openanalysis Documentation

Release 1.0-rc

OpenWeavers

Aug 28, 2017

Table of contents

Part I

The Python Language

1 Introduction to Python

1.1 What is Python?

Python is a widely used high-level programming language for general-purpose programming, created by Guido Van Rossum and first released in 1991. An interpreted language, Python has a design philosophy which emphasizes code readability (notably using whitespace indentation to delimit code blocks rather than curly brackets or keywords), and a syntax which allows programmers to express concepts in fewer lines of code than might be used in languages such as C++ or Java. The language provides constructs intended to enable writing clear programs on both a small and large scale.

1.2 Prerequisites

We assume that you have:

- Basic understanding of what computer does and what computer programs do
- Knowledge of `C` Language
- Knowledge of Object Oriented Concepts like Objects,Classes,Inheritance,Polymorphism, etc...
- Knowledge of any Object Oriented Language like `C++`,`Java` or `C#`

These Software must be installed to follow the Language tutorial.

- Python 3 (Download from Python Website[1] or `apt install` it)
- IPython 3 (An interactive Python Shell, Download from IPython Project Site[2] or `apt install` it)

We will be using several libraries throughout the tutorial. They can be installed with `pip` (or `pip3`) as `pip install <library-name>`. Following is a list of libraries that has to be installed to follow the tutorial.

- `matplotlib` (For visualizing data sets)
- `numpy` (For faster array operations)
- `networkx` (Provides Graph Data Structure) `-` `OpenAnalysis` (An open source package to analyse and visualise algorithms and data structures)

[1] https://www.python.org/
[2] https://ipython.org/

WARNING

The Python executable is `python` or `python3` depending on your type of installation. Use `--version` flag with `python` executable to determine the version of Python installed on your system

1.3 Your First Program

As a tradition, we start with a program to display `Hello World` on the Console Screen (the `stdout`)

Open the Interative Python Shell by typing `ipython`(or `ipython3`) from the terminal. If everything goes well, you will get a prompt where you can enter statements and see the effect of it. Now enter the following statement to get started

```
In [1]: print("Hello World")
```

```
Hello World
```

Congrats! You have successfully executed your first statement in Python. In fact there are many ways to execute python statements. Interactive Console is one of them. You can also pack the statements into single file, whose name ends with an extension `.py`, and call `python` (or `python3`) to execute them. We will also check this method to execute Python statements. Save the contents of below cell into a file named `first.py`.

Run this command to execute the file

```
python3 first.py   # Or python
```

```
In [1]: print("Hello World")
        print("Hi from Python File")
```

```
Hello World
Hi from Python File
```

Note: The usage of `print()` function is

```
print(list_of_values[,sep,end])
```

- `sep` is the seperator string to be printed in between `values` in `list_of_values`
- `end` is the terminating string that has to be printed after `list_of_values`

Example:

```
In [2]: print('a','b','c',sep=':',end=',')
        print('e','f','g',sep='.')
```

```
a:b:c,e.f.g
```

2
Formatting Output

In C Language, we had `printf()` function to display a formatted string to `stdout`. We also had format specifiers like `%s` for string, `%d` for integers and so on...

In Python, there are many ways to format a string, We shall have a look at each way.

Python provides `%` as String formatting operator, which has to be used with C style format specifiers. General usage of this operator is as follows.

```
result = format_string % collection_of_items
```

Now Let's see the Formatting Operator in action

```
In [16]: '8 = %d , 8.5 = %.1f, name = %s, 3 = %04d' % (8, 8.5, 'Ravi', 3)
```

```
Out[16]: '8 = 8 , 8.5 = 8.5, name = Ravi, 3 = 0003'
```

There is one more way to format, without the hassle of remembering the format specifiers. You can use `format()` method of the current string. Same output of above example can be obtained as follows

```
In [18]: '8 = {}, 8.5 = {}, name = {}, 3 = {:04}'.format(8, 8.5, 'Ravi', 3)
```

```
Out[18]: '8 = 8, 8.5 = 8.5, name = Ravi, 3 = 0003'
```

If the number of items to be formatted goes long, it would become hard to remember their positions. You can name each entry in format string, and refer to them in the call to `format()` as shown below. Note that the order of items can be changed now as the items are only referred by name.

```
In [19]: '8 = {a}, 8.5 = {b}, name = {c}, 3 = {d:04}'.format(a = 8, c = 'Ravi', d = 3, b =
↪  8.5)
```

```
Out[19]: '8 = 8, 8.5 = 8.5, name = Ravi, 3 = 0003'
```

Note

- In C and other related languages, `''` is used to refer character and `""` is used to refer string. But in Python, both refer to string. In Python single character is also a string
- Know more about Python String Formatting at PyFormat[3]

---

[3] https://pyformat.info/

3 Arithmetic and Logical Operators

The main factor that led to the invention of Computers was the search to simplify Mathematical Operations. Every computer language provides extensive support for wide range of arithmetic operations. Python's arithmetic operators are superset of those in `C`.

Let's have look at some of operations...

3.1 Arithmetic Operators

```
In [21]: 4 + 3
```

```
Out[21]: 7
```

```
In [32]: 'hi' + ' ' + 'how are you'
```

```
Out[32]: 'hi how are you'
```

```
In [37]: 'c' + 1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-37-4e0a1d805b35> in <module>()
----> 1 'c' + 1

TypeError: Can't convert 'int' object to str implicitly
```

In `C`, `char` is equivalent to `uint8` and arithmetic operations can be done. In Python, it's not the case. Thus trying to do so raises a `TypeError`

```
In [22]: 4 - 3
```

```
Out[22]: 1
```

```
In [23]: 4 / 3
```

```
Out[23]: 1.3333333333333333
```

Note

Division results in floating point number, unlike `C`. This behaviour is default from Python 3. Earlier version behaved in the same way as `C`. Use `//` operator to perform floor division.

```
In [24]: 4 // 3
```

```
Out[24]: 1
```

```
In [27]: 5.9 // 3.0
```

Out[27]: 1.0

// operator results in integer division, it rounds down the result to nearest integer

```
In [34]: 4 * 5
```

Out[34]: 20

```
In [4]: 'he' * 2 + 'h'   # A string expression!
```

Out[4]: 'heheh'

```
In [28]: 4 ** 3
```

Out[28]: 64

```
In [2]: 4 ** 0.5
```

Out[2]: 2.0

`**` operator is Power operator. `a**b` gives `a` raised to the power `b`

```
In [3]: 5 % 4
```

Out[3]: 1

All other arithmetic operators and bitwise operators and comparison operators that are present in `C` are supported. But the Logical Operators differs from `C`.

3.2 Logical Operators - `and`, `or` and `not`

Before starting with Logical Operators, note that `True` and `False` are boolean primitives in Python as opposed to `true` and `false` in `C++`,`Java` and `C#`

Note

- In Python, Single line comments start with `#`
- Multiline comments start and end with triple quotes, i.e., `'''`

Example

```
# This is a single line comment
'''This is
  a multi-
  line comment'''
```

```
In [43]: # Initialize 2 integer variables
         a = 20
         b = 10
```

```
In [44]: a == 20 and b == 10
```

Out[44]: True

```
In [45]: a is 20 or b is 0
```

Out[45]: True

```
In [46]: not a == 20
```

Out[46]: False

4 Control Structures

Control Structures construct a fundamental part of language along with syntax,semantics and core libraries. It is the Control Structures which makes the program more lively. Since they contol the flow of execution of program, they are named Control Structures

4.1 `if` statement

4.1.1 Usage:

```
if condition:
    statement_1
    statement_2
    ...
    statement_n
```

Note

In `Python`, block of code means, the lines with same indentation(i.e., same number of tabs or spaces before it). Here `statement_1` upto `statement_n` are in `if` block. This enhances the code readability

4.1.2 Example:

```
In [1]: response = input("Enter an integer : ")
        num = int(response)
        if num % 2 == 0:
            print("{} is an even number".format(num))
```

```
Enter an integer : 4
4 is an even number
```

Note Typecasting

`int(response)` converted the string `response` to integer. If user enters anything other than integer, `ValueError` is raised

4.2 if-else statement

4.2.1 Usage:

```
if condition:
    statement_1
    statement_2
    ...
    statement_n
else:
    statement_1
    statement_2
    ...
    statement_n
```

4.2.2 Example:

```
In [59]: response = input("Enter an integer : ")
         num = int(response)
         if num % 2 == 0:
             print("{} is an even number".format(num))
         else:
             print("{} is an odd number".format(num))
```

```
Enter an integer : 5
5 is an odd number
```

4.3 Single Line if-else

This serves as a replacement for ternery operator avaliable in `C`

4.3.1 Usage:

`C` ternery

```
result = (condition) ? value_true : value_false
```

Python Single Line `if else`

```
result = value_true if condition else value_false
```

4.3.2 Example:

```
In [60]: response = input("Enter an integer : ")
         num = int(response)
         result = "even" if num % 2 == 0 else "odd"
         print("{} is {} number".format(num,result))
```

```
Enter an integer : 9
9 is odd number
```

4.4 `if-else` ladder

4.4.1 Usage:

```
if condition_1:
    statements_1
elif condition_2:
    statements_2
elif condition_3:
    statements_3
...
...
...
elif condition_n:
    statements_n
else:
    statements_last
```

Note

Python uses `elif` instead of `else if` like in `C`,`Java` or `C#`

4.4.2 Example:

```
In [63]: response = input("Enter an integer (+ve or -ve) : ")
         num = int(response)
         if num > 0:
             print("{} is +ve".format(num))
         elif num == 0:
             print("Zero")
         else:
             print("{} is -ve".format(num))
```

```
Enter an integer (+ve or -ve) : -78
-78 is -ve
```

Note: No `switch-case`

There is no `switch-case` structure in Python. It can be realized using `if-else ladder` or any other ways

---

4.5 `while` loop

4.5.1 Usage:

```
while condition:
    statement_1
    statement_2
    ...
    statement_n
```

4.5.2 Example:

```
In [65]: response = input("Enter an integer : ")
         num = int(response)
         prev,current = 0,1
         i = 0
         while i < num:
             prev,current = current,prev + current
             print('Fib[{}] = {}'.format(i,current),end=',')
             i += 1
```

```
Enter an integer : 5
Fib[0] = 1,Fib[1] = 2,Fib[2] = 3,Fib[3] = 5,Fib[4] = 8,
```

Note

- Multiple assignments in single statement can be done -`Python` doesn't support `++` and `--` operators as in `C`
- There is no `do-while` loop in Python

4.6 `for` loop

4.6.1 Usage:

```
for object in collection:
    do_something_with_object
```

Notes

- `C` like `for(init;test;modify)` is not supported in Python
- Python provides `range` object for iterating over numbers

Usage of `range` object:

```
x = range(start = 0,stop,step = 1)
```

now `x` can be iterated, and it generates numbers including `start` excluding `stop` differing in the steps of `step`

4.6.2 Example:

```
In [66]: for i in range(10):
             print(i, end=',')
```

```
0,1,2,3,4,5,6,7,8,9,
```

```
In [67]: for i in range(2,10,3):
             print(i, end=',')
```

```
2,5,8,
```

```
In [68]: response = input("Enter an integer : ")
         num = int(response)
         prev,current = 0,1
         for i in range(num):
             prev,current = current,prev + current
             print('Fib[{}] = {}'.format(i,current),end=',')
```

```
Enter an integer : 5
Fib[0] = 1,Fib[1] = 2,Fib[2] = 3,Fib[3] = 5,Fib[4] = 8,
```

Note

Loop control statements `break` and `continue` work in the same way as they work in `C`

5 Functions

If a task has to be performed in a program many times, it is better to code that task as a function. Function is a piece of reusable code that can be invoked(called) from anywhere. They perform the intended task with supplied parameters and return the result if needed.

Python function has several advanatages over `C` functions and `Java` methods:

- Functions can take variable number of arguments. This is supported natively
- Functions can have named arguments (you had seen it in `print()`)
- Functions can return multiple values
- If you need a helper function for a function, you can define it inside the function

5.1 Defining a function

The syntax for defining a function is as follows

```
def function_name(argument_list):
    statement_1
    statement_2
    ...
    statement_n
    return values
```

Let's write a function for calculating $Fib(n)$, n'th Fibonacci Number, defined by

$Fib(n) = Fib(n-1) + Fib(n-2)$, where $Fib(0) = a$ and $Fib(1) = b$

First implementation uses $a = 0, b = 1$. Further implementations include options for modifying a and b

```
In [2]: def fibonacci_first(n):
            first,second = 0,1
            while n != 0:
                n, first, second = n - 1, second, first + second
            return first
```

```
In [3]: fibonacci_first(10)        # Function call
```

```
Out[3]: 55
```

Let's have an option to choose a and b

```
In [4]: def fibonacci_second(n,a,b):
            first,second = a,b
            while n != 0:
```

```
        n, first, second = n - 1, second, first + second
    return first
```

```
In [5]: fibonacci_second(9,1,1)
```

```
Out[5]: 55
```

Let a and b have the default values 0 and 1 respectively

```
In [6]: def fibonacci_third(n,a=0,b=1):
            first,second = a,b
            while n != 0:
                n, first, second = n - 1, second, first + second
            return first
```

```
In [7]: fibonacci_third(10)  # behaves like fibonacci_first()
```

```
Out[7]: 55
```

```
In [8]: fibonacci_third(9,1) # behaves like fibonacci_second(9,1,1)
```

```
Out[8]: 55
```

```
In [9]: fibonacci_third(9,1,2) # Run with fully different parameters
```

```
Out[9]: 89
```

You can also change one default value. You can do this by passing named argument to function

```
In [10]: fibonacci_third(9,b=3)
```

```
Out[10]: 102
```

5.1.1 What we have to do if we want n Fibonacci Numbers instead of nth Fibonacci Number?

- One soulution is to return a list of n numbers. We will see that once we learn about Lists in next chapter
- What we can do now is return an iterable object, that iterates through n Fibonacci numbers. Instead of `returning` a number, we can simply `yield` it to construct a genertor. The resulting Generator object can be used with `for` loop. (remember `range` object)

```
In [2]: def fibonacci_generator(n,a=0,b=1):
            first,second = a,b
            while n != 0:
                yield first
                n, first, second = n - 1, second, first + second
```

```
In [3]: for num in fibonacci_generator(10):
            print(num,end=',')
```

```
0,1,1,2,3,5,8,13,21,34,
```

Now you can also use `in` operator to check the membership of an element in the Generator Object

```
In [4]: 8 in fibonacci_generator(10)
```

```
Out[4]: True
```

```
In [5]: 10 in fibonacci_generator(10)
```

```
Out[5]: False
```

Let's modify above loop in order to print Fibonacci Numbers with numbering

```
In [6]: for i,num in enumerate(fibonacci_generator(10,a = 2, b = 3)):
            print('Fib({})={}'.format(i,num))
```

```
Fib(0)=2
Fib(1)=3
Fib(2)=5
Fib(3)=8
Fib(4)=13
Fib(5)=21
Fib(6)=34
Fib(7)=55
Fib(8)=89
Fib(9)=144
```

`enumerate()` function takes an iterable object as an argument and returns an iterator which is the original iterator enumerated.

