        cubes_of_even.append(num**3)
```

In [ ]:

```
# equivalent list comprehension
# syntax: [expression for variable in iterable if condition]
cubes_of_even = [num**3 for num in nums if num % 2 == 0] # [8, 64]
```

In [ ]:

```
# for loop to flatten a 2d-matrix
matrix = [[1, 2], [3, 4]]
items = []
for row in matrix:
    for item in row:
        items.append(item)
```

In [ ]:

```
# equivalent list comprehension
items = [item for row in matrix for item in row] # [1, 2, 3, 4]
```

# 7. Object Oriented Programming (OOP)

In [ ]:

```
import math

class Shape2D:
    def area(self):
        raise NotImplementedError()

# __init__ is a special method called the constructor
# Inheritance + Encapsulation
class Square(Shape2D):
    def __init__(self, width):
        self.width = width
    def area(self):
        return self.width ** 2

class Disk(Shape2D):
```

```python
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return math.pi * self.radius ** 2

shapes = [Square(2), Disk(3)]

# Polymorphism
print([s.area() for s in shapes])

s = Shape2D()
try:
    s.area()
except NotImplementedError as e:
    print("NotImplementedError")
```

```
[4, 28.274333882308138]
NotImplementedError
```

# 8. Exercises

## Exercise 1: functions

Create a function that acts as a simple calulator If the operation is not specified, default to addition If the operation is misspecified, return an prompt message Ex: calc(4,5,"multiply") returns 20 Ex: calc(3,5) returns 8 Ex: calc(1, 2, "something") returns error message

### Solution

In [ ]:

```python
def calc(n1, n2, op='add'):
    if op == 'add':
        return n1 + n2
    elif op == 'subtract':
        return n1 - n2
    elif op == 'multiply':
        return n1 * n2
    elif op == 'divide':
        if n2 != 0:
            return n1 / n2
        else:
            print('Error')
    else:
        print('Error')
```

# Exercise 2: list + loop

Q1) Given a list of numbers, return a list where all adjacent duplicate elements have been reduced to a single element. Ex: [1, 2, 2, 3, 2] returns [1, 2, 3, 2]. You may create a new list or modify the passed in list.

Q2) Remove all duplicate values (adjacent or not) Ex: [1, 2, 2, 3, 2] returns [1, 2, 3]

## Solution

In [ ]:

```
# Q1)
lst = [1, 2, 2, 3, 2]
i = 1
while i < len(lst):
    if lst[i] == lst[i-1]:
      lst.pop(i)
      i -= 1
    i += 1

print(lst)
```

```
[1, 2, 3, 2]
```

In [ ]:

```
# Q2)
lst = [1, 2, 2, 3, 2]
new = []
for n in lst:
    if n not in new:
        new.append(n)

print(new)
```

```
[1, 2, 3]
```

# Exercise 4: OOP

Create a class `Employee` with 2 attributes provided in the constructor: `name`, `years_of_service`. With one method salary with is obtained by `1500 + 100 * years_of_service`. The class must hold the employee name and its salary in a dictionary. The `salary` method returns the salary given a name.

In [ ]:

```
class Employee:
    def __init__(self, name, years_of_service):
        self.data = {}
        self.data[name] = years_of_service

    def salary(self, name):
        x = self.data[name]
        return 1500 + 100 * x

emp1 = Employee('Ali', 1)
emp2 = Employee('Ahmed', 2)

print(emp1.salary('Ali'))
print(emp2.salary('Ahmed'))
```

```
1600
1700
```

# Task

Q1)

Write a program which can compute the factorial of a given numbers. The results should be printed in a comma-separated sequence on a single line.

Suppose the following input is supplied to the program:

8

Then, the output should be:

40320

Q2)

Use a list comprehension to square each odd number in a list. The list is input by a sequence of comma-separated numbers.

Suppose the following input is supplied to the program:

```
1,2,3,4,5,6,7,8,9
```

Then, the output should be:

```
1,3,5,7,9
```