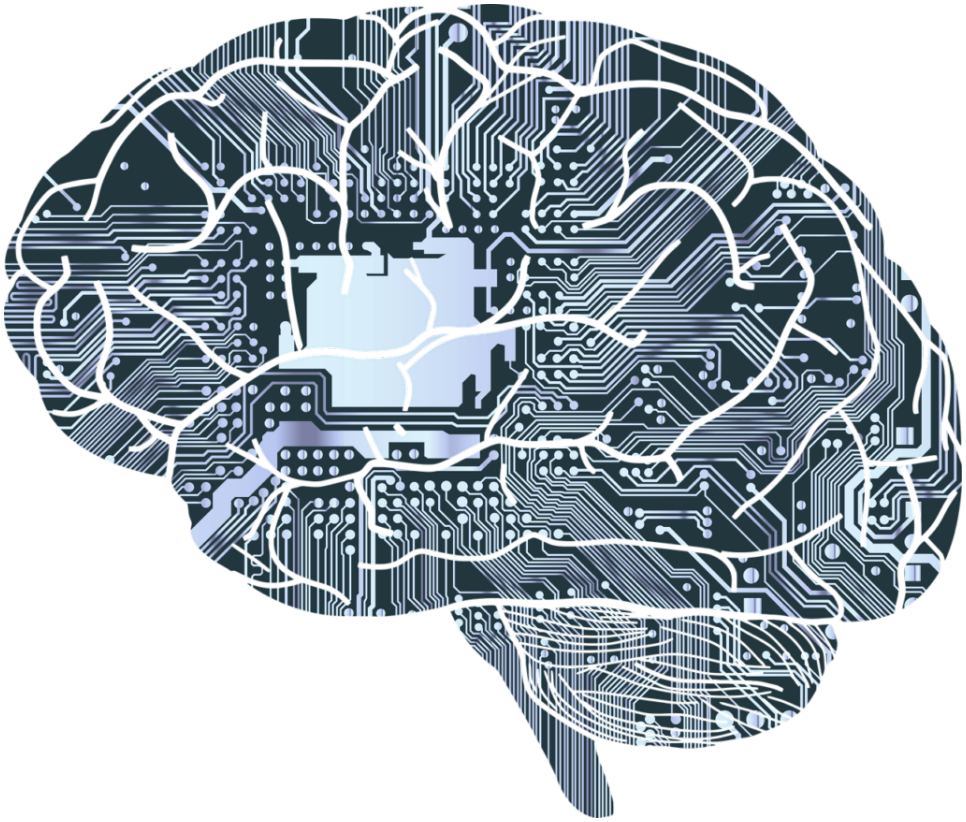
PYTHON

THE FUNDAMENTAL BASICS



T. ABDEL ALIM

CONTENTS

Part I

PYTHON PROGRAMMING

1 OBJECT AND DATA STRUCTURES

1.1 PYTHON BASIC OPERATORS

Arthimetic Operators

+ -	Addition/Subtraction	*	Multiplication	//	Floor Division
**	Exponent	/	Division	%	Modulo

Assignment Operators

+=	Add/Subtract AND	*=	Multiply AND	//=	Floor Division AND
**=	Exponent AND	/=	Divide AND	%=	Modulo AND

The following two operations are identical and show the use of an Assignment Operator:

```
1        c = c + a
2        c += a
```

Comparison Operators

a == b	a is equal to b	a > b	a is bigger than b	a >= b	a is bigger than or equal to b
a != b	a is not equal to b	a < b	a is smaller than b	a <= b	a is smaller than or equal to b

Logical Operators

`and` If both the operands are true then condition becomes true.

`or` If any of the two operands are non-zero then condition becomes true.

`not` Used to reverse the logical state of its operand.

Membership Operators - *test for membership in a sequence*

`in` True if it finds a variable in the specified sequence and false otherwise.

`not in` True if it does not find a variable in the specified sequence and false otherwise.

Identity Operators - *compare the memory locations of two objects*

`is` True if the variables on either side of the operator point to the same object.

`is not` False if the variables on either side of the operator point to the same object.

1.2 VARIABLE TYPES

Variable types can be checked using the `type(...)` function. Common data types include: `int` (whole numbers), `float` (decimal numbers (e.g. 0.3)), `str` (for string), `list`, `tuple`, `dict` (for dictionary), `set` and `bool` (for Boolean True/False).

str

Strings are used in Python to record text information, such as names.

```
mystr = 'this is a string'
mystr_double = "also a string but using double quotes"
len(mystr) # Returns the number of elements in the string
```

String Indexing

```
mystr[0] # Returns the first element at index 0
Out: t

mystr[1:] # Returns the second element till the end
mystr[:5] # Returns all the elements up to the 5th index
mystr[:-1] # Grabs everything but the last letter
mystr[::2] # Grabs everything with step size 2
mystr[::-1] # Returns the string backwards/reversed
```

Concatenating

```
mystr + ' concatenate me!'
Out: 'this is a string concatenate me!'
```

Basic Built-in String methods

```
mystr.upper() # Upper case all elements of string
mystr.lower() # Lower case all elements of string
mystr.split() # Split a string by blank space (default)
```

String Formatting: .format & f-strings

```
name = 'Alex'
print('My name is {}'.format(name))# .format-method
Out: "My name is Alex."

print(f"He said his name is {name}.") # f-strings method
Out: "He said his name is Alex."
```

list

Lists can be thought of the most general version of a sequence in Python. Unlike strings, they are mutable, meaning the elements inside a list can be changed!

```
my_list = ['A string',23,100.232,'o'] # This is a list
len(my_list) # Returns the number of elements in the list
```