6 Inbuilt Data Structures

Data Structures in a language determine the level of flexibility of using the language. If a Language has efficient, inbuilt data structures then the effort of the programmer is reduced. He does not have to code everything from the scratch. Furthermore, if it has user friendly syntax, it also makes the code more readable.

Python accounts for both readability and efficiency. It provides many inbuilt data structure classes that are suitable for day to day programming. In next 4 chapters, we will look at the details of `lists`,`tuples`,`sets` and `dictionarys`.

First, let's look at ***Zen*** of Python - Python Design Principles

```
In [1]: import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Python is designed according to this philosophy. Now we shall examine basic data structures which comes handy in our journey of Python.

7 Lists

List is a mutable collection of elements(may be of same or different types), which is indexed by a 0-based integer. Lists are so much like C arrays. But the capability of Python lists called Slicing makes them more powerful.

7.1 Creating Lists

- Creating an empty list

```
x = []   # [] denotes a list type
# or
x = list()
```

- Creating list with some initial elements

```
x = [2,3,0,'g']
```

```
In [1]: x = [1,2,4,5]

In [2]: x

Out[2]: [1, 2, 4, 5]
```

7.2 Accessing List elements

List elements can be accessed by 0-based integer index as in C. In addition to this, Negative indexes are also supported. If x is a list, x[-1] gives 1st element from the last, x[-2] gives second element from the last and so on. . .

```
In [3]: x[3]

Out[3]: 5

In [4]: x[-2]

Out[4]: 4
```

7.3 Obtaining Partitions of the List - Slicing

One can extract a portion of a list, and modify the value of it. If x is a list, it is achieved by a statement in the form of

```
x[start:stop:step]
```

It returns elements of x from index start to the index stop (excluding stop) in the steps of step. These 3 arguments are not mandatory. If not specified start is set to 0, stop is set to length of list and step is set to 1

```
In [5]: x = [1,2,5,6,7,0,3]

In [6]: x[1:3]  # Access from x[1] to x[2]

Out[6]: [2, 5]

In [7]: x[2:5:2] # Access from x[2] to x[4] in the steps of 2

Out[7]: [5, 7]

In [8]: x[1:3] = [6] # They can modify original list

In [9]: x   # Look at modified list, 6 is replaced twice

Out[9]: [1, 6, 6, 7, 0, 3]

In [25]: x[::-1] # Access the array in reverse order

Out[25]: [3, 0, 7, 6, 6, 1]

In [10]: x[:]  # Returns copy of list x

Out[10]: [1, 6, 6, 7, 0, 3]
```

You have observed that slices return a list, which have the reference to original list. Hence modifying slice results the change in original array.

7.4 Deleting List elements by index - del

If the position of element to be deleted is known, it can be deleted by del statement

To delete the ith element of list x,

```
del x[i]
```

```
In [11]: del x[2]

In [12]: x

Out[12]: [1, 6, 7, 0, 3]
```

7.5 Using Operators on List

```
In [13]: x = [4,3,5,0,1]
         y = [2,1,5,4,0]

In [14]: x + y

Out[14]: [4, 3, 5, 0, 1, 2, 1, 5, 4, 0]
```

Note

x + y returns a new list that contains elements of y appended to x. This has no effect on original lists x and y

```
In [15]: y * 2

Out[15]: [2, 1, 5, 4, 0, 2, 1, 5, 4, 0]
```

7.6 Operations on List

Unlike the Operators, operations performed on list can act directly on lists and may not return anything

Here are some of operations on list. They are member functions of `class list`. If `x` is a list,

- `x.append(elem)` - adds a single element to the end of the list. It does not return the new list, just modifies the original list `x`.
- `x.insert(index, elem)` - inserts the element at the given index, shifting elements to the right.
- `x.extend(list2)` - adds the elements in `list2` to the end of the list. Using `+` or `+=` on a list is similar to using `extend()`.
- `x.index(ele)` - searches for the given element from the start of the list and returns its index. Throws a `ValueError` if the element does not appear (use `in` to check without a ValueError).
- `x.remove(elem)` - searches for the first instance of the given element and removes it (throws `ValueError` if not present)
- `x.sort()` - sorts the list in place (does not return it). (The `sorted()` function is preferred.)
- `x.reverse()` - reverses the list in place (does not return it)
- `x.pop(index)` - removes and returns the element at the given index. Returns the rightmost element if `index` is omitted (roughly the opposite of append()).

```
In [16]: x = [0,3,7,2,1]
```

```
In [17]: x.append(9)
         x
```

```
Out[17]: [0, 3, 7, 2, 1, 9]
```

```
In [18]: x.insert(4,4)
         x
```

```
Out[18]: [0, 3, 7, 2, 4, 1, 9]
```

```
In [19]: x.extend([8,7,6])
         x
```

```
Out[19]: [0, 3, 7, 2, 4, 1, 9, 8, 7, 6]
```

```
In [20]: x.remove(6)
         x
```

```
Out[20]: [0, 3, 7, 2, 4, 1, 9, 8, 7]
```

```
In [21]: x.sort()
         x
```

```
Out[21]: [0, 1, 2, 3, 4, 7, 7, 8, 9]
```

```
In [22]: x.reverse()
         x
```

```
Out[22]: [9, 8, 7, 7, 4, 3, 2, 1, 0]
```

```
In [23]: x.pop()
```

```
Out[23]: 0
```

```
In [24]: x.pop(0)
```

```
Out[24]: 9
```

```
In [25]: x
```

```
Out[25]: [8, 7, 7, 4, 3, 2, 1]
```

```
In [26]: sorted(x)
```

```
Out[26]: [1, 2, 3, 4, 7, 7, 8]
```

List elements can also be lists, which gives 2-D array like structure

```
In [27]: x = [[2,3,4],
              [1,2,2],
              [2,3,4]]
```

```
In [28]: x[1]
```

```
Out[28]: [1, 2, 2]
```

```
In [29]: x[2][1]
```

```
Out[29]: 3
```

Note

There is no rule that the length of each sublist in a list must be same

7.7 Obtaining length of list - `len`

```
In [30]: x = [1,2,3]
         len(x)
```

```
Out[30]: 3
```

```
In [31]: x = [[2,3,4],2,['a','v']]
```

```
In [32]: len(x)
```

```
Out[32]: 3
```

7.8 Membership Operator `in`

`in` operator can be used to check the existance of an element in the list

```
In [33]: x = [1,2,3,0,5,4]
         4 in x
```

```
Out[33]: True
```

```
In [34]: 10 in x
```

```
Out[34]: False
```

7.9 Converting an iterator to list

Using `yield` keyword, one can create an iterator. Using `list()`, one can make a list of all values `yield`ed by iterator

```
In [60]: list(range(10))
```

```
Out[60]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

8 Tuples

Tuple is an immutable collection of elements (may be of same or different types), which is indexed by a 0-based integer. A 2-tuple can represent a point in 2-D plane, or a 3-Tuple can represent a point in 3-D plane.

8.1 Creating Tuples

- Creating an empty tuple

```
x = ()  # () denotes a tuple type
# or
x = tuple()
```

- Creating list with some initial elements

```
x = (2,3,0,'g')
```

```
In [1]: x = (2,3)

In [2]: x

Out[2]: (2, 3)

In [3]: x = (2,)  # x = (2) assigns int 2 to x. To make it a tuple, a comma is appended

In [4]: x

Out[4]: (2,)

In [5]: x + (1,2)

Out[5]: (2, 1, 2)
```

Tuples are immutable. So once a tuple is created, its contents are permanent unless it is reassigned with another tuple.

Tuples can also be Indexed and Sliced like lists

```
In [6]: x

Out[6]: (2,)

In [7]: x = x + (1,3,4)   # Reassignment

In [8]: x

Out[8]: (2, 1, 3, 4)

In [9]: x[1]
```

```
Out[9]: 1

In [10]: x[2:5]

Out[10]: (3, 4)

In [11]: x[::-1]

Out[11]: (4, 3, 1, 2)
```

8.2 Operations on Tuples

Since tuples are immutable, operations do not modify the original tuple

Here are some of the operations on list. They are member functions of `class tuple`. If `x` is a tuple,

- `x.index(ele)` - searches for the given element from the start of the list and returns its index. Throws a `ValueError` if the element does not appear (use `in` to check without a ValueError).
- `x.count(ele)` - counts the number of occurances of `ele` in `x`

Membership operator `in` is also supported

```
In [12]: x

Out[12]: (2, 1, 3, 4)

In [13]: 2 in x

Out[13]: True

In [14]: x.count(2)

Out[14]: 1

In [17]: x.index(3)

Out[17]: 2
```

Using a list of tuples, one can model a collection of points in space

9 Sets

Set is a mutable, unordered collection of unique, hashable elements(may be of same or different types), which is not indexed.

9.1 Creating Set

- Creating an empty set

```
x = set()
```

- Creating set with some initial elements

```
x = {2,3,0,'g'}
```

- Creating an empty set with `{}` is not possible as `{}` is reserved for dictionary `dict` objects

```
In [1]: x = {1,2,5,3}
        x
```

```
Out[1]: {1, 2, 3, 5}
```

9.2 Accessing Set Elements

Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.

9.3 Operations on Set

Like any other collection, `set` supports membership operators `in` and `not in`, elements of `set` can be iterated. If `A` and `B` are 2 sets,Following is a list of other operations on set

- `A.union(B)` - returns $A \cup B$
- `A.union_update(B)` - $A = A \cup B$
- `A.intersection(B)` - returns $A \cap B$
- `A.intersection_update(B)` - $A = A \cap B$
- `A.isdisjoint(B)` - returns $A \cap B == \emptyset$
- `A.issubset(B)` - returns $A \subseteq B$

- `A.issuperset(B)` - returns $A \supseteq B$

Other operations like set difference are also supported

```
In [2]: x

Out[2]: {1, 2, 3, 5}

In [3]: x.union([2,4,6])

Out[3]: {1, 2, 3, 4, 5, 6}

In [4]: x.intersection([2,3])

Out[4]: {2, 3}

In [5]: x.intersection_update([1,3,4])

In [6]: x

Out[6]: {1, 3}
```

Note Graph and Sets

Many Graph Algorithms are modelled using sets. A Graph G is considered as a collection of sets of vertices V and sets of edges E

9.4 Set of Sets

In many cases, it is required to have set of sets as in case of finding subsets of a set. Since `set` is not hashable, it is **not possible to have a "set" as an element of "set"**. In this case `frozenset` comes handy. The only difference between `frozenset` and a `set` is that `frozenset` is immutable. We have to reassign value to it if we want to modify it.

10 Dictionaries

Dictionary is a set of key-value pairs, where value is any hashable object. As Lists are indexed by integers, Dictionries are indexed by keys.

10.1 Creating Dictionaries

- Creating an Empty Dictionary

```
x = {}       # {} denotes dictionary type (not set)
x = dict()
```

- Creating Dictionary with initial values

```
x = {'eight':8,'nine':9}
```

```
In [1]: x = {'eight':8,'nine':9}
```

```
In [2]: x['eight']
```

```
Out[2]: 8
```

```
In [3]: x[0]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-3-1ae75c28907a> in <module>()
----> 1 x[0]

KeyError: 0
```

If a key is not present in Dictionary, `KeyError` is raised

```
In [4]: x['zero'] = 0 # Adding a key-value to dictionary
        x
```

```
Out[4]: {'eight': 8, 'nine': 9, 'zero': 0}
```

```
In [5]: del x['nine']   # Deleting based on key
        x
```

```
Out[5]: {'eight': 8, 'zero': 0}
```

```
In [6]: 'zero' in x     # Only key membership can be checked
```

```
Out[6]: True
```

10.2 Dictionary Methods

If d is a dictionary

- d.keys() returns a view of d's keys
- d.values() returns a view of d's values
- d.items() returns a view of d's key-value pairs

```
In [7]: x
```

```
Out[7]: {'eight': 8, 'zero': 0}
```

```
In [8]: x.keys()
```

```
Out[8]: dict_keys(['zero', 'eight'])
```

```
In [9]: x.values()
```

```
Out[9]: dict_values([0, 8])
```

```
In [10]: x.items()
```

```
Out[10]: dict_items([('zero', 0), ('eight', 8)])
```

```
In [12]: for k,v in x.items():
             print("{} = {}".format(k,v))
```

```
zero = 0
eight = 8
```

Dictionary Values can be any hashable object. This means they can be lists, tuples,… . Using Dictionaries, one can implement an Adjacency List Representation of Graph Data Structure.

11 Strings

Strings play a major role in a programming language. Apart from providing nicer user interactions, they can also serve as communication tool within the parts of the program. Python provides a rich set of tools for Pattern matching using RegEx[4]s, String formatting based on locate and Various encryption methods. We have seen basic String Formatting in *previous chapter*. In this chapter let's study about some basic functions that operates on strings.

11.1 Creating Strings

- Creating Empty String

```
s = str()
```

or

```
s = '' # quotes can be any of '' or ""
```

- Creating String with Initial Value

```
s = "Some text goes here"
```

11.2 Accessing the elements of Strings

String elements can be accessed using integer indices. A slice can also be specified for accessing a required part of the string

```
In [1]: s = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
↪  incididunt ut labore et dolore magna aliqua"
        s

Out[1]: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
↪  incididunt ut labore et dolore magna aliqua'

In [2]: s[10]

Out[2]: 'm'

In [3]: s[20:] # start from 10 to end of string

Out[3]: 't amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et
↪  dolore magna aliqua'
```

[4] http://regexr.com/

```
In [4]: s[:20] # start from 0 to index 19
Out[4]: 'Lorem ipsum dolor si'
In [5]: s[10:30:2] # start from 10, end at 29 with steps of 2
Out[5]: 'mdlrstae,c'
In [6]: s[30:10:-2] # in reverse order
Out[6]: 'nc,eatsrld'
```

11.3 Operators on Strings

```
In [7]: 'Lorem' in s
Out[7]: True
In [8]: 'Some Random text : ' + s
Out[8]: 'Some Random text : Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
↪  eiusmod tempor incididunt ut labore et dolore magna aliqua'
In [10]: for i,c in enumerate(s): # string is iterable
             if i == 11:
                 break
             else:
                 print(c,end='')
Lorem ipsum
```

11.4 Operations on Strings

If `s` is a string,

- `s.format(elements)` formats `s` and returns it
- `s.join(elements)` returns a string in which the `elements` have been joined by `s` separator.
- `s.capitalize()` Return a copy of the string with its first character capitalized and the rest lowercased.
- `s.count(sub[, start[, end]])` Return the number of non-overlapping occurrences of substring `sub` in the range `[start, end]`. Optional arguments `start` and `end` are interpreted as in slice notation.
- `s.find(sub[, start[, end]])` Return the lowest index in the string where substring `sub` is found within the slice `s[start:end]`. Optional arguments `start` and `end` are interpreted as in slice notation. Return `-1` if `sub` is not found.
- `s.isalpha()` Return `True` if all characters in the string are alphabetic and there is at least one character, `False` otherwise.
- `s.split(sep=None, maxsplit=-1)` Return a list of the words in the string, using `sep` as the delimiter string. If `maxsplit` is given, at most `maxsplit` splits are done (thus, the list will have at most `maxsplit+1` elements). If `maxsplit` is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).
- `s.strip([chars])` Return a copy of the string with the leading and trailing characters removed. The `chars` argument is a string specifying the set of characters to be removed. If omitted or None, the chars argument defaults to removing whitespace. The `chars` argument is not a prefix or suffix; rather, all combinations of its values are stripped

```
In [17]: s
```

```
Out[17]: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
↪  incididunt ut labore et dolore magna aliqua'

In [18]: s.count('it')

Out[18]: 2

In [19]: s.find('it')

Out[19]: 19

In [20]: s.split(',')

Out[20]: ['Lorem ipsum dolor sit amet',
          ' consectetur adipiscing elit',
          ' sed do eiusmod tempor incididunt ut labore et dolore magna aliqua']

In [21]: part = s.split(',')[2]
         part

Out[21]: ' sed do eiusmod tempor incididunt ut labore et dolore magna aliqua'

In [22]: part = part.strip()
         part

Out[22]: 'sed do eiusmod tempor incididunt ut labore et dolore magna aliqua'

In [23]: part = part.upper()
         part

Out[23]: 'SED DO EIUSMOD TEMPOR INCIDIDUNT UT LABORE ET DOLORE MAGNA ALIQUA'

In [24]: '-'.join('defg')

Out[24]: 'd-e-f-g'

In [25]: s = 'abcd'

In [26]: s += 'defg'        # Appending
         s

Out[26]: 'abcddefg'
```

This is just an overview of Python String Functions. There are many more functions which can do various tasks. You will get to know them when you need the functionality.