Basic List Methods

```
my_list = my_list + ['add new item']
my_list.append('append me!') # Append an item to my_list
my_list.pop(0) # Pop off the 0 indexed item
my_list.revers() # Use reverse to reverse order (this is permanent!)
my_list.sort() # Use sort to sort the list (in this case alphabetical order
    , but for numbers it will go ascending)
```

Nested Lists

```
lst_1=[1,2,3]
lst_2=[4,5,6]
lst_3=[7,8,9]
matrix = [lst_1,lst_2,lst_3]

matrix
Out: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

matrix[0]
Out: [1, 2, 3]

matrix[0][0]
Out: 1
```

List Comprehensions

```
# Build a list comprehension by deconstructing a for loop within a []
first_col = [row[0] for row in matrix]
# "row" is a random variable name assigned to loop through each element in
    "matrix" using a for-loop. If we want to execute an operation on this
    variable "row" (e.g. grab the first index of the element row[0]), we
    add this operation in front of the for-loop
first_col
Out:[1, 4, 7]
```

dict

A dictionary, hash table or mapping is a collection of objects that are stored by a key, unlike a sequence that stored objects by their relative position. This is an important distinction, since mappings won't retain order since they have objects defined by a key.
A Python dictionary consists of a key and then an associated value. That value can be almost any Python object.

```
my_dict = {'key1':'value1','key2':'value2'}
my_dict['key2']
Out: 'value2'
```

Constructing a Dictionary

```
# A dictionary can contain different types of information
my_dict =
{'key1':123,'key2':[12,23,33],'key3':['str0','str1','str2']
        my_dict['key3'][0].upper # Indexing and methods on dictionary
        Out: 'STR0'

        d = {} # Create a new dictionary
        d['animal'] = 'Dog' # Create a new key through assignment
        d['answer'] = 42 # Can do this with any object
        d
        Out: {'animal': 'Dog', 'answer': 42}
```

A few Dictionary Methods

```
        d = {'key1':1,'key2':2,'key3':3}
        d.keys() # Method to return a list of all keys
        Out: dict_keys(['key1', 'key2', 'key3'])

        d.values() # Method to grab all values
        Out: dict_values([1, 2, 3])

        d.items() # Method to return tuples of all items
        Out: dict_items([('key1', 1), ('key2', 2), ('key3', 3)])
```

tuple

In Python tuples are very similar to lists, however, unlike lists they are immutable meaning they can not be changed. Once a tuple is made we can not add to it. Indexing and slicing works just like we did in lists.
You would use tuples to present things that shouldn't be changed, such as days of the week, or dates on a calendar.

```
        t = ('one',2) # Create Tuple (can also mix object types)
```

Basic Tuple methods

```
        t.index('one') # Returns the index position of the defined value
        t.count('one') # Returns the number of times a value appears
```

set

Sets are an unordered collection of unique elements. We can construct them by using the set() function.

```
x = set() # Creating a set
x.add(2) # We can add a unique element to a set (only once)
Out: {2}

list1 = [1,1,2,2,3,4,5,6,1,1]
set(list1) # Returns the unique element from list1
Out: {1, 2, 3, 4, 5, 6}
```

bool

Python comes with Booleans (with predefined True and False displays that are basically just the integers 1 and 0). It also has a placeholder object called None. This can be used for an object that we don't want to reassign yet.

```
a = True
b = False
c = None
```

Comparison Operators

```
a == b # Is a equal to b?
Out: False # No

1 < 2 # Is 1 smaller than 2?
Out: True # Yes
```

2 PYTHON STATEMENTS

2.1 IF, ELIF, ELSE STATEMENTS

if, **elif**, **else** Statements in Python allow us to tell the computer to perform alternative actions based on a certain set of results. Verbally, we tell the computer: "Hey if this case happens, perform some action. Else, if another case happens, perform some other action. Else, if none of the above cases happened, perform this action."

```
person = 'George'
if person == 'Sammy':
print('Welcome Sammy!')
elif person =='George':
print('Welcome George!')
else:
print("Welcome, what's your name?")

Out: Welcome George!
```

2.2 FOR LOOPS

A for loop acts as an iterator in Python; it goes through items that are in a sequence or any other iterable item. Objects that we've learned about that we can iterate over include strings, lists, tuples, and even built-in iterables for dictionaries, such as keys or values.

```
list1 = [1,2,3,4,5,6,7,8,9]
list_even = [] # Empty list
for num in list1: # Each iteration we will grab an element from list1 and
    we call these elements "num" (random name)
if num%2 == 0: # If num is even
list_even.append(num) # Append to list_even
print("{} is even".format(num))
else: # If num is not even
print("{} is odd".format(num))
print(list_even) # Print list of even numbers
```

For loops and dictionaries:

```
d = {'k1':1,'k2':2,'k3':3}
# Dictionary unpacking
for k,v in d.items(): # k = key, v = value
print(k)
print(v)
Out: k1 1 k2 2 k3 3
```

2.3 WHILE LOOPS

The while statement in Python is one of most general ways to perform iteration. A while statement will repeatedly execute a single statement or group of statements as long as the condition is true. The reason it is called a 'loop' is because the code statements are looped through over and over again until the condition is no longer met.

We can use **break**, **continue**, and **pass** statements in our loops to add additional functionality for various cases. The three statements are defined by:

- **break**: Breaks out of the current closest enclosing loop.
- **continue**: Goes to the top of the closest enclosing loop.
- **pass**: Does nothing at all.

```
x = 0
boundary = 8 # Random boundary condition
while x < 10:
print('x is currently: ',x)
print(' x is still less than 10, adding 1 to x')
x+=1 # x=x+1
if x==boundary:
print('Boundary Condition met')
break # Program stops when x == boundary
else:
print('continuing...')
continue
```

We can also create an endless loop using the **while** `True` function. Make sure to include a **break** inside the loop, otherwise it will keep running forever.

```
i = 0
while True:
i += 1
if i == 12:
break
print (i)
```

2.4 USEFUL OPERATORS

range

The range function allows you to quickly generate a list of integers, this comes in handy a lot, so take note of how to use it! There are 3 parameters you can pass, a start, a stop, and a step size. Default step size is 1.

```
range(0,51,10) # range 0 up to 51 with a step size 10
Out: range(0,51,10) # Returns generator function
list(range(0,51,10))
Out: [0, 10, 20, 30, 40, 50]
```

enumerate

A very useful function to use with for loops. An example could be to keep track of how many loops you have gone through.

```
        # Index counting without the enumerate function
        index_count = 0
        for letter in 'abcde':
        print("Index:{} Letter:{}".format(index_count,letter))
        index_count += 1

        # Index counting using enumerate
        for i,letter in enumerate('abcde'):
        print("Index:{} Letter:{}".format(i,letter))
        Out: Index:0 Letter:a
        Index:1 Letter:b
        ...
```

zip

This function allows us to quickly create a list of tuples by "zipping" up together two lists.

```
        mylist1 = [1,2,3,4,5]
        mylist2 = ['a','b','c','d','e']

        zip(mylist1,mylist2) # returns generator function
        list(zip(mylist1,mylist2))
        Out: [(1, 'a'), (2, 'b'), (3, 'c'), ...]
```

To use the generator function we can use a for loop:

```
        for item1, item2 in zip(mylist1,mylist2):
        print('{},{}'.format(item1,item2))
        Out: 1,a
        2,b
        ...
```

in

The function is used in for loops. However, it can also be used to quickly check if an item is in a list.

```
        'x' in ['x','y','z']
        Out: True
```

min max

Quickly check the minimum or maximum of a list with these functions.

```
        mylist = [10,20,30,40,100]
        min(mylist)
        Out: 10
```

`random`

Python comes with a built in random library with a lot of built in functions.

```
from random import shuffle
mylist = [10,20,30,40,100]
shuffle(mylist) # Shuffles list "in place" meaning it won't return anything
    , instead it will effect the list passed
Out: [40, 10, 100, 30, 20]

from random import randint
randint(0,100) # Return random integer in range [a, b], including both end
    points.
Out: 23
```

`input`

Requests an input from the user.

```
name = input('Enter your name: ')
Enter your name: Alex
print("Hello {}".format(name))
Out: Hello Alex

# To let the user input numbers we include int()
x = int(input('Enter your favorite number here: '))
Enter your favorite number here: 7
print("Your favorite number is {}".format(x))
Out: Your favorite number is 7
```

2.5 LIST COMPREHENSION

In addition to sequence operations and list methods, Python includes a more advanced operation called a list comprehension.
List comprehensions allow us to build out lists using a different notation. You can think of it as essentially a one line for loop built inside of brackets.

```
# Some examples of list comprehension are presented below:
# Grab every letter in a string
lst = [x for x in 'word']
Out: ['w', 'o', 'r', 'd']

# Square numbers in range and turn into list
lst = [x**2 for x in range(0,11)]
Out: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

# Check for even numbers in a range
lst = [x for x in range(11) if x % 2 == 0]
Out: [0, 2, 4, 6, 8, 10]

```

```
14          # Convert Celsius to Fahrenheit
15          celsius = [0,10,20.1,34.5]
16          fahrenheit = [((9/5)*temp + 32) for temp in celsius ]
17          Out: [32.0, 50.0, 68.18, 94.1]
18
19          # Nested comprehension
20          lst = [ x**2 for x in [x**2 for x in range(11)]]
21          Out: [0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561, 10000]
```

3

CHALLENGE 1: A SIMPLE GUESSING GAME

The Challenge:
Write a program that picks a random integer from 1 to 100, and has players guess the number. The rules are:

1. If a player's guess is less than 1 or greater than 100, say "OUT OF BOUNDS"
2. On a player's first turn, if their guess is within 10 of the number, return "WARM!". If the guess is further than 10 away from the number, return "COLD!"
3. On all subsequent turns, if a guess is closer to the number than the previous guess return "WARMER!". If the guess is further from the number than the previous guess, return "COLDER!"
4. When the player's guess equals the number, tell them they've guessed correctly and how many guesses it took!

```
1   import random
2   num = random.randint(1,100)
3
4   print("WELCOME TO GUESS ME!")
5   print("I'm thinking of a number between 1 and 100")
6   print("If your guess is more than 10 away from my number, I'll tell you you
        're COLD")
7   print("If your guess is within 10 of my number, I'll tell you you're WARM")
8   print("If your guess is further than your most recent guess, I'll say you'
        re getting COLDER")
9   print("If your guess is closer than your most recent guess, I'll say you're
         getting WARMER")
10  print("LET'S PLAY!")
11
12  guesses = [0] # Hint: zero is a good placeholder value. It's useful because
         it evaluates to "False"
13
14  while True: # Endless loop until break
15  guess = int(input("I'm thinking of a number between 1 and 100.\n  What is
        your guess? "))
16
17  if guess < 1 or guess > 100:
18  print('OUT OF BOUNDS! Please try again: ')
19  continue
20
21
22  # here we compare the player's guess to our number
```

```
23          if guess == num:
24          print(f'CONGRATULATIONS, YOU GUESSED IT IN ONLY {len(guesses)} GUESSES!!')
25          break
26
27          # if guess is incorrect, add guess to the list
28          guesses.append(guess)
29
30          # when testing the first guess, guesses[-2]==0, which evaluates to False
31          # and brings us down to the second section
32          if guesses[-2]:
33          if abs(num-guess) < abs(num-guesses[-2]):
34          print('WARMER!')
35          else:
36          print('COLDER!')
37
38          else:
39          if abs(num-guess) <= 10:
40          print('WARM!')
41          else:
42          print('COLD!')
```