12 Comprehensions

Sometimes, it is useful to make some operations on Data Structures and return the same Data Structure. Examples may include squaring every element of a collection, constructing a lookup table and so on. Python provides an easier syntax for doing these tasks. Let's understand comprehension techiniques by solving some problems

12.1 Problem 1

Given a list of integers, Create a new list containing their squares

12.1.1 Classic, C like approach

```
In [3]: num = [10,8,3,5,2,7,0,1,4,9,6]
        num

Out[3]: [10, 8, 3, 5, 2, 7, 0, 1, 4, 9, 6]

In [6]: def square_classic_approach(x):
            """
            Input: x - List of Integers
            Return: A list containing squares of each element of list
            """
            squared = []     # Empty list
            for i in range(len(num)):    # Equivalent to for(i=0;i<num;i++)
                squared.append(num[i]**2) # Power operator
            return squared
```

Note

Documentation of `Python` code can be done using docstrings[5]s like in above code

```
In [7]: square_classic_approach(num)

Out[7]: [100, 64, 9, 25, 4, 49, 0, 1, 16, 81, 36]
```

12.1.2 Comprehension Based Approach

```
In [13]: def square_pythonic_approach(x):    # Pythonic! :P
             """
```

[5] https://www.python.org/dev/peps/pep-0257/

```
            Input: x - List of Integers
            Return: A list containing squares of each element of list
            """
            return [ num**2 for num in x ]
```

```
In [9]: square_pythonic_approach(num)
```

```
Out[9]: [100, 64, 9, 25, 4, 49, 0, 1, 16, 81, 36]
```

Note

Comprehension increases code readability. Comprehension can be applied to any collection.

12.2 Problem 2

Given a list of integers, square them if they are even number and return a list

12.2.1 Classic, C like approach

```
In [23]: def square_if_even_classic_approach(x):
             """
             Input: x - List of Integers
             Return: A list containing squares of each element of list if element is even
             """
             squared = []     # Empty list
             for i in range(len(x)):    # Equivalent to for(i=0;i<num;i++)
                 if x[i] % 2 == 0:
                     squared.append(x[i]**2) # Power operator
                 else:
                     squared.append(x[i])
             return squared
```

```
In [24]: square_if_even_classic_approach([10,9,2,4,5,67])
```

```
Out[24]: [100, 9, 4, 16, 5, 67]
```

Observe how list is created and passed on the fly

12.3 Comprehension based approach

```
In [16]: def square_if_even_pythonic_approach(x):    # Pythonic! :P
             """
             Input: x - List of Integers
             Return: A list containing squares of each element of list if element is even
             """
             return [ num**2 if num % 2 == 0 else num for num in x ]
```

```
In [17]: square_if_even_pythonic_approach(([10,9,2,4,5,67]))
```

```
Out[17]: [100, 9, 4, 16, 5, 67]
```

12.4 Problem 3

Given a string, return the vowels occuring in it, ignoring the case

```
In [31]: def vowels_in(string):
             """
             Input: string - a string
             Return: List of vowels occuring in string
             Example:
             >>> vowels_in('Apple`)
             ['a','e']
             """
             # We use a set because it stores unique elements
             l_string = str.lower(string) # Converting to unique form
             vowel_set = {c for c in l_string if c in 'aeiou'}  # Note the imposal of condition
             return sorted(list(vowel_set))
```

```
In [29]: vowels_in('Apple')
```

```
Out[29]: ['a', 'e']
```

```
In [30]: vowels_in('Karnataka')
```

```
Out[30]: ['a']
```

Above function can be written in more compact form

```
In [32]: def vowels_in_compact(string):
             """
             Input: string - a string
             Return: List of vowels occuring in string
             Example:
             >>> vowels_in('Apple`)
             ['a','e']
             """
             return sorted(list({c for c in str.lower(string) if c in 'aeiou'}))
```

```
In [33]: vowels_in_compact('violin')
```

```
Out[33]: ['i', 'o']
```

12.5 Problem 4

Given a string, count the number of occurance of each character, ignoring the case

The nature of the problem makes us to use the dicionary data structure.

```
In [36]: def alphabet_occurance_count(string):
             """
             Input: a string
             Output: the number of occurance od f each character in the string
             """
             return {x:string.count(x) for x in string}
```

```
In [38]: alphabet_occurance_count("Hello from Notebook")
```

```
Out[38]: {' ': 2,
          'H': 1,
          'N': 1,
          'b': 1,
          'e': 2,
          'f': 1,
          'k': 1,
          'l': 2,
          'm': 1,
          'o': 5,
          'r': 1,
          't': 1}
```

12.6 *Zen* revisited

In the beginning of this chapter, we looked at *Zen* of Python. Now we inspect the things in **"this"** module **In the "this" module of Python, "this.s" contains the encoded text, which has to be decoded using "this.d". Decode it**

```
In [39]: import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Obviously, **this** is *Zen*

Now let's have a look at other things in this module

```
In [40]: this.c
```

```
Out[40]: 97
```

```
In [41]: this.d
```

```
Out[41]: {'A': 'N',
          'B': 'O',
          'C': 'P',
          'D': 'Q',
          'E': 'R',
          'F': 'S',
          'G': 'T',
          'H': 'U',
          'I': 'V',
          'J': 'W',
          'K': 'X',
          'L': 'Y',
          'M': 'Z',
          'N': 'A',
          'O': 'B',
          'P': 'C',
          'Q': 'D',
          'R': 'E',
          'S': 'F',
          'T': 'G',
          'U': 'H',
          'V': 'I',
          'W': 'J',
          'X': 'K',
          'Y': 'L',
          'Z': 'M',
```

```
          'a': 'n',
          'b': 'o',
          'c': 'p',
          'd': 'q',
          'e': 'r',
          'f': 's',
          'g': 't',
          'h': 'u',
          'i': 'v',
          'j': 'w',
          'k': 'x',
          'l': 'y',
          'm': 'z',
          'n': 'a',
          'o': 'b',
          'p': 'c',
          'q': 'd',
          'r': 'e',
          's': 'f',
          't': 'g',
          'u': 'h',
          'v': 'i',
          'w': 'j',
          'x': 'k',
          'y': 'l',
          'z': 'm'}
```

It looks like a mapping from one character to another... Hmm... Interesting!

```
In [42]: this.i
```

```
Out[42]: 25
```

```
In [43]: this.s
```

```
Out[43]: "Gur Mra bs Clguba, ol Gvz Crgref\n\nOrnhgvshy vf orggre guna htyl.\nRkcyvpvg vf
↪   orggre guna vzcyvpvg.\nFvzcyr vf orggre guna pbzcyrk.\nPbzcyrk vf orggre guna
↪   pbzcyvpngrq.\nSyng vf orggre guna arfgrq.\nFcnefr vf orggre guna qrafr.\nErnqnovyvgl
↪   pbhagf.\nFcrpvny pnfrf nera'g fcrpvny rabhtu gb oernx gur ehyrf.\nNygubhtu cenpgvpnyvgl
↪   orngf chevgl.\nReebef fubhyq arire cnff fvyragyl.\nHayrff rkcyvpvgyl fvyraprq.\nVa gur snpr
↪   bs nzovthvgl, ershfr gur grzcgngvba gb thrff.\nGurer fubhyq or bar-- naq cersrenoyl bayl
↪   bar --boivbhf jnl gb qb vg.\nNygubhtu gung jnl znl abg or boivbhf ng svefg hayrff lbh'er
↪   Qhgpu.\nAbj vf orggre guna arire.\nNygubhtu arire vf bsgra orggre guna *evtug* abj.\nVs gur
↪   vzcyrzragngvba vf uneq gb rkcynva, vg'f n onq vqrn.\nVs gur vzcyrzragngvba vf rnfl gb
↪   rkcynva, vg znl or n tbbq vqrn.\nAnzrfcnprf ner bar ubaxvat terng vqrn -- yrg'f qb zber bs
↪   gubfr!"
```

Wow!...Looks like encoded text.

We will decode it using `this.d` mapping

```
In [47]: decoded = ''.join([this.d[c] if str.isalnum(c) else c for c in this.s]) # join() joins
↪   the iterable with string
         print(decoded)
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
```

```
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

This is again ***Zen of Python!***

In fact **“this“** module uses a function to build the *Translation Table “d“* in it’s `__init.py__` to print the ***Zen***. The encoding done here is `rot13` encoding. We will look about modules in upcoming chapters.

12.7 Fibonacci Again

Now you have understood the lists and operations. Let’s look at a recursive Fibonacci Number Generator

```
In [53]: def fibonacci_recursive(n,first=0,second=1):
             return [] if n == 0 else [first] + fibonacci_recursive(n - 1, second, first +
 ↪  second)
```

```
In [54]: fibonacci_recursive(5)
```

```
Out[54]: [0, 1, 1, 2, 3]
```

13 Filtering Lists - Need for `lambdas`

In Python, functions are also objects. It means that you can pass them to other function like a variable. This flexibility of functions allows us to do many useful tasks. `filter`ing a collection is one of them.

13.1 Problem : Find even numbers in a given sequence

13.1.1 Solution 1: Use List comprehension

```
In [55]: x = [0,1,3,5,8,7,6]

In [57]: evens = [i for i in x if i%2 == 0]
         evens

Out[57]: [0, 8, 6]
```

13.1.2 Solution 2: Use `filter` with functions

`filter` function takes a list and a function returning bool as argument and `filter`'s the list, returns the iterator through `filter`ed list. One can use `list()` to convert iterator to a list

Usage

```
result = list(filter(condition,collection))
```

`condition` is a boolean function that takes an element as input

```
In [58]:  def is_even(item):
              return item % 2 == 0

In [61]: list(filter(is_even,x))

Out[61]: [0, 8, 6]
```

13.2 Solution 3: Use λs

In previous example, we passed a function object to `filter()`. The same case happens in many situations. In some cases function to be passed might be too short like `is_even()`. In this case `lambdas` can be used. `lambdas` create function in place.

Usage:

```
function_name = lambda argument_list : executable_statements
```

This has the same effect as that of

```
def function_name(argument_list):
    executable_statements
```

Now our `is_even` function can be defined in terms of lambdas

```
is_even = lambda x : x % 2 == 0 # Note that return may be omitted
```

If multi-line statements are needed, Statements can be put inside `()`s or line can be extended with \s

```
In [62]: list(filter(lambda x: x % 2 == 0,x))
```

```
Out[62]: [0, 8, 6]
```

Lambdas are a fundamental concept of *Functional Programming* where every task is achieved via a function. They constitute the basis of a branch of Mathematics and Computation Theory called **:math:'lambda' calculus**.

As a final thought, we shall see Recursive Fibonacci Generator in terms of `lambdas`

```
In [5]: fibonacci_lambda = \
            lambda n,first=0,second=1 :\
                    [] if n == 0 \
                       else \
                          [first] + fibonacci_lambda(n - 1, second, first + second)
```

```
In [6]: fibonacci_lambda(4)
```

```
Out[6]: [0, 1, 1, 2]
```

Note that lines are broken with \

14 Modules

In previous chapter, we saw the ***Zen*** of Python . We also noticed that this resides in `this` module. In this chapter, we discuss about modules. We also study how to create modules.

14.1 What is a module?

According to official documentation, a module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended.

14.2 An Example

In a directory, create a file called `fibo.py` and paste the following code in it.

`fibo.py`

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Now open `ipython` (`ipython3`) in the same directory and execute the following statements:

```
>>> import fibo
```

This does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters the module name fibo there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

```
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

14.3 More ways to `import` methods `from` a module

There is a variant of the `import` statement that imports names from a module directly into the importing module's symbol table. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (_). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

14.4 Executing modules as scripts

When you run a Python module with:

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the "main" file:

`python fibo.py 50 1 1 2 3 5 8 13 21 34`

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

14.5 The Module Search Path

When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
- The installation-dependent default.

14.6 Packages

Packages are a way of structuring Python's module namespace by using "dotted module names". For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or the Python Imaging Library from having to worry about each other's module names.

Suppose you want to design a collection of modules (a "package") for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: `.wav`, `.aiff`, `.au`), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here's a possible structure for your package (expressed in terms of a hierarchical filesystem):

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

When importing the package, Python searches through the directories on sys.path looking for the package subdirectory.

The `__init__.py` files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as string, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Users of the package can import individual modules from the package, for example:

```
import sound.effects.echo
```

This loads the submodule `sound.effects.echo`. It must be referenced with its full name.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

An alternative way of importing the submodule is:

```
from sound.effects import echo
```

This also loads the submodule `echo`, and makes it available without its package prefix, so it can be used as follows:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Yet another variation is to import the desired function or variable directly:

```
from sound.effects.echo import echofilter
```

Again, this loads the submodule `echo`, but this makes its function `echofilter()` directly available:

```
echofilter(input, output, delay=0.7, atten=4)
```

Note that when using `from package import item`, the `item` can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The import statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an `ImportError` exception is raised.

Contrarily, when using syntax like `import item.subitem.subsubitem`, each item except for the last must be a package; the last item can be a module or a package but can't be a class or function or variable defined in the previous item.

14.6.1 Reference

This chapter is copied from Official Python Documentation[6]

[6] https://docs.python.org/3/tutorial/modules.html

15 Object Oriented Programming

In Object Oriented Programming, everything is an object. Objects are real world entities having some attributes and some related methods that operate on attributes. We assume that the reader has some familiarity with Object Oriented Concepts such as Inheritance, Polymorphism, Abstraction and so on . . .

15.1 Defining Classes

Syntax:

```
class ClassName:
    <statement 1>
    <statement 2>
    ....
    ....
    <statement n>
```

15.2 Special Methods inside the class

Unlike `C++` and `Java` classes, class methods does not hold the reference of current object (`this` object). Class methods should take the class object as their first argument. This is not required for static methods. At the point of invocation of object methods, the object is passed to method implicitly. It is a covention to name the first parameter of class method as `self`. Now let's see some special functions of classes.

- `__init__(self,elements)` : Constructor, called when object is created. All properties of the object have to be declared here.
- `__del__(self)` : Destructor, called when `del` is applied to an object.
- `__str__(self)` : Returns the string representation of object. Called when `str()` is called on the object.
- `__iter__(self)` : Returns the iterator of elements of the object. Called when `iter()` is called on the object. Also this enables us to use the `for ele in object` like construct.
- `__len(self)__` : Returns the length of the collection. Called when `len()` is invoked on the object.
- `__getitem(self,item)__` : Allows us to use `object[item]` like accessor to get an item

15.3 Static members and methods

Any member declared inside the class, but not in the methods, are shared by all instances of classes. A method annotated with `@staticmethod` is static method, and doesn't recieve class object as it's first parameter.

15.4 A note on `private` members

A member or method whose name starts with '`__`' is regarded as a private member or method.

15.5 A sample class, `Student`

Here we implement a simple `Student` class.