4 METHODS AND FUNCTIONS

4.1 METHODS

We've already seen a few example of methods when learning about Object and Data Structure Types in Python. Methods are essentially functions built into objects. You can use the `help()` function to get more information about a method.
Methods are in the form:

```
object.method(arg1,arg2,etc...)
```

Each object has corresponding methods assigned to it. Below an example is given of the various methods that belong to the list object.

`.append(x)`	Adds an item x to the end of the list.
`.count(x)`	Counts the number of occurrences of element x in a list text
`.extend(...)`	Extends the list by appending all the items located inside (...) to the list. This allows you to join two lists together.
`.insert(i, x)`	Inserts an item at a given position. The first argument is the index of the element before which to insert.
`.pop([i])`	Removes the item at the given position in the list, and returns it. If no index is specified, `.pop()` removes and returns the last item in the list.
`.remove(x)`	Removes the first item from the list that has a value of x. Returns an error if there is no such item.
`.reverse()`	Reverses the elements of the list in place.
`.sort()`	Sorts the items of the list in place.
`.clear()`	Removes all items from the list.
`.copy()`	Returns a shallow copy of the list.

4.2 FUNCTIONS

Formally, a function is a useful device that groups together a set of statements so they can be run more than once. They can also let us specify parameters that can serve as inputs to the functions.
On a more fundamental level, functions allow us to not have to repeatedly write the same code again and again. If you remember back to the lessons on strings and lists, remember that we used a function len() to get the length of a string. Since checking the length of a sequence is a common task you would want to write a function that can do this repeatedly at command.
It should be noted that as soon as a function returns something, it shuts down. A function can deliver multiple print statements, but it will only obey one return.

The function's syntax in python:

```
def name_of_function(arg1,arg2): # Lower case letters!
'''
This is where the function's Document String  (docstring) goes --> help(
    name_of_function) will return this information
'''
# Do stuff here
# Return desired result
```

A simple example of a function that adds two numbers:

```
def add_num(num1,num2):
return num1+num2

add_num(1,8)
Out: 9
```

A function that checks if a number is prime:

```
import math

def is_prime(num):
'''
Better method of checking for primes.
'''
if num % 2 == 0 and num > 2:
return False
for i in range(3, int(math.sqrt(num)) + 1, 2):
if num % i == 0:
return False
return True

is_prime(18)
Out: False
```

4.3 LAMBDA EXPRESSIONS, MAP AND FILTER

The `map` function allows you to "map" a function to an iterable object. That is to say you can quickly call the same function to every item in an iterable, such as a list.

```
def square(num):
return num**2

my_nums = [1,2,3,4,5] # An arbitrary list with numbers
list(map(square,my_nums)) # Execute the function on a list
Out: [1, 4, 9, 16, 25]
```

The `filter` function returns an iterator yielding those items of iterable for which function(item) is `True`. Meaning you need to filter by a function that returns either `True` or `False`. Then passing that into filter (along with your iterable) and you will get back only the results that would return `True` when passed to the function.

```
        def check_even(num):
        return num % 2 == 0

        nums = [0,1,2,3,4,5,6,7,8,9,10]
        list(filter(check_even,nums))
        Out: [0, 2, 4, 6, 8, 10]
```

A `lambda` expression allows us to create "anonymous" functions. This basically means we can quickly make ad-hoc functions without needing to properly define a function using def.
So why use this? Many function calls need a function passed in, such as map and filter. Often you only need to use the function you are passing in once, so instead of formally defining it, you just use the lambda expression.

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

        list(map(lambda num: num ** 2, my_nums))
        Out: [1, 4, 9, 16, 25]
```

4.4 THE THREE RULES OF SCOPE

When you create a variable name in Python the name is stored in a *name-space*. Variable names also have a scope, the scope determines the visibility of that variable name to other parts of your code.
This idea of scope in your code is very important to understand in order to properly assign and call variable names. In simple terms, the idea of scope can be described by 3 general rules:

- Name assignments will create or change local names by default.
- Name references search (at most) four scopes (LEGB Rule), these are:
    1. **L**ocal — Names assigned in any way within a function (def or lambda), and not declared global in that function. If not found locally Python will search in the enclosing functions.
    2. **E**nclosing functions — Names in the local scope of any and all enclosing functions (def or lambda), from inner to outer. If not found locally Python will search globally.
    3. **G**lobal — Names assigned at the top-level of a module file, or declared global in a def within the file. If not found locally Python will search for built-in names.
    4. **B**uilt-in — Names preassigned in the built-in names module : open, range, SyntaxError,...
- Names declared in global and nonlocal statements map assigned names to enclosing module and function scopes.

4.5 *ARGS AND **KWARGS

*args

When a function parameter starts with an asterisk*, it allows for an arbitrary number of arguments, and the function takes them in as a tuple of values.

```
        def myfunc(a,b):
        return sum((a,b))*.05
        myfunc(40,60,20)
        Out: Error

        # An inefficient solution:
        def myfunc(a=0,b=0,c=0,d=0,e=0):
        return sum((a,b,c,d,e))*.05

        myfunc(40,60,20)
        Out: 6.0

        # The use of *args
        def myfunc(*args):
        return sum(args)*.05

        myfunc(40,60,20)
        Out: 6.0
```

**kwargs

Similarly, Python offers a way to handle arbitrary numbers of keyworded arguments. Instead of creating a tuple of values, **kwargs builds a dictionary of key/value pairs.

```
        def myfunc(**kwargs):
        if 'fruit' in kwargs:
        print(f"My favorite fruit is {kwargs['fruit']}")
        else:
        print("I don't like fruit")
        myfunc(fruit='pineapple')
        Out: My favorite fruit is pineapple
```

We can also combine `*args` and `**kwargs` in a single function:

```
def myfunc(*args, **kwargs):
if 'fruit' and 'juice' in kwargs:
print(f"I like {' and '.join(args)} and my favorite fruit is {kwargs['fruit
    ']}")
print(f"May I have some {kwargs['juice']} juice?")
else:
pass

myfunc('eggs','spam',fruit='cherries',juice='orange')
Out: I like eggs and spam and my favorite fruit is cherries
May I have some orange juice?
```

Placing keyworded arguments ahead of positional arguments raises an exception:

```
myfunc(fruit='cherries',juice='orange','eggs','spam')
Out: SyntaxError: positional argument follows keyword argument
```

4.6 BUILT-IN FUNCTIONS

map

map() is a built-in Python function that takes in two or more arguments: a function and one or more iterables, in the form:
map(function, iterable, ...) map() returns an iterator - that is, map() returns a special object that yields one result at a time as needed. We can visualize the data using **list**:

```
def fahrenheit(celsius): # Fahrenheit - celcius conversion
return (9/5)*celsius + 32

temps = [0, 22.5, 40, 100] # List with temps


F_temps = map(fahrenheit, temps) # Insert temps into function

list(F_temps) # Show
Out: [32.0, 72.5, 104.0, 212.0]
```

reduce

The function reduce(function, sequence) continually applies the function to the sequence. It then returns a single value.
If seq = [s1, s2, s3, ... , sn], calling reduce(function, sequence) works like this:

- At first the first two elements of seq will be applied to function, i.e. func(s1,s2)
- The list on which reduce() works looks now like this: [function(s1, s2), s3, ... , sn]
- In the next step the function will be applied on the previous result and the third element of the list, i.e. function(function(s1, s2),s3)

- The list looks like this now: [function(function(s1, s2),s3), ... , sn]
- It continues like this until just one element is left and return this element as the result of reduce()

```
from functools import reduce

lst =[47,11,42,13]
reduce(lambda x,y: x+y,lst) # for x,y in lst: return x+y
Out: 113
```

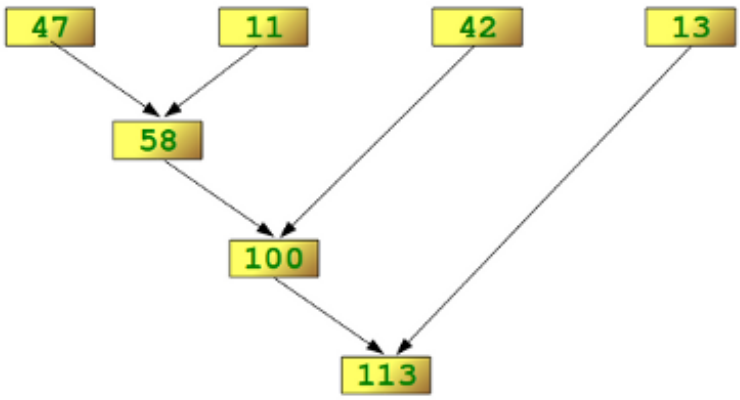


Figure 1: Visualization of reduce()

filter

The function filter(function, list) offers a convenient way to filter out all the elements of an iterable, for which the function returns True.

```
def even_check(num): # Function returns even numbers
if num%2 ==0:
return True

lst =range(20)

list(filter(even_check,lst)) # Using pre-defined function
Out: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

list(filter(lambda x: x%2==0,lst)) # Using lambda function
Out: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

zip

zip() makes an iterator that aggregates elements from each of the iterables. It is defined by the shortest iterable length and returns an iterator of tuples.

```
x = [1,2,3]
y = [4,5,6,7,8]

# Zip the lists together
list(zip(x,y))
Out: [(1, 4), (2, 5), (3, 6)]
```

enumerate

Enumerate allows you to keep a count as you iterate through an object. It does this by returning a tuple in the form (count,element).
Enumerate() can be used as a tracker:

```
lst = ['a','b','c']

for count,item in enumerate(lst):
if count >= 2:
break
else:
print(item)
Out: a
b
```

enumerate() can also take an optional "start" argument to override the default value of zero:

```
months = ['March','April','May','June']

list(enumerate(months,start=3))
Out: [(3, 'March'), (4, 'April'), (5, 'May'), (6, 'June')]
```

all any

all() and any() are built-in functions in Python that allow us to conveniently check for boolean matching in an iterable. all() will return True if all elements in an iterable are True. any() will return True if any of the elements in the iterable are True.

```
lst = [True,True,False,True]
all(lst) # Check if all True
Out: False # Not all elements are True.

any(lst)
Out: True
```

complex

complex() returns a complex number with the value real + imag*1j or converts a string or number to a complex number.

```
1 complex(2,3) # Create 2+3j
2 complex('12+2j') # Create 12+2j using a string
```