```
In [86]: class Student:
             count = 0 # Total number of objects created so far, it is static variable as it is
↪  declared outside

             def __init__(self,name,usn,marks):
                 """
                 Constructor of class Student
                 Input: name - name of the student : string
                        usn  - university serial number : string
                        marks - marks in 3 subjects out of 20
                 """
                 Student.count += 1
                 self.name = name
                 self.usn  = usn
                 self.marks = marks[:] # Copy marks to self.marks .. a simple self.marks =
↪  marks make only reference equal

             def print_details(self):
                 print(str(self))

             def total_marks(self):
                 return sum(self.marks)

             def __iter__(self):
                 details = {'name':self.name,'usn':self.usn,'marks':self.marks}
                 for k,v in details.items():
                     yield k,v # A tuple

             def __str__(self):
                 return "Name : {0} \nUSN = {1}\nMarks in 3 subjects =
↪  {2}".format(self.name,self.usn,self.marks)

             @staticmethod
             def get_total_count():
                 return Student.count
```

```
In [87]: s1 = Student('Ramesh','4jc11cs111',[20,16,18])
         s2 = Student('Ravi','4jc15cs112',[15,18,18])
```

```
In [88]: print(s1) # calls __str__()
```

```
Name : Ramesh
USN = 4jc11cs111
Marks in 3 subjects = [20, 16, 18]
```

```
In [89]: print(s2)
```

```
Name : Ravi
USN = 4jc15cs112
Marks in 3 subjects = [15, 18, 18]
```

```
In [91]: Student.count
```

```
Out[91]: 2
```

```
In [90]: Student.get_total_count()
```

```
Out[90]: 2
```

```
In [92]: for k,v in s1:
             print('{} = {}'.format(k,v))
```

```
usn = 4jc11cs111
name = Ramesh
marks = [20, 16, 18]
```

```
In [95]: s1.print_details()       # self of Student.print_details(self) is passed as s1
```

```
Name : Ramesh
USN = 4jc11cs111
Marks in 3 subjects = [20, 16, 18]
```

```
In [97]: Student.print_details(s1) # Explicitly passing self parameter
```

```
Name : Ramesh
USN = 4jc11cs111
Marks in 3 subjects = [20, 16, 18]
```

```
In [98]: Student.get_total_count()
```

```
Out[98]: 2
```

```
In [100]: s1.get_total_count()   # This is also possible, @staticmethod attribute prevents
↪  passing object to method
```

```
Out[100]: 2
```

15.6 Duck Typing and Interfaces

In `C`, `C++`, `Java` and `C#`, we have to predefine the data type of every variable declared. In Python, you may have observed that you are not defining any data type during variable declaration. In fact, Python does not require you to do that.

In `C`,

```
int x;
```

means storage space allocated to `x` is constant 8 bytes (on `x64` system) and this space will never change. This also implies that `x` will never hold other values than int. Trying to do so will raise a compiler error. This nature of `C` makes the language `statically typed`, i.e., data type of a variable is determined at the compile time.

On the other hand, in Python, the type of variable is determined entirely during runtime. Storage space allocated to a variable can vary dynamically. When we assign a string to a variable `x`, it will be `str`. If we reassign it to a list, it will be `list`. This nature of `Python` makes it `dynamically typed` language. It is also called as `Duck typing`.

Duck typing is an application of the *duck test* in type safety. It requires that type checking be deferred to runtime, and is implemented by means of dynamic typing or reflection.

The Duck test is a humorous term for a form of abductive reasoning. This is its usual expression:

```
If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.
```

The duck test can be seen in the following example. As far as the function `in_the_forest` is concerned, the Person object is a duck:

```
In [1]: class Duck:
            def quack(self):
                print("Quaaaaaack!")
            def feathers(self):
                print("The duck has white and gray feathers.")

        class Person:
            def quack(self):
                print("The person imitates a duck.")
            def feathers(self):
                print("The person takes a feather from the ground and shows it.")
            def name(self):
                print("John Smith")

        def in_the_forest(duck):
            duck.quack()
            duck.feathers()

        def game():
            donald = Duck()
            john = Person()
            in_the_forest(donald)
            in_the_forest(john)

        game()
```

```
Quaaaaaack!
The duck has white and gray feathers.
The person imitates a duck.
The person takes a feather from the ground and shows it.
```

15.7 `type()` - Obtaining the data type of a variable

```
In [102]: x = 8
          type(x)
```

```
Out[102]: int
```

```
In [103]: type(8.5)
```

```
Out[103]: float
```

```
In [104]: type('hello')
```

```
Out[104]: str
```

```
In [105]: type([1,2,1])
```

```
Out[105]: list
```

```
In [106]: type({})
```

```
Out[106]: dict
```

```
In [108]: type((1,))
```

```
Out[108]: tuple
```

```
In [109]: type(s1)
```

```
Out[109]: __main__.Student
```

```
In [111]: import random
          type(random)
```

```
Out[111]: module
```

The main intention of interfaces in `Java` and `C#` was to make the classes to have a set of common functions, which makes their usage alike. Due to Duck Typing, the need for interfaces is now gone

16 Inheritance

Inheritance means extending the properties of one class by another. Inheritance implies code reusability, because of which client classes do not need to implement everything from scratch. They can simply refer to their base classes to execute the code.

Unlike `Java` and `C#`, like `C++`, `Python` allows Multiple inheritance. Name resolution is done by the order in which the base classes are specified.

16.1 Syntax

```
class ClassName(BaseClass1[,BaseClass2,....,BaseClassN]):
    <statement 0>
    <statement 1>
    <statement 2>
    ...
    ...
    ...
    <statement n>
```

16.1.1 A First Example

```
In [3]: class Person:

            # Constructor
            def __init__(self, name, age):
                self.name = name
                self.age = age

            def __str__(self):
                return 'name = {}\nage = {}'.format(self.name,self.age)

        # Inherited or Sub class
        class Employee(Person):

            def __init__(self, name, age, employee_id):
                Person.__init__(self, name, age) # Referring Base class
                # Can also be done by super(Employee, self).__init__(name, age)
                self.employee_id = employee_id

            # Overriding implied code reusability
            def __str__(self):
                return Person.__str__(self) + '\nemployee id = {}'.format(self.employee_id)
```

```
In [4]: s = Person('Kiran',18)
        print(s)
```

```
name = Kiran
age = 18
```

```
In [6]: e = Employee('Ramesh',18,48)
        print(e)
```

```
name = Ramesh
age = 18
employee id = 48
```

Note

Base class can be referred from derived class in two ways

- Base Class name - `BaseClass.function(self,args)`
- using `super()` - `super(DerivedClass, self).function(args)`

16.1.2 Multiple inheritance and Order of Invocation of Methods

```
In [7]: class Base1:
            def some_method(self):
                print('Base1')


        class Base2:
            def some_method(self):
                print('Base2')

        class Derived1(Base1,Base2):
            pass

        class Derived2(Base2,Base1):
            pass
```

Note how `pass` statement is used to leave the class body empty. Otherwise it would have raised a Syntax Error. Since `Drived1` and `Derived2` are empty, they would have imported the methods from their base classes

```
In [8]: d1 = Derived1()
        d2 = Derived2()
```

Now what will be the result of invoking `some_method` on `d1` and `d2`? . . . Does the name clash ocuur? . . . Let's see

```
In [9]: d1.some_method()
```

```
Base1
```

```
In [10]: d2.some_method()
```

```
Base2
```

Wow! . . . It executed smoothly . . .

If a name of a function is same in base classes, the one will be executed, which appears first in the base class list

17 Exceptions

In an ideal situation, our program runs smoothly without any errors. However it is not always the case. Errors may be due to developer's fault or programmer's mistake or of computer. Source of some errors might be hard to undertsand. However it is the task of Good Programmer to handle all kinds of errors that might occur in his program. If some error condition escapes from the developer and user catches it, It is a **bug** in the program. Developers must update the programs periodically to fix the bugs in the software. You may remember that recent ransomware attack which caused the loss of enormous amount of data, was due to a bug in Microsoft Windows.

17.1 Facing a first exception

Let's write a lambda to divide 2 numbers

```
In [1]: div = lambda x,y : x/y
```

```
In [2]: div(8,2)
```

```
Out[2]: 4.0
```

```
In [3]: div(0/0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-3-d742f81f0a4a> in <module>()
----> 1 div(0/0)

ZeroDivisionError: division by zero
```

Oh No!... It was a error. Let's handle it.

17.2 `try-except-finally`

`try-except-finally` provides an easy way to handle errors that can arise during program execution. It works similar to `try-catch-finally` blocks in `Java` and `C#`

Syntax:

```
try:
    <statement 1>
    <statement 2>
    ...
    <statement n>
except (Exception List):    # Refer note
```

```
    <statement 1>
    <statement 2>
    ...
    <statement n>
finally:
    <cleanup 1>
    <cleanup 2>
    ...
    <cleanup n>
```

Note:

- `finally` block is optional
- If Exception List is empty all exceptions are handled by `except` block
- If catching a single exception, it can be referred with its name.

```
except RangeError as e:
        <do-something-with-e>
```

- Base Exception classes must be captured at last, if catching exceptions in hierarchy

17.3 `div` with exception handling

```
In [5]: def div_good(x,y):
            try:
                return x/y
            except ZeroDivisionError:
                print("Division by zero")
```

```
In [6]: div_good(8,2)
```

```
Out[6]: 4.0
```

```
In [7]: div_good(0,0)
```

```
Division by zero
```

Note how the exception was handled

17.4 Cleaning the things up

In this version of `div`, we will return a `NaN` if a `ZeroDivisionError` occures. `'NaN'` is **Not a Number**. `'Inf'` refers infinity

```
In [4]: def div_clean(x,y):
            try:
                value = x/y
            except ZeroDivisionError:
                value = float('NaN')
            return value
```

```
In [2]: div_clean(4,3)
```

```
Out[2]: 1.3333333333333333
```

```
In [3]: div_clean(8,0)
```

```
Out[3]: nan
```

17.5 Raising Exceptions

The `raise` statement allows the programmer to force a specified exception to occur. For example:

```
In [5]: raise NameError('HiThere')

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-93385ba972b1> in <module>()
----> 1 raise NameError('HiThere')

NameError: HiThere
```

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
In [6]: raise ValueError  # shorthand for 'raise ValueError()'

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-f4e87a14b34e> in <module>()
----> 1 raise ValueError  # shorthand for 'raise ValueError()'

ValueError:
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception:

```
In [7]: try:
            raise NameError('HiThere')
        except NameError:
            print('An exception flew by!')
            raise

An exception flew by!

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-3f47609917d7> in <module>()
      1 try:
----> 2     raise NameError('HiThere')
      3 except NameError:
      4     print('An exception flew by!')
      5     raise

NameError: HiThere
```

17.6 User-defined Exceptions

Programs may name their own exceptions by creating a new exception class. Exceptions should typically be derived from the `Exception` class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception. When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

```
In [1]: class Error(Exception):
            """Base class for exceptions in this module."""
            pass

        class InputError(Error):
```

```python
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is not allowed
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

Most exceptions are defined with names that end in `Error`, similar to the naming of the standard exceptions.

18 File Handling

So far, we have worked with the objects in Primary Memory. However Primary Memory is volatile. In order to save the current state of program, objects for future use, we have to save it in Secondary Memory. It is achieved via file handling.

18.1 Opening Files

`open()` returns a file object, and is most commonly used with two arguments: `open(filename, mode)`. `mode` is a string that determines how the file should be opened. Normally, files are opened in text mode, that means, you read and write strings from and to the file, which are encoded in a specific encoding. If encoding is not specified, the default is platform dependent (see open()). 'b' appended to the mode opens the file in binary mode: now the data is read and written in the form of bytes objects. This mode should be used for all files that don't contain text.

- `r` - Read
- `w` - Write
- `a` - Append
- `r+` - Read and Write, similarly `w+` and `a+`

If no mode is specified, it is defaulted to `r`

Normally, files are opened in *text mode*, that means, you read and write strings from and to the file, which are encoded in a specific encoding. If encoding is not specified, the default is platform dependent. `'b'` appended to the mode opens the file in binary mode: now the data is read and written in the form of bytes objects. This mode should be used for all files that don't contain text.

It is good practice to use the `with` keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point. Using with is also much shorter than writing equivalent try-finally blocks:

```
>>> with open('workfile') as f:
...     read_data = f.read()
>>> f.closed
True
```

If you're not using the `with` keyword, then you should call `f.close()` to close the file and immediately free up any system resources used by it. If you don't explicitly close a file, Python's garbage collector will eventually destroy the object and close the open file for you, but the file may stay open for a while. Another risk is that different Python implementations will do this clean-up at different times.

After a file object is closed, either by a with statement or by calling `f.close()`, attempts to use the file object will automatically fail.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file
```

18.2 Methods of File Objects

The rest of the examples in this section will assume that a file object called `f` has already been created.

To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode). **size** is an optional numeric argument. When size is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most size bytes are read and returned. If the end of the file has been reached, `f.read()` will return an empty string (`''`).

```
>>>
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` reads a single line from the file; a newline character (`\n`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by '`\n`', a string containing only a single newline.

```
>>>
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
>>>
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

If you want to read all the lines of a file in a list you can also use `list(f)` or `.readlines()`.

`f.write(string)` writes the contents of string to the file, returning the number of characters written.

```
>>>
>>> f.write('This is a test\n')
15
```

Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) – before writing them:

```
>>>
>>> value = ('the answer', 42)
>>> s = str(value)  # convert the tuple to string
```

```
>>> f.write(s)
18
```

`f.tell()` returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.

To change the file object's position, use `f.seek(offset, from_what)`. The position is computed from adding offset to a reference point; the reference point is selected by the `from_what` argument. A from_what value of `0` measures from the beginning of the file, `1` uses the current file position, and `2` uses the end of the file as the reference point. `from_what` can be omitted and defaults to 0, using the beginning of the file as the reference point.

```
>>>
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

In text files (those opened without a `b` in the mode string), only seeks relative to the beginning of the file are allowed (the exception being seeking to the very file end with `seek(0, 2)`) and the only valid offset values are those returned from the `f.tell()`, or zero. Any other offset value produces undefined behaviour.

File objects have some additional methods, such as `isatty()` and `truncate()` which are less frequently used; consult the Library Reference for a complete guide to file objects.

Note : This chapter is copied from Python Reference[7]

[7] https://docs.python.org/3/tutorial/inputoutput.html

19 Going Further

Now that you have touched Python, you can tell how easy it is! Python makes many things possible that would be possible with many complications in `C` or `Java`.

- You can use as many libraries as you want, **Don't** code everything from the scratch
  - `requests` for sending HTTP Request
  - `scipy` for Scientific Computation
  - `numpy` for High Performace Arrays
  - `tensorflow` for neural networks (By Google) and many more
- Make your code well documented with the usage of docstrings
- Don't hesitate to Google
- StackOverflow[8] is the place
- Feel free to contribute to this project at GitHub[9]

Good Luck Ahead!

[8] https://stackoverflow.com/questions/tagged/python
[9] https://github.com/OpenWeavers/OpenAlgorithm

Part II

Essential Libraries

20 Need for a faster `array`

We know how lists work in Python. We also know that lists can hold the data items of various data types. This means that the list storage allocated to elements can vary in size. This factor makes the list access slow, and operations on array could take long time. `numpy` provides a elagent solution in the form of `ndarray`, a n - Dimensional collections of elements with same data types. `numpy` also provides easier way to manipulate arrays, This makes it High Performance Numerical Calculation possible.

20.1 Importing `numpy`

Following is the standard statement to import `numpy`. In future examples and library usages, we assume that you have imported the library in this way

```
In [1]: import numpy as np
```

20.2 Creating `ndarray` from Lists

`numpy` allows us to create an array from exsisting Python List. Datatype conversions are performed if the input list contains elements of multiple datatypes. Datatypes are always promoted. Let's look at some examples.

```
In [2]: a = np.array([1,2,3,4])
        a

Out[2]: array([1, 2, 3, 4])

In [3]: b = np.array([[1],[2],[3]])
        b

Out[3]: array([[1],
               [2],
               [3]])

In [4]: c = np.array([1,2,'x'])
        c

Out[4]: array(['1', '2', 'x'],
              dtype='<U21')
```

Note how `int` is converted into `str`. `U21` is 32-bit Unicode Encoding. (Actual bits needed to store data is 21)

```
In [5]: d = np.array([1.3,1,3])
        d

Out[5]: array([ 1.3,  1. ,  3. ])
```

Note how `int` is converted to `float`

20.3 Accessing array elements and random shuffling

Array elements can be accessed using indices, slices and using masked arrays

Let's create a random array and then illustrate the methods of accessing array elements

```
In [6]: x = np.arange(20)  # Like range(), but returns ndarry instead
        x
```

```
Out[6]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
               17, 18, 19])
```

```
In [7]: x.shape # (rows,cols)
```

```
Out[7]: (20,)
```

```
In [8]: x.shape = (4,5) # 4 rows 5 cols
        x
```

```
Out[8]: array([[ 0,  1,  2,  3,  4],
               [ 5,  6,  7,  8,  9],
               [10, 11, 12, 13, 14],
               [15, 16, 17, 18, 19]])
```

```
In [9]: x.size  # Total number of elements
```

```
Out[9]: 20
```

```
In [10]: np.random.shuffle(x) # shuffles ndarray in-place
         x
```

```
Out[10]: array([[ 5,  6,  7,  8,  9],
                [ 0,  1,  2,  3,  4],
                [15, 16, 17, 18, 19],
                [10, 11, 12, 13, 14]])
```

This is how we can shuffle an array. `random.shuffle()` function takes an `ndarray` as an argument and sorts it *in place*. **NEVER** treat its return value as result!

```
In [11]: x[0]
```

```
Out[11]: array([5, 6, 7, 8, 9])
```

```
In [12]: x[0][2]  # Ok, inefficient
```

```
Out[12]: 7
```

Above method is inefficient access because, it fetches `x[0]` first and accesses it's element at index 2. Next method computes the address from 2 co-ordinates directly, and fetches the element at one access

```
In [13]: x[0,2]   # Efficient
```

```
Out[13]: 7
```

```
In [14]: x[0,1:4]
```

```
Out[14]: array([6, 7, 8])
```

Above example selects the elements at indices (0,1),(0,2),(0,3). Note that the slices can also be used to select elements from multi-dimensional array

```
In [15]: x[1:4,0]
```

```
Out[15]: array([ 0, 15, 10])
```

```
In [16]: x > 15
```

```
Out[16]: array([[False, False, False, False, False],
                [False, False, False, False, False],
                [False,  True,  True,  True,  True],
                [False, False, False, False, False]], dtype=bool)
```

Note that it returned a boolean array after performing suitable operation. It is called masked array

```
In [17]: x [ x > 15 ]
```

```
Out[17]: array([16, 17, 18, 19])
```

This method to access array element is called as Access by Masked array

20.4 Functions that operates on `ndarrays`

Numpy provides many Mathematical functions, that not only operates on inividual numbers, but also on entire arrays. Let's illustrate them

```
In [18]: np.sin(x)
```

```
Out[18]: array([[-0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849],
                [ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ],
                [ 0.65028784, -0.28790332, -0.96139749, -0.75098725,  0.14987721],
                [-0.54402111, -0.99999021, -0.53657292,  0.42016704,  0.99060736]])
```

```
In [19]: x[np.sin(x) > 0] # elements whose sine is non-negative
```

```
Out[19]: array([ 7,  8,  9,  1,  2,  3, 15, 19, 13, 14])
```

many trigonometrical functions like *sin*,*cos*, calculus related functions like *grad* are also available

20.4.1 `concatenate` the arrays

`concatenate((a1, a2, ...), axis=0)` Join a sequence of arrays along an existing axis.

Parameters: - `a1, a2, ...` : sequence of array_like

- The arrays must have the same shape, except in the dimension corresponding to axis (the first, by default).
- axis : int, optional
- The axis along which the arrays will be joined. Default is 0.
- Returns:
- res : ndarray
- The concatenated array.
- `hstack((a1, a2, ...))` combines `a1, a2, ...` horizontally
- `vstack((a1, a2, ...))` combines `a1, a2, ...` vertically
- `dstack((a1, a2, ...))` combines `a1, a2, ...` depthwise

```
In [20]: a = np.array([1,2,3,4])
         b = np.array([9,8,7,6])
```

```
In [21]: a
```

```
Out[21]: array([1, 2, 3, 4])
```

```
In [22]: b
```

```
Out[22]: array([9, 8, 7, 6])
```

```
In [23]: np.concatenate((a,b),axis=0) # (a,b) is a tuple of arrays
```

```
Out[23]: array([1, 2, 3, 4, 9, 8, 7, 6])
```

```
In [24]: np.dstack((a,b))

Out[24]: array([[[1, 9],
                 [2, 8],
                 [3, 7],
                 [4, 6]]])

In [25]: np.vstack((a,b))

Out[25]: array([[1, 2, 3, 4],
                [9, 8, 7, 6]])

In [26]: np.hstack((a,b))

Out[26]: array([1, 2, 3, 4, 9, 8, 7, 6])
```

We will use these functions frequently in upcoming chapters

20.4.2 Aggregate Functions

Aggregate Functions are those which operates on entire array, to provide an overview of the elements

`sum`, `average` like functions fall in this catagory

We will use the below array to illustrate the usage of aggregate functions

```
In [27]: s = np.sin(x)
         s

Out[27]: array([[-0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849],
                [ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ],
                [ 0.65028784, -0.28790332, -0.96139749, -0.75098725,  0.14987721],
                [-0.54402111, -0.99999021, -0.53657292,  0.42016704,  0.99060736]])

In [28]: np.sum(s) # You understood it, right?!

Out[28]: 0.085276633692154657

In [29]: np.average(s)

Out[29]: 0.0042638316846077325

In [30]: np.min(x)

Out[30]: 0

In [31]: np.max(s)

Out[31]: 0.99060735569487035
```

At current point, we will stop. This basic understanding of `numpy` is enough to understand the concepts of Algorithm Analysis in upcoming part.

Interested readers can refer the NumPy Official Tutorial at SciPy[10]

[10] https://docs.scipy.org/doc/numpy-dev/user/quickstart.html

21 Data Visualization

When we have thousands of sampled numerical data, it makes no sense without classifying them and analyzing them. Many Statistical tools are available to classify the data in Python. `pandas` is one such library. After classifying the data, it is useful to visualize the classified data. Visualization can result in greater understanding of Data, such as Corelation and so on. `matplotlib` is one of the famous, easy-to-use library for data visualization

21.1 Standard Import statement

In `matplotlib`, we won't use entire library. We just use a part of library which is dedicated for plotting data. In further discussions related about `matplotlib`, we assume that the reader has imported the library in following manner

```
In [1]: import matplotlib.pyplot as plt
```

21.2 Our First Graph - A Parabola

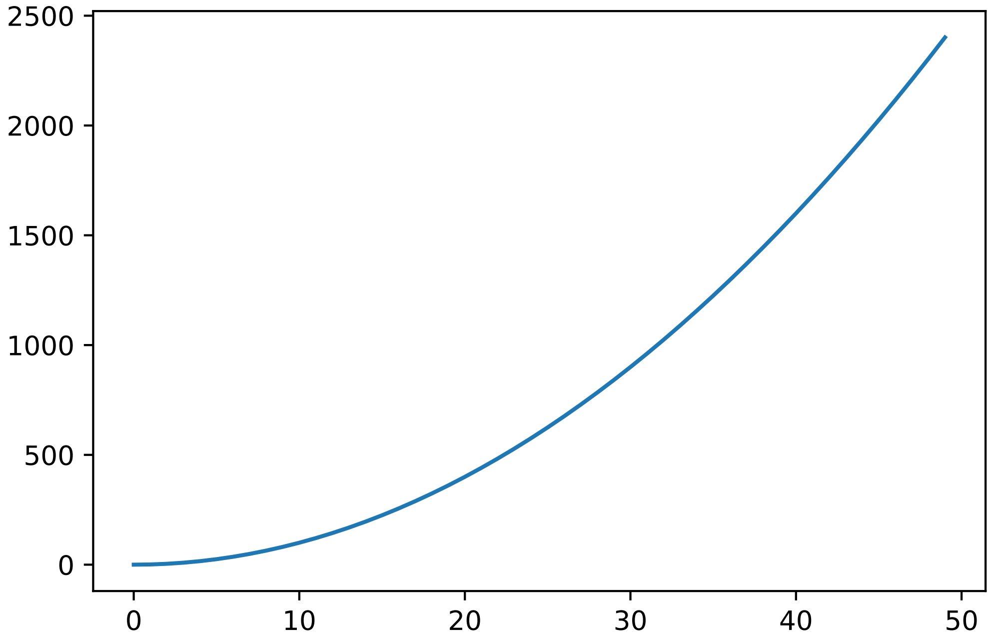
$y = x^2$ is the equation of standard parabola. We sample some x values and calculate the square of them. Then we plot a graph of y versus x to obtain the parabola

```
In [2]: import numpy as np

In [3]: x = np.arange(50)   # 0..19
        y = x**2

In [4]: plt.plot(x,y) # First argument is x data, second data is y data
        # plt.show() , If in Python Script

Out[4]: [<matplotlib.lines.Line2D at 0x7fda54f14ba8>]
```



Note

If not using a Interactive Notebook or `IPython` shell, then issue a

```
plt.show()
```

to see the plot

Also see how `matplotlib` converted set of points to represent a parabola by **interpolation**

21.3 Customizing the Graph - Changing its type and color

When representing various data in graph, different style must be used to distinguish between the data sets. In this section, we will see how to manipulate the line style and color. Following are the named arguments that are sent to `plot()` functions.

21.3.1 `linestyle = value`

can be used to change line style.

We shall see the inbuilt `lineStyles` dict to see what are the possible styles for `value`

```
{'-': '_draw_solid', '--': '_draw_dashed', '-.': '_draw_dash_dot', ':': '_draw_dotted', 'None
↪': '_draw_nothing', ' ': '_draw_nothing', '': '_draw_nothing'}
```

21.3.2 `color = value`

can be used to change color of line. `value` can be one of

- b: blue
- g: green
- r: red
- c: cyan

- m: magenta
- y: yellow
- k: black
- w: white

21.3.3 `alpha = value`

α - value determines the visibility of plot. It is a floating point number between 0 and 1. $\alpha = 0$ implies that the plot is not visible. $\alpha = 1$ implies that the plot is completely visible

21.4 Plotting multiple graphs on same axis

Many times, it is required to plot many datasets on same axis, so that we can compare them. MatPlotLib makes it possibe in a simple way. One can achieve this by issuing plotting commands successively and finally issuing a `show()`.

21.5 An All-in-One example

Let's examine all these things by plotting $y = \frac{1}{x}, y = \sin(x), y = \cos(2x)$ and $y = 2\sin(2x)$ in a single plot. Instead of using `np.arange()` for x data, We shall use the `np.linspace()` method

21.5.1 `np.linspace(start, stop, num=50)`

Return evenly spaced numbers over a specified interval.

Returns num evenly spaced samples, calculated over the interval `[start, stop]`.

The endpoint of the interval can optionally be excluded.

Parameters:

- `start : scalar` : The starting value of the sequence.
- `stop : scalar` : The end value of the sequence, unless endpoint is set to False. In that case, the sequence consists of all but the last of num + 1 evenly spaced samples, so that stop is excluded. Note that the step size changes when endpoint is False.
- `num : int`, optional : Number of samples to generate. Default is 50. Must be non-negative. endpoint : bool, optional

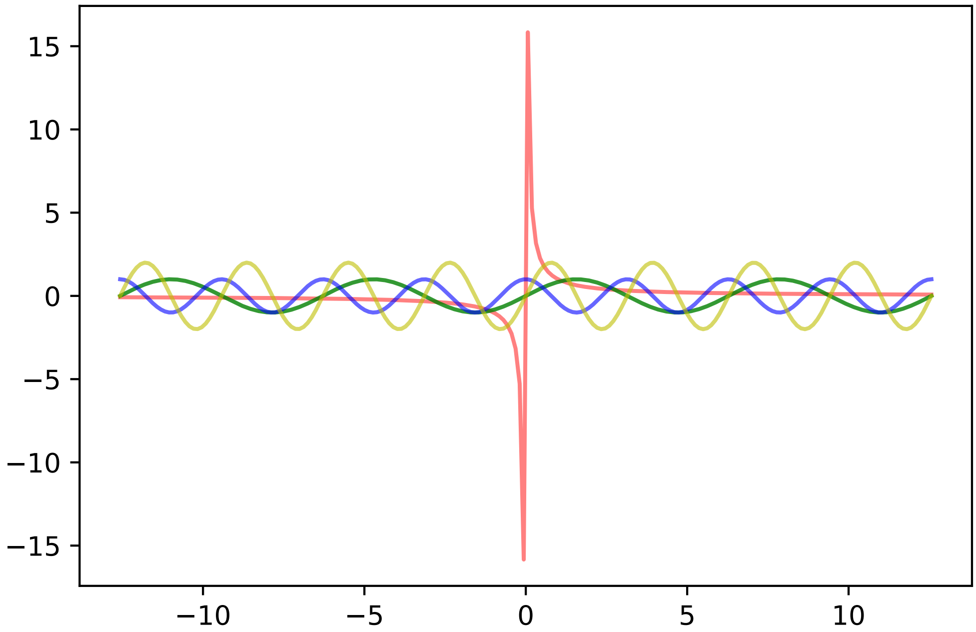
Returns:

- `samples : ndarray` : There are num equally spaced samples in the closed interval [start, stop] or the half-open interval [start, stop) (depending on whether endpoint is True or False).

```
In [5]: f1 = lambda x: 1/x
        f2 = lambda x: np.sin(x)
        f3 = lambda x: np.cos(2 * x)
        f4 = lambda x: 2 * np.sin(2 * x)
        x = np.linspace(-4 * np.pi, 4 * np.pi , 200)
        p1 = plt.plot(x,f1(x), color = 'r', alpha = 0.5)
        plt.plot(x,f2(x), color = 'g', alpha = 0.8)
        plt.plot(x,f3(x), color = 'b', alpha = 0.6)
```

```
        plt.plot(x,f4(x), color = 'y', alpha = 0.6)
        # plt.show() # If using in Python Script
```

Out[5]: [<matplotlib.lines.Line2D at 0x7fda54bc49b0>]



21.6 Subplots

In many cases, we want the opposite of what we have just discussed. We want to plot the data sets in different subplots. MatplotLib has many ways to obtain the subplots of given plot. Here we will just discuss one of them.

```
plt.subplot(nrows,ncols,active)
```

creates the subplots with shape $nrows \times ncols$, and selects a subplot for plotting specified on `active`. `active` is a `1` based index for selecting subplot. It selects subplots in row-wise order.