5 OBJECT ORIENTED PROGRAMMING (OOP)

Object-oriented Programming, or OOP for short, is a programming paradigm which provides a means of structuring programs so that properties and behaviors are bundled into individual objects.
In Python, everything is an object. With `type()` we can check the type of an object (e.g. `int`, `list`, `dict`).

5.1 CREATING CUSTOM OBJECT TYPES AND ATTRIBUTES

User defined objects are created using the `class` keyword. The class is a blueprint that defines the nature of a future object.
By convention we give classes a name that starts with a capital letter. Inside a class we can assign attributes and methods. An attribute is a characteristic of an object. A method is an operation we can perform with the object.

Below we create a class `Employee` and two instances of the class `emp_1` and `emp_2`. The class itself only passes.

```
1 class Employee:
2     pass
3
4 emp_1 = Employee() # A unique instance of the Employee class
5 emp_2 = Employee() # A second instance of the Employee class
```

We can assign attributes (e.g. name and salary) to the employees and fill the values in manually.

```
1 emp_1.name = 'Hans' # name attribute added
2 emp_1.salary = 5000 # salary attribute added
3 emp_2.name = 'Fred'
4 emp_2.salary = 3500
5
6 print(emp_1.salary)
7 Out: 5000
```

This is very inefficient and there is no point in using objects this way. What we can do is use a special method called `__init__(self, arg1, arg2, ...)`.
When we create methods within a class, the instance (e.g. `emp_1 = Employee()`) is automatically received as the first argument. This argument is by convention called `self`. We can also add other arguments that we want to include such as, `name` and `salary`.

```
class Employee:

    def __init__(self,name,salary):
        self.name = name # Attribute 1
        self.salary = salary # Attribute 2

emp_1 = Employee('Hans',5000) # Unique instance 1
emp_2 = Employee('Fred',3500) # Unique instance 2
```

Now we can treat the instances as objects. For example, `emp_2.`(press TAB), shows the attributes (`name salary`) that can be selected from the `emp_2` object, as it is part of the Employee class.

```
emp_1.salary
Out: 5000
emp_2.name
Out: 'Fred'
```

It is important to note that for each object, the argument **`self`**, is replaced by the object name (e.g. `emp_1`) of which an instance was made. This means for `emp_1` that **`self`**`.name = name` is in fact `emp_1.name = 'Hans'`, because `'Hans'` was entered for the `name` argument in:

```
emp_1 = Employee('Hans',5000)
```

5.2 METHODS

Methods are functions defined inside the body of a class. They are used to perform operations with the attributes of our objects. Methods are a key concept of the OOP paradigm. They are essential to dividing responsibilities in programming, especially in large applications.

```
class Circle:
pi = 3.14
    # Circle gets instantiated with a radius (default is 1)
    def __init__(self, radius=1):
        self.radius = radius

    # Method for getting Circumference
    def getCircumference(self):
        return self.radius * self.pi * 2

c = Circle()

print('Radius is: ',c.radius)
print('Circumference is: ',c.getCircumference())

Out: Radius is:  1
     Circumference is:  6.28
```

5.3 INHERITANCE

Inheritance is a way to form new classes using classes that have already been defined. The newly formed classes are called derived classes, the classes that we derive from are called base classes. Important benefits of inheritance are code reuse and reduction of complexity of a program. The derived classes (descendants) override or extend the functionality of base classes (ancestors).

```
class Animal: # Base class
    def __init__(self):
        print("Animal created")

    def whoAmI(self):
        print("Animal")

    def eat(self):
        print("Eating")

# New class Dog will inherit the Animal (base) class
class Dog(Animal): # Inheritance
    def __init__(self):
        Animal.__init__(self) # Inheritance
        print("Dog created")

    def whoAmI(self):
        print("Dog")

    def bark(self):
        print("Woof!")

d = Dog()
Out: Animal created # Originates from the Animal class
     Dog created

d.whoAmI() # Derived class modifies behavior of base class
Out: Dog

d.eat() # Derived class inherits functionality of base class
Out: Eating

d.bark() # Derived class extends functionality of base class
Out: Woof!
```

5.4 POLYMORPHISM

In Python, polymorphism refers to the way in which different object classes can share the same method name, and those methods can be called from the same place even though a variety of different objects might be passed in.

```
class Dog:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return self.name+' says Woof!'

class Cat:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return self.name+' says Meow!'

niko = Dog('Niko')
felix = Cat('Felix')

print(niko.speak())
print(felix.speak())

Out: Niko says Woof!
     Felix says Meow!
```

Both objects, `Dog` and `Cat`, share the method `.speak`. With polymorphism we can this method using a for loop or a function:

```
# Polymorphism for loop method
for pet in [niko,felix]:
    print(pet.speak())

Out: Niko says Woof!
     Felix says Meow!


# Polymorphism function method
def pet_speak(pet):
    print(pet.speak())

pet_speak(niko)
pet_speak(felix)

Out: Niko says Woof!
     Felix says Meow!
```