21.7 Adding Title

Adding title to subplot can be achieved via

```
plt.title('label')
```

Adding title to Super plot can be achieved by

```
plt.suptitle('label')
```

21.8 An example

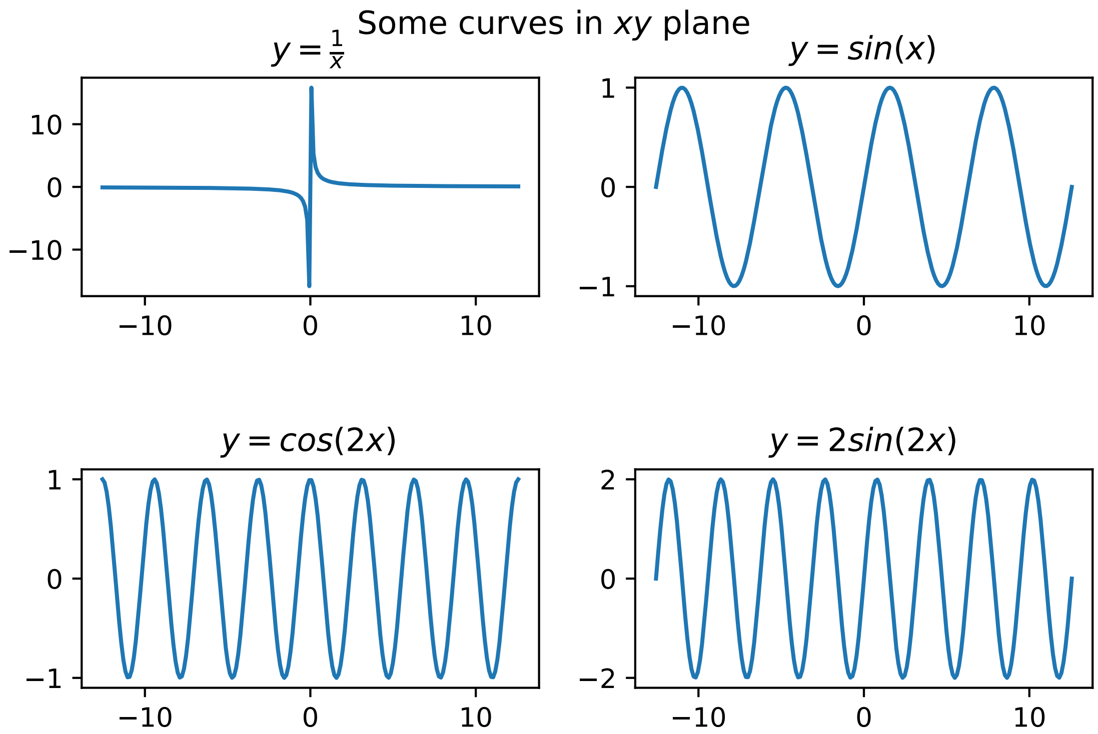
In the below example, Let's see all of the things discussed in action

```
In [6]: functions = [ lambda x: 1/x, lambda x: np.sin(x), lambda x: np.cos(2 * x), lambda x: 2
↪  * np.sin(2 * x) ]
        lables = [r'$y = \frac{1}{x}$' , '$y = sin(x)$', '$y = cos(2x)$', '$y = 2 sin(2x)$' ]
```

```
x = np.linspace(-4 * np.pi, 4 * np.pi , 200)
plt.suptitle('Some curves in $xy$ plane')


for i,(function,label) in enumerate(zip(functions,lables),start = 1):
    # zip() combines 2 iterables as list of tuples
    # enumerate() enumerated the zip here
    # enumerate returns an iterator through (count,value) tuples
    # but value is iteself is a tuple of (funciton,label) here
    # So we have to catch a tuple (count,(function,lablel))
    plt.subplot(2,2,i)
    plt.plot(x, function(x))
    plt.title(label)

plt.tight_layout(h_pad=3) # Exclude this and see what happens
# plt.show() # if using in script
```

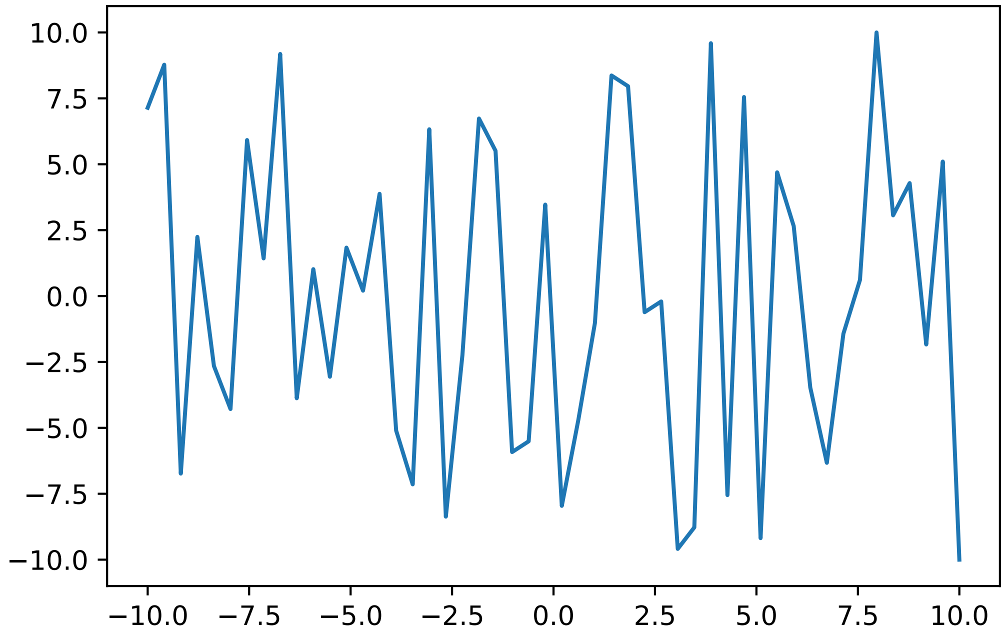


21.9 Plotting irregular data - Scatter and Bar Plots

Some data shows irregular pattern, due to which they can't be interpolated. When plotting such data, MatplotLib behaves crazily. In this situation, we have to use some other plotting method other than `plot()`. Before exploring other methods, Let's see a situation where ordinary plotting doesn't work.

```
In [7]: arr = np.linspace(-10,10)
        x = np.copy(arr) # If you use x = arr, their reference will be copied
        np.random.shuffle(arr)
        plt.plot(x,arr)
        # plt.show() # if using in Python Script
```

```
Out[7]: [<matplotlib.lines.Line2D at 0x7fda540f7eb8>]
```



Above image does not seem to be like a plot of some Polynomial or Other function. In fact, We will not treat them as plot of some function. They are just **data**.

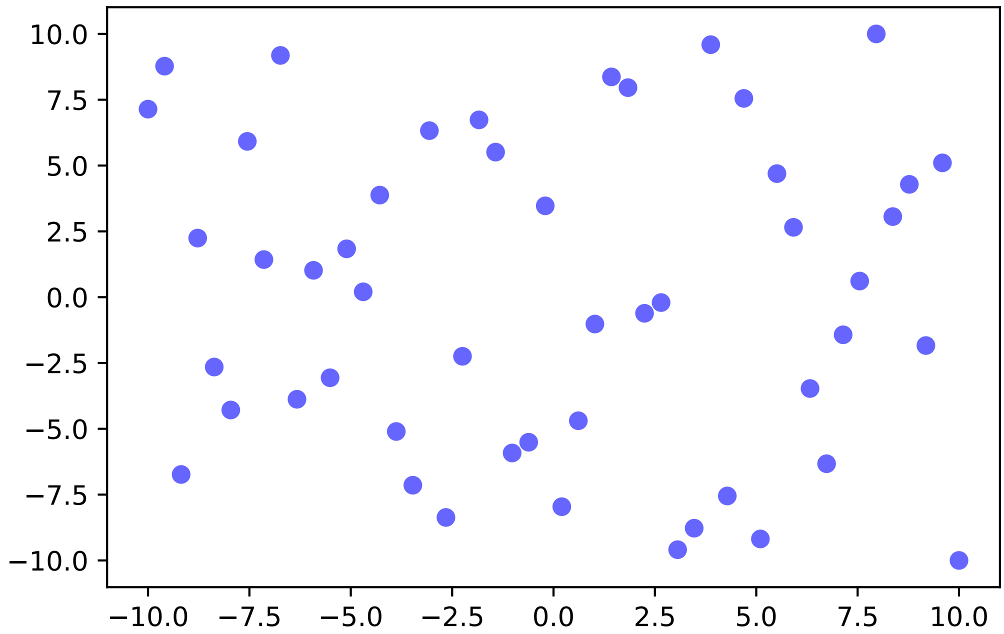
To visualize this kind of data, Scatter and Bar plots can be used

21.9.1 Scatter Plot

Scatter plot only plots the sample points, instead of interpolation and drawing lines between them. It takes the same arguments as that of `plot()`. Let's see one

```
In [8]: plt.scatter(x, arr, color='b',alpha = 0.6)
```

```
Out[8]: <matplotlib.collections.PathCollection at 0x7fda540dec50>
```



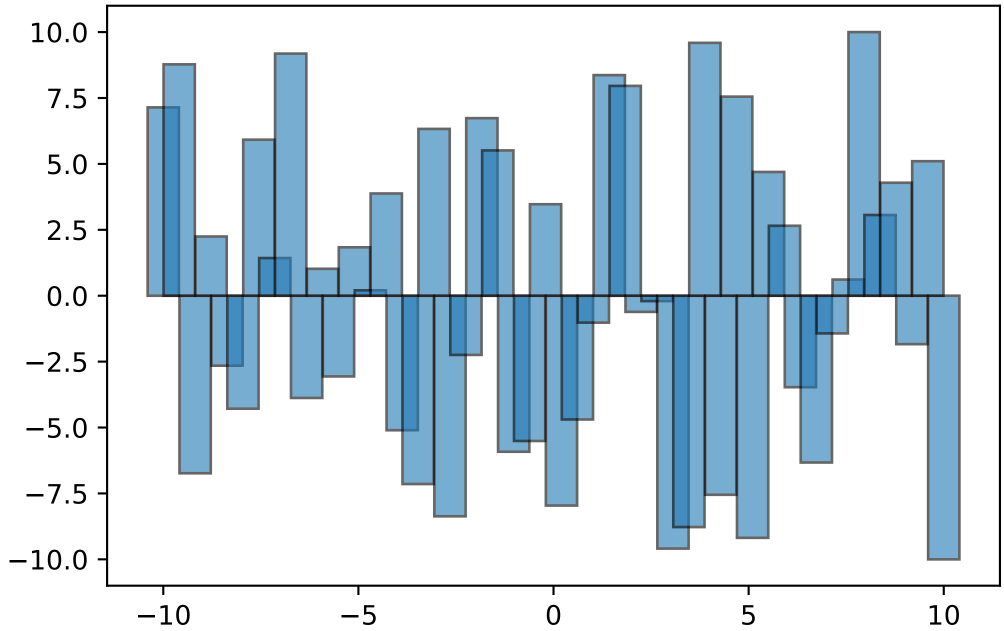
Note how we changed the color and alpha of plot.

21.9.2 Bar Plot

Bar plot visualizes the data as bars, whose height is proportional to the magnitude of data. Let's plot the same data as bar chart and understand it's customization.

```
In [9]: plt.bar(x, arr, alpha = 0.6, edgecolor='k')
```

```
Out[9]: <Container object of 50 artists>
```



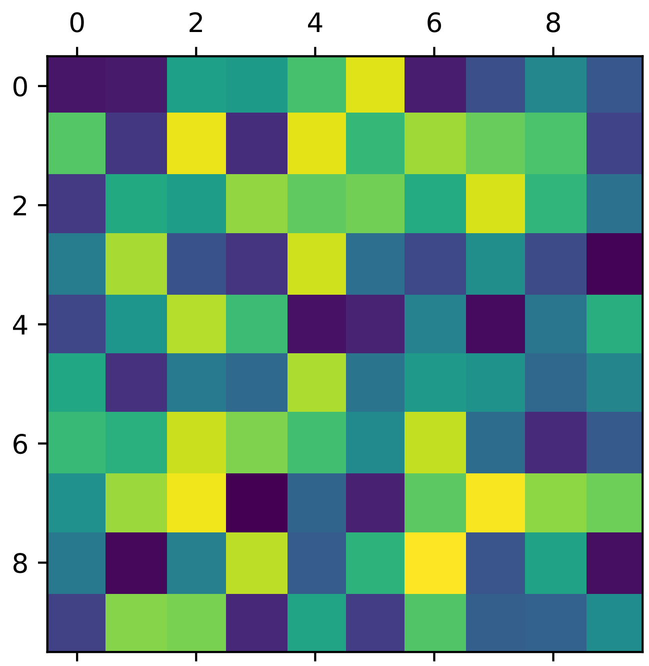
Note how rectangle edges are visible with black color. Overlapping rectangles are also visible with $\alpha = 0.6$

21.10 Visualizing 2D Data - Matrix

A matrix can be interpreted as values of a function $f(i, j)$ where i and j are indices of matrix. Now f can be visualized as a surface over $ijplane$. This requires switching to 3D co-ordinates. Instead of doing that, one can visualize the same in 2D plane by mapping the each value to a colormap. In MatplotLib, we can do this by `imshow()` and `matshow()`

```
In [10]: data = np.arange(100)
         np.random.shuffle(data)
         data.shape = (10,10)
         plt.matshow(data)
```

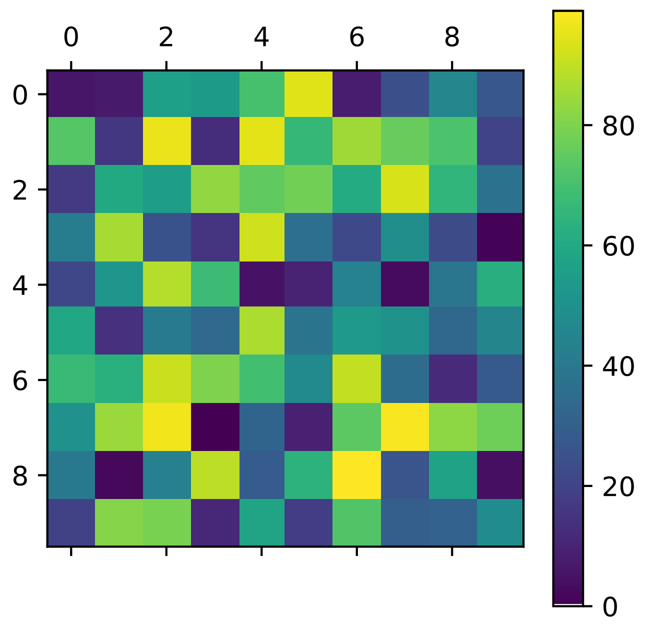
```
Out[10]: <matplotlib.image.AxesImage at 0x7fda54bfa400>
```



To know what color means what value, one can enable the `colorbar`

```
In [11]: plt.matshow(data)
         plt.colorbar()
```

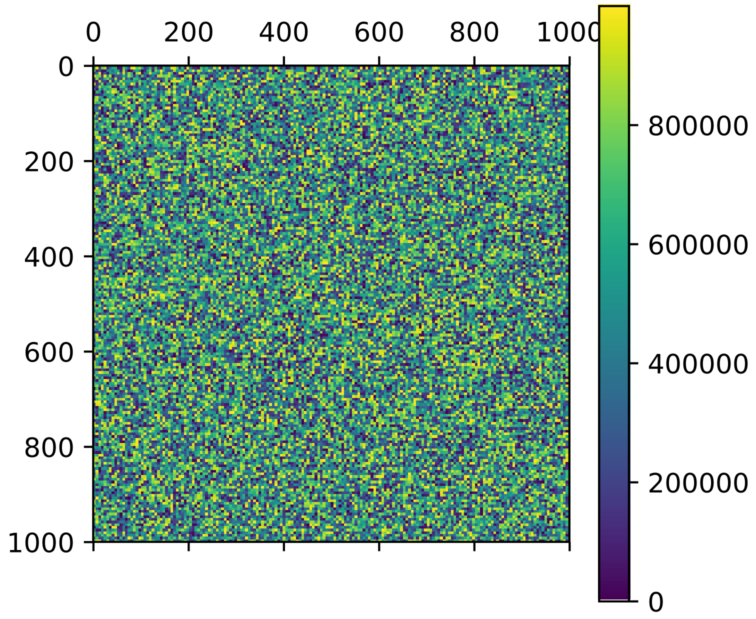
```
Out[11]: <matplotlib.colorbar.Colorbar at 0x7fda54b9ddd8>
```



Let's experiment with some large data

```
In [12]: n = 1000
         data = np.arange(n**2)
         np.random.shuffle(data)
         data.shape = (n,n)
         plt.matshow(data)
         plt.colorbar()
```

```
Out[12]: <matplotlib.colorbar.Colorbar at 0x7fda4efbf828>
```



It looks like above plot is like a random image. In fact, images are also matrices. Different file formats like `jpeg`,`png` and `tiff` store the matrix and associated data in different ways.

Consider an image with resolution `1900 * 1600`

- Its data is a matrix with shape `(1900 , 1600)`
- If it is a color image, Each element of matrix is either a value, 3-tuple or 4-tuple based on it's color scheme
- If image is monochromatic, each element of matrix is value. 0 representing white, 255 representing black
- If color scheme of image is RGB, each element of matrix is `(Red,Green,Black)` tuple with each element ranging from 0 to 256
- If color scheme of image is CMYK, each element of matrix is `(Cyan,Magenta,Yellow,blacK)` tuple with each element ranging from 0 to 256

Since image is a matrix, any operation on matrix is a operation on image. It is the basis of how Photo Editing Softwares work. It is also the fundamental of a field of Computer Science known as **Image Processing**

21.11 Going Further

In this tutorial, we have seen just the fundamentals of Data Visualizations using `matplotlib`. There are many more kinds of plots, one can even animate the plots. Interested reader can refer Official Tutorial[11]

[11] https://matplotlib.org/users/pyplot_tutorial.html

22 Introduction to Graph Analysis with networkx

Graph theory deals with various properties and algorithms concerned with Graphs. Although it is very easy to implement a Graph ADT in Python, we will use `networkx` library for Graph Analysis as it has inbuilt support for visualizing graphs. In future versions of `networkx`, graph visualization might be removed. When this happens, it is required to modify some parts of this chapter

22.1 Standard import statement

Throughout this tutorial, we assume that you have imported `networkx` as follows

```
In [38]: import networkx as nx
```

22.2 Creating Graphs

Create an empty graph with no nodes and no edges.

```
In [39]: G = nx.Graph()
```

By definition, a `Graph` is a collection of nodes (vertices) along with identified pairs of nodes (called edges, links, etc). In NetworkX, nodes can be any hashable object e.g. a text string, an image, an XML object, another Graph, a customized node object, etc. (Note: Python's None object should not be used as a node as it determines whether optional function arguments have been assigned in many functions.)

22.3 Nodes

The graph `G` can be grown in several ways. NetworkX includes many graph generator functions and facilities to read and write graphs in many formats. To get started, we'll look at simple manipulations. You can add one node at a time,

```
In [40]: G.add_node(1)
```

add a list of nodes,

```
In [41]: G.add_nodes_from([2,3])
```

22.4 Edges

G can also be grown by adding one edge at a time,

```
In [42]: G.add_edge(1,2)
         e=(2,3)
         G.add_edge(*e)  # Unpacking tuple
```

by adding a list of edges,

```
In [43]: G.add_edges_from([(1,2),(1,3)])
```

we add new nodes/edges and NetworkX quietly ignores any that are already present.

At this stage the graph G consists of 3 nodes and 3 edges, as can be seen by:

```
In [44]: G.number_of_nodes()
```

```
Out[44]: 3
```

```
In [45]: G.number_of_edges()
```

```
Out[45]: 3
```

22.5 Accessing edges

In addition to the methods `Graph.nodes`, `Graph.edges`, and `Graph.neighbors`, iterator versions (e.g. `Graph.edges_iter`) can save you from creating large lists when you are just going to iterate through them anyway.

Fast direct access to the graph data structure is also possible using subscript notation.

Warning

Do not change the returned `dict`–it is part of the graph data structure and direct manipulation may leave the graph in an inconsistent state.

```
In [46]: G.nodes()
```

```
Out[46]: [1, 2, 3]
```

```
In [47]: G.edges()
```

```
Out[47]: [(1, 2), (1, 3), (2, 3)]
```

```
In [48]: G[1]
```

```
Out[48]: {2: {}, 3: {}}
```

```
In [49]: G[1][2]
```

```
Out[49]: {}
```

You can safely set the attributes of an edge using subscript notation if the edge already exists.

```
In [50]: G[1][2]['weight'] = 10
```

```
In [51]: G[1][2]
```

```
Out[51]: {'weight': 10}
```

Fast examination of all edges is achieved using adjacency iterators. Note that for undirected graphs this actually looks at each edge twice.

```
In [52]: FG=nx.Graph()
         FG.add_weighted_edges_from([(1,2,0.125),(1,3,0.75),(2,4,1.2),(3,4,0.375)])
         for n,nbrs in FG.adjacency_iter():
             for nbr,eattr in nbrs.items():
                 data=eattr['weight']
                 if data<0.5: print('(%d, %d, %.3f)' % (n,nbr,data))
```

```
(1, 2, 0.125)
(2, 1, 0.125)
(3, 4, 0.375)
(4, 3, 0.375)
```

```
In [53]: list(FG.adjacency_iter())
```

```
Out[53]: [(1, {2: {'weight': 0.125}, 3: {'weight': 0.75}}),
          (2, {1: {'weight': 0.125}, 4: {'weight': 1.2}}),
          (3, {1: {'weight': 0.75}, 4: {'weight': 0.375}}),
          (4, {2: {'weight': 1.2}, 3: {'weight': 0.375}})]
```

Convenient access to all edges is achieved with the `edges` method.

```
In [54]: for (u,v,d) in FG.edges(data='weight'):
             if d<0.5: print('(%d, %d, %.3f)'%(n,nbr,d))
```

```
(4, 3, 0.125)
(4, 3, 0.375)
```

22.6 Adding attributes to graphs, nodes, and edges

Attributes such as weights, labels, colors, or whatever Python object you like, can be attached to graphs, nodes, or edges.

Each graph, node, and edge can hold key/value attribute pairs in an associated attribute dictionary (the keys must be hashable). By default these are empty, but attributes can be added or changed using `add_edge`, `add_node` or direct manipulation of the attribute dictionaries named `G.graph`, `G.node` and `G.edge` for a graph `G`.

22.6.1 Graph attributes

Assign graph attributes when creating a new graph

```
In [55]: G = nx.Graph(day="Friday")
         G.graph
```

```
Out[55]: {'day': 'Friday'}
```

Or you can modify attributes later

```
In [56]: G.graph['day']='Monday'
         G.graph
```

```
Out[56]: {'day': 'Monday'}
```

22.6.2 Node attributes

Add node attributes using `add_node()`, `add_nodes_from()` or G.node

```
In [57]: G.add_node(1,time = '5pm')
```

```
In [58]: G.add_nodes_from([3], time='2pm')
```

```
In [59]: G.node[1]
```

```
Out[59]: {'time': '5pm'}
```

```
In [60]: G.node[1]['room'] = 714
```

```
In [61]: G.nodes(data=True)
```

```
Out[61]: [(1, {'room': 714, 'time': '5pm'}), (3, {'time': '2pm'})]
```

Note that adding a node to `G.node` does not add it to the graph, use `G.add_node()` to add new nodes.

22.6.3 Edge Attributes

Add edge attributes using `add_edge()`, `add_edges_from()`, subscript notation, or `G.edge`.

```
In [62]: G.add_edge(1, 2, weight=4.7 )
```

```
In [63]: G[1][2]
```

```
Out[63]: {'weight': 4.7}
```

```
In [64]: G.add_edges_from([(3,4),(4,5)], color='red')
```

```
In [65]: G.add_edges_from([(1,2,{'color':'blue'}), (2,3,{'weight':8})])
```

```
In [66]: G[1][2]['weight'] = 4.7
```

```
In [67]: G.edge[1][2]['weight'] = 4
```

```
In [68]: G.edges(data=True)
```

```
Out[68]: [(1, 2, {'color': 'blue', 'weight': 4}),
          (2, 3, {'weight': 8}),
          (3, 4, {'color': 'red'}),
          (4, 5, {'color': 'red'})]
```

22.7 Converting Graph to Adjacency matrix

You can use `nx.to_numpy_matrix(G)` to convert `G` to `numpy` matrix. If the graph is weighted, the elements of the matrix are weights. If an edge doesn't exsist, its value will be 0, not Infinity. You have to manually modify those values to Infinity (`float('inf')`)

```
In [69]: nx.to_numpy_matrix(G)
```

```
Out[69]: matrix([[ 0.,  4.,  0.,  0.,  0.],
                 [ 4.,  0.,  8.,  0.,  0.],
                 [ 0.,  8.,  0.,  1.,  0.],
                 [ 0.,  0.,  1.,  0.,  1.],
                 [ 0.,  0.,  0.,  1.,  0.]])
```

```
In [70]: nx.to_numpy_matrix(FG)
```

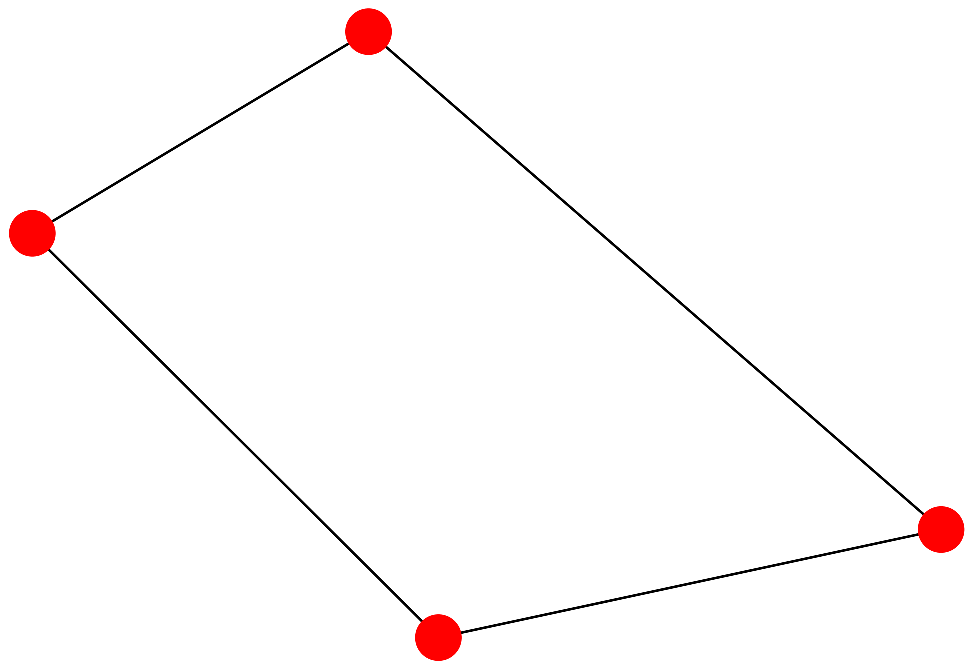
```
Out[70]: matrix([[ 0.   ,  0.125,  0.75 ,  0.   ],
                 [ 0.125,  0.   ,  0.   ,  1.2  ],
                 [ 0.75 ,  0.   ,  0.   ,  0.375],
                 [ 0.   ,  1.2  ,  0.375,  0.   ]])
```

22.8 Drawing graphs

NetworkX is not primarily a graph drawing package but basic drawing with Matplotlib as well as an interface to use the open source Graphviz software package are included. These are part of the networkx.drawing package and will be imported if possible

```
In [71]: %matplotlib inline
         import matplotlib.pyplot as plt
```

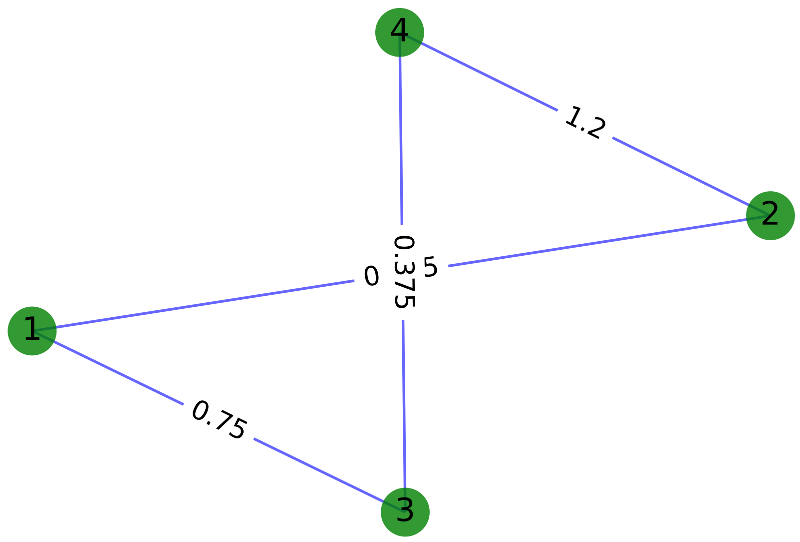
```
In [72]: nx.draw(FG)
```



Now we shall draw the graph using graphviz layout

```
In [73]: from networkx.drawing.nx_agraph import graphviz_layout
         pos = graphviz_layout(FG)
         plt.axis('off')
         nx.draw_networkx_nodes(FG,pos,node_color='g',alpha = 0.8)  # draws nodes
         nx.draw_networkx_edges(FG,pos,edge_color='b',alpha = 0.6)  # draws edges
         nx.draw_networkx_edge_labels(FG,pos,edge_labels = nx.get_edge_attributes(FG,'weight'))
↪  # edge lables
         nx.draw_networkx_labels(FG,pos) # node lables
```

```
Out[73]: {1: <matplotlib.text.Text at 0x7f2e2eecacc0>,
          2: <matplotlib.text.Text at 0x7f2e2eecaba8>,
          3: <matplotlib.text.Text at 0x7f2e2ee97e80>,
          4: <matplotlib.text.Text at 0x7f2e2ee97be0>}
```



22.9 Going Further

We have only seen the basic graph functionalities. In addition to this, NetworkX provides many Graph Algorithms, and Many types of Graphs. Interested reader can look at Official Documentation[12]

[12] https://networkx.readthedocs.io/en/stable/

Part III

Exploring openanalysis

23

Introduction to openanalysis

In our daily life, we encounter many algorithms. Knowingly or Unknowingly, algorithms make our life easier. Analysis of algorithms is a special field of interest in Computer Science. Analysis evaluates the algorithm, and leads to invention of faster algorithms. Visualization leads to the better understanding of how algorithms work. The package `openanalysis` is inteded as a tool for analyzing and visualizing algorithms.