A more common practice is to use abstract classes and inheritance. An abstract class is one that never expects to be instantiated. For example, we will never have an Animal object, only Dog and Cat objects, although Dogs and Cats are derived from Animals:

```
class Animal:
    def __init__(self, name):    # Constructor of the class
        self.name = name

    def speak(self):             # Abstract method, defined by convention only
        raise NotImplementedError("Subclass must implement abstract method")


class Dog(Animal):

    def speak(self):
        return self.name+' says Woof!'

class Cat(Animal):

    def speak(self):
        return self.name+' says Meow!'

fido = Dog('Fido')
isky = Cat('Isky')

print(fido.speak())
print(isky.speak())

Out: Fido says Woof!
     Isky says Meow!
```

5.5 SPECIAL METHODS

Classes in Python can implement certain operations with special method names. These methods are not actually called directly but by Python specific language syntax. These special methods are defined by their use of underscores (`__str__()`, `__len__()`, `__del__()`). They allow us to use Python specific functions (`print()`, `len()`, `del()`) on objects created through our class. A book class example is used to demonstrate these special methods.

```
1  class Book:
2      def __init__(self, title, author, pages):
3          print("A book is created")
4          self.title = title
5          self.author = author
6          self.pages = pages
7
8      def __str__(self):
9          return "Title: %s, author: %s, pages: %s" %(self.title, self.author, self.
               pages)
10
11     def __len__(self):
12         return self.pages
13
14     def __del__(self):
15         print("A book is destroyed")
16
17
18 book = Book("Python Rocks!", "Jose Portilla", 159)
19
20 #Special Methods
21 print(book)
22 print(len(book))
23 del book
24
25 Out: A book is created
26      Title: Python Rocks!, author: Jose Portilla, pages: 159
27      159
28      A book is destroyed
```

6 CHALLENGE 2: MY FIRST BANK ACCOUNT

The Challenge:
Create a bank account class that has two attributes:

- owner
- balance

and two methods:

- deposit
- withdraw

As an added requirement, withdrawals may not exceed the available balance.
Instantiate your class, make several deposits and withdrawals, and test to make sure the account can't be overdrawn.

```
class Account:
    def __init__(self,owner,balance=0):
        self.owner = owner
        self.balance = balance

    def __str__(self):
        return f'Account owner:   {self.owner}\nAccount balance: ${self.balance}'

    def deposit(self,dep_amt):
        self.balance += dep_amt
        print('Deposit Accepted')

    def withdraw(self,wd_amt):
        if self.balance >= wd_amt:
            self.balance -= wd_amt
            print('Withdrawal Accepted')
        else:
            print('Funds Unavailable!')


# Instantiate the class and you are good to go!
acct1 = Account('Tim',1000)
```

7 ERROR HANDLING

7.1 TRY, EXCEPT AND FINAL

`try`: Used when we could expect errors/exceptions to occur in a block of code.
`except`: Handling implementation for the errors/exceptions that occured in the try-block
`finally`: Block of code that always prints
The example below is a function that keeps asking for an integer until it is entered correctly:

```
1  def askint():
2      while True:
3          try:
4              val = int(input("Please enter an integer: "))
5          except:
6              print("Looks like you did not enter an integer!")
7              continue
8          else:
9              print("Yep that is an integer!")
10             break
11         finally:
12             print("Finally, I executed!")
13             print(val)
14
15 askint()
16 Out: Please enter an integer: five
17      Looks like you did not enter an integer!
18      Finally, I executed!
19      Please enter an integer: four
20      Looks like you did not enter an integer!
21      Finally, I executed!
22      Please enter an integer: 3
23      Yep that is an integer!
24      Finally, I executed!
```

7.2 UNIT TESTING

A good way to test your code is to write tests that send sample data to your program, and compare what's returned to a desired outcome. Two such tools are available from the standard library:

- unittest
- doctest

To evaluate your code in Spyder using a built-in test, press F8.

8 PYTHON MODULES AND PACKAGES

8.1 IMPORTING MODULES

Modules in Python are simply Python files with the .py extension, which implement a set of functions. Modules are imported using the `import` command. Full list of built-in modules in the standard library: https://docs.python.org/3/py-modindex.html.

```
1  import math
2  print(dir(math))
3  Out: ['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', '
       acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', '
       cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', '
       floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', '
       isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2'
       , 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh',
       'tau', 'trunc']
4
5  # Use help formore information about a function
6  help(math.ceil) # Use help formore information about a function
7  Out: ceil(...)
8       ceil(x)
9
10      Return the ceiling of x as an Integral.
11      This is the smallest integer >= x.
```

8.2 WRITING MODULES AND PACKAGES

Module
A module is just a python file that you can use in your program by importing it using the `import` or `from module import var,function` statements.
To create a module of your own, simply create a new .py file with the module name, and then import it using the Python file name (without the .py extension) using the import command.

Package
A package is basically a way to organize your code. Packages let you import directories on your computer into your programs using the `import` or `from import` statements. A directory on your system can become a package if it contains an empty `__init__.py` file. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.

If we create a directory called `foo`, which marks the package name, we can then create a

module inside that package called `bar`. We also must not forget to add the `_init_.py` file inside the `foo` directory. To use the module `bar`, we can import it in two ways:

```
import foo.bar
from foo import bar
```

8.3 NAME AND MAIN

To determine whether the function of an imported module is being used as an import, or if you are using the original .py file of that module we an use the following line of code:

```
if __name__ == "__main__":
```

```
# file one.py
def func():
    print("func() in one.py")

print("top-level in one.py")

if __name__ == "__main__":
    print("one.py is being run directly")
else:
    print("one.py is being imported into another module")


# file two.py
import one

print("top-level in two.py")
one.func()

if __name__ == "__main__":
    print("two.py is being run directly")
else:
    print("two.py is being imported into another module")


RUN: one.py
Out: top-level in one.py

RUN: two.py
Out: top-level in one.py
     one.py is being imported into another module
     top-level in two.py
     func() in one.py
     two.py is being run directly
```

9 DECORATORS AND GENERATORS

9.1 PYTHON DECORATORS

Decorators can be thought of as functions which modify the functionality of another function. They help to make your code shorter and more "Pythonic". The idea of a decorator is presented below:

```
def new_decorator(func):

    def wrap_func():
        print("Code would be here, before executing the func")

    func()

    print("Code here will execute after the func()")

    return wrap_func

def func_needs_decorator():
    print("This function is in need of a Decorator")

func_needs_decorator = new_decorator(func_needs_decorator)
Out: Code would be here, before executing the func
     This function is in need of a Decorator
     Code here will execute after the func()
```

A decorator simply wrapped the function and modified its behavior. We can rewrite this code using the @ symbol, which is what Python uses for Decorators:

```
@new_decorator
def func_needs_decorator():
    print("This function is in need of a Decorator")
Out: Code would be here, before executing the func
     This function is in need of a Decorator
     Code here will execute after the func()
```

9.2 PYTHON ITERATORS AND GENERATORS

Generators allow us to generate as we go along, instead of holding everything in memory by using the `yield` function. The main advantage here is that instead of having to compute an entire series of values up front, the generator computes one value and then suspends its activity awaiting the next instruction. This feature is known as state suspension.

```
# Generator function for the cube of numbers (power of 3)
def gencubes(n):
    for num in range(n):
        yield num**3

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

Out: 0
     1
     8
     27
     64
```

The `next()` function allows us to access the next element in a sequence:

```
def simple_gen():
    for x in range(3):
        yield x

g = simple_gen() # Assign simple_gen
print(next(g))
Out: 0
print(next(g))
Out: 1
print(next(g))
Out: 2
print(next(g))
Out: StopIteration # All the values have been yielded
```

Using the `iter()` function we can iterate over an iterable object like a string:

```
s = 'hello' # A string is an iterable object

next(s) # We can not directly iterate over a string
Out: TypeError: 'str' object is not an iterator

# Iterable object converted into an iterator using iter()
s_iter = iter(s)
next(s_iter)
Out: 'h'
next(s_iter)
Out: 'e'
```

10 GRAPHICAL USER INTERFACE (GUI)

10.1 GETTING STARTED WITH PYQT5

PyQt5 allows easy design of the GUI (without functionality) by means of a drag and drop software program called *Qt Designer*.
Once a design was made the following steps need to be taken:
- Save file as *myfile.ui*
- Open command prompt (CMD) and locate the folder in which the file is saved (TIP: browse to the folder and type CMD in the browser bar. This opens the CMD in that specific folder)
- Enter: `pyuic5 -x myfile.ui -o myfile.py` (*-x = execute, -o = output file*)

10.2 GETTING STARTED WITH TKINTER

Import everything from the `tkinter` class:

```
from tkinter import *
```

The following two lines of code create a GUI window and make sure that the program keeps running:

```
root = Tk() # Creates an empty main window

# GUI code here

root.mainloop() # mainloop runs the root (GUI Object) and makes sure it is
    continuously shown to the user
```

Here we define what widgets/objects we would like to use in our GUI:

```
# Labels are just assigned locations for strings
label_1 = Label(root, text="Name:")
label_2 = Label(root, text="Password:")
entry_1 = Entry(root)
entry_2 = Entry(root)
check_1 = Checkbutton(root, text="Remember Password")
```

We can position a widget using `.pack()` or `.grid()`. The grid method uses cells (like in Excel) defined as rows and columns and is presented below:

```
label_1.grid(row=0, sticky=E) # E = East = align right
label_2.grid(row=1, sticky=E)
entry_1.grid(row=0,column=1)
entry_2.grid(row=1,column=1)
check_1.grid(columnspan=2) # Takes two cells and merges them
```

The result of the code below is a simple login window with a clickable checkbox:

```
from tkinter import * # Import everything from tkinter class

root = Tk() # Empty (main) window

# Widgets defined
label_1 = Label(root, text="Name:")
label_2 = Label(root, text="Password:")
entry_1 = Entry(root)
entry_2 = Entry(root)
check_1 = Checkbutton(root, text="Remember Password")

# Widgets selected and placed in root (window)
label_1.grid(row=0, sticky=E)
label_2.grid(row=1, sticky=E)
entry_1.grid(row=0,column=1)
entry_2.grid(row=1,column=1)
check_1.grid(columnspan=2)

root.mainloop()
```

10.3 .PY TO .EXE

Using `pyinstaller` in the command window we can create a standalone executable. These are some easy steps that can be used:

- copy the python script (e.g. myscript.py) script to a folder (e.g. myfolder) on the desktop
- open command prompt cmd and navigate to this folder on the desktop desktop: cd desktop –ENTER– cd myfolder –ENTER–
- pyinstaller -w - F myscript.py –ENTER–
- myscript.exe is now found in the dist folder