23.1 Types of supported algorithms

The following types of algorithms are currently supported. We plan to support more kind of algorithms in the future.

- Comparison based Sorting Algorithms (Analysis + Visualization)
- Comparison based Searching Algorithms (Analysis)
- Comparison based Pattern Matching Algorithms (Analysis)
- Data Structures and Related algorithms (Visualization)
- Graph Algorithms based on Tree Growth technique (Visualizaiton)
- Graph Algorithms utilizing Matrix and Dynamic Programming (Visualization)

23.2 Setting up openanalysis

23.2.1 Dependency Binary Packages

`openanalysis` expects few binary packages to be installed, which are not installed automatically by the installer. In Linux, you can install these packages via your package manager. For Windows, grab the downloads from their websites.

- graphviz[13]
- ffmpeg[14]
- `libgraphviz-dev` for compiling `pygraphviz` in Linux
- `pkg-config` for compiling `pygraphviz` in Linux
- `python3-tk` as `matplotlib` backend in Linux

[13] http://www.graphviz.org/
[14] https://johnvansickle.com/ffmpeg/

- Visual C++ 2015 Build Tools[15] for compiling `pygraphviz` in Windows
- Python 3.5 or later

23.2.2 Installation

```
pip install openanalysis  # Or pip3 depending on your configuration
```

If all things go well, you have a working installation of `openanalysis`.

23.3 Inside the package

`openanalysis` has following package structure.

```
openanalysis/
├── base_data_structures.py          - Provides PriorityQueue and UnionFind data  structures
├── datastructures.py                - Provides classes for Data Structure Visualization
├── matrix_animator.py               - Provides classes for DP based Graph algorithms
├── searching.py                     - Provides classes for Sorting algorithms
├── sorting.py                       - Provides classes for Searching algorithms
├── string_matching.py               - Provides classes for String Matching algorithms
└── tree_growth.py                   - Provides classes for Tree growth based Graph␣
↪Algorithms
```

23.4 `importing` the modules

Since `openanalysis` root does not have any classes as is, we will import methods from its modules. In further chapters, we shall see the purpose of every modules and shall use it.

23.5 Key factor for analysis

In Computer Science, running time of algorithms is greately considered. Every alorithm solves the given instance of problem by performing some basic operation. The time taken by the algorithm is directly proportional to number of basic operations it has performed.

In normal working environment, time taken by the algorithm to solve a problem is affected by task scheduling performed by OS. We have to fit the obtained running time data in order to analyse the algorithm. Instead of using running time as a key for analysis, we will use number of basic operations as a key in `openanalysis`.

This change in key factor implies, we have to adhere to a standard for implementing algorithms. In fact, `openanalysis` provides such standards, either in the form of rules, or in the form of Base Classes. We shall see those rules in upcomming chapter. In future builds, we plan to include Time-based analysis also.

[15] http://landinghub.visualstudio.com/visual-cpp-build-tools

24 Sorting Analysis

Consider a finite collection of orderable elements. Re-arranging that collection, so that the collection is completely ordered is called sorting. There are many techiniques to sort a collection. Following are some of the comparision based Sorting Algorithms.

- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort
- Quick Sort
- Heap Sort

Before looking at the analysis part, we shall examine the Language in built methods to sorting

24.1 sorted(collection,reverse = False[,key])

This function takes an iterable as argument, and returns it in sorted form based on `key`. If `key` is not given, sorting is done according to default comparision rules. Let's see the examples and understand the working of `sorted()`. If `reverse` is `True`, reversed collection is returned after sorting.

```
In [1]: x = list(range(10))
        import random
        random.shuffle(x)

In [2]: x

Out[2]: [6, 7, 9, 0, 4, 5, 8, 2, 1, 3]

In [3]: sorted(x)

Out[3]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [4]: import math
        y  = sorted(x,key = lambda x: math.sin(x))  # Sort elements of x in increasing order of
 ↪   their sines
        y

Out[4]: [5, 4, 6, 0, 3, 9, 7, 1, 2, 8]

In [5]: [math.sin(i) for i in y]

Out[5]: [-0.9589242746631385,
         -0.7568024953079282,
         -0.27941549819892586,
         0.0,
```

```
0.1411200080598672,
0.4121184852417566,
0.6569865987187891,
0.8414709848078965,
0.9092974268256817,
0.9893582466233818]
```

Note how the elements of `sin(y)` are in increasing order.

24.2 Standard `import` statement

```
In [2]: from openanalysis.sorting import SortingAlgorithm,SortAnalyzer
        import numpy as np    # for doing vstack()
```

`SortingAlgorithm` is the base class providing the standards to implement sorting algorithms, `SortAnalyzer` visualizes and analyses the algorithm

24.3 `SortingAlgorithm` class

Any sorting algorithm, which has to be implemented, has to be derived from this class. Now we shall see data members and member functions of this class.

24.3.1 Data Members

- `name` - Name of the Sorting Algorithm
- `count` - Holds the number of basic operations performed
- `hist_arr` - A 2D `numpy` array, holding the instances of array, as exchange is performed

24.3.2 Member Functions

- `__init__(self, name):` - Initializes algorithm with a `name`
- `sort(self, array, visualization):` - The base sorting function. Sets `count` to 0. `array` is 1D `numpy` array, `visualization` is a `bool` indicating whether `array` has to be `vstack`ed into `hist_arr`

24.4 An example .... Bubble Sort

Now we shall implement the class `BubbleSort`

```
In [7]: class BubbleSort(SortingAlgorithm):                           # Derived from
↪ SortingAlgorithm
            def __init__(self):
                SortingAlgorithm.__init__(self, "Bubble Sort")        # Initializing with name

            def sort(self, array, visualization=False):               # MUST have this signature
                SortingAlgorithm.sort(self, array, visualization)     # sets self.count to 0
                for i in range(0, array.size):                        # Not len(array)
                    exch = False
                    for j in range(0, array.size - i - 1):
                        self.count += 1                               # Increment self.count after
↪ each basic operation
                        if array[j] > array[j + 1]:
                            array[j], array[j + 1] = array[j + 1], array[j]
                            exch = True
```

```
                    if visualization:
                        self.hist_array = np.vstack([self.hist_array, array]) # Save the
↪   current state to hist_array
                if not exch:
                    break
            if visualization:
                self.hist_array = np.vstack([self.hist_array, array])  # Save the final
↪   state to hist_array
```

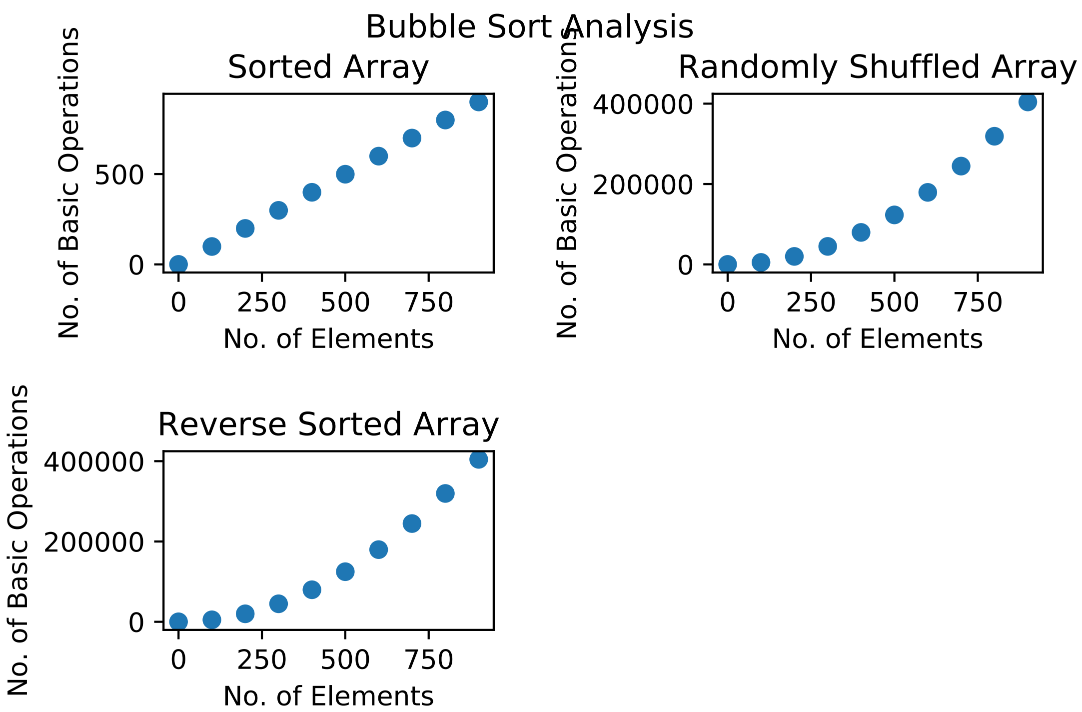
24.5 `SortAnalyzer` class

This class provides the visualization and analysis methods. Let's see its methods in detail

- `__init__(self, sorter)`: Initializes visualizer with a Sorting Algorithm.
  - `sorter` is a class, which is derived from `SortingAlgorithm`
- `visualize(self, num=100, save=False)`: Visualizes the given algorithm with a randomly shuffled array.
  - `num` size of randomly shuffled array
  - `save` is `True` means animation is saved in `output/`
- `analyze(self, maxpts=1000)`:
  - Plots the running time of sorting algorithm by sorting for 3 cases
  - Already Sorted array, reverse sorted array and Shuffled array
  - Analysis is done by inputting randomly shuffled integer arrays with size staring from 100, and varying upto `maxpts` in the steps of 100, and counting the number of basic operations
  - `maxpts` - Upper bound on size of elements chosen for analysing efficiency

```
In [8]: bubble_visualizer = SortVisualizer(BubbleSort)
```

```
In [9]: bubble_visualizer.efficiency()
```



As you can see in the above plot, BubbleSort takes $\mathcal{O}(n)$ time on best case and $\mathcal{O}(n^2)$ time on both avarage and worst cases

You can call the visualize function as shown below and see the 'mp4' file saved at output/ folder

```
bubble_visualizer.visualize(save=True)
```

24.6 compare(algs)

algs is a list of classes derived from SortingAlgorithm. It performs tests and plots the bar graph comparing the number of basic operations performed by each algorithm.

24.7 Why use a class if sorting could be done using a function

We have just seen how BubbleSort is implemented. Every sorting algorithm is not as simple as BubbleSort. QuickSort and MergeSort needs several auxiliary methods to work with. If they are scattered throughout the code, they decrease the readability. So it is better to pack everything in a class.

24.8 Example File

You can see more examples at Github[16]

[16] https://github.com/OpenWeavers/openanalysis/blob/master/analysistest/sorting.py

25 Searching Analysis

Consider a finite collection of element. Finding whether element exsists in collection is known as Searching. Following are some of the comparision based Searching Algorithms.

- Linear Search
- Binary Search

Before looking at the analysis part, we shall examine the Language in built methods to searching

25.1 The `in` operator and `list.index()`

We have already seen the `in` operator in several contexts. Let's see the working of `in` operator again

```
In [1]: x = list(range(10))

In [2]: x

Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: 6 in x

Out[3]: True

In [4]: 100 in x

Out[4]: False

In [5]: x.index(6)

Out[5]: 6

In [6]: x.index(100)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-0f87238c301b> in <module>()
----> 1 x.index(100)

ValueError: 100 is not in list
```

25.2 Standard `import` statement

```
In [16]: from openanalysis.searching import SearchingAlgorithm,SearchAnalyzer
         %matplotlib inline
         %config InlineBackend.figure_formats={"svg", "pdf"}
```

`SearchingAlgorithm` is the base class providing the standards to implement searching algorithms, `SearchAnalyzer` analyses the algorithm

25.3 `SearchingAlgorithm` class

Any searching algorithm, which has to be implemented, has to be derived from this class. Now we shall see data members and member functions of this class.

25.3.1 Data Members

- `name` - Name of the Searching Algorithm
- `count` - Holds the number of basic operations performed

25.3.2 Member Functions

- `__init__(self, name):` - Initializes algorithm with a `name`
- `search(self, array, key):` _ The base searching function. Sets `count` to 0. `array` is 1D `numpy` array,`key` is the key of element to be found out

25.4 An example . . . . Binary Search

Now we shall implement the class `BinarySearch`

```
In [17]: class BinarySearch(SearchingAlgorithm):                         # Inheriting
             def __init__(self):
                 SearchingAlgorithm.__init__(self, "Binary Search") # Initailizing with name

             def search(self, arr, key):
                 SearchingAlgorithm.search(self, arr, key)          # call base class search
                 low, high = 0, arr.size - 1
                 while low <= high:
                     mid = int((low + high) / 2)
                     self.count += 1                                 # Increment for each basic
↪    operation performed
                     if arr[mid] == key:
                         return True
                     elif arr[mid] < key:
                         low = mid + 1
                     else:
                         high = mid - 1
                 return False
```

25.5 `SearchAnalyzer` class

This class provides the visualization and analysis methods. Let's see its methods in detail

- `__init__(self, searcher):` Initializes visualizer with a Searching Algorithm.
  - `searcher` is a class, which is derived from `SearchingAlgorithm`
- `analyze(self, maxpts=1000):`
  - Plots the running time of searching algorithm by searching in 3 cases
  - Key is the first element, Key is the last element, Key at random index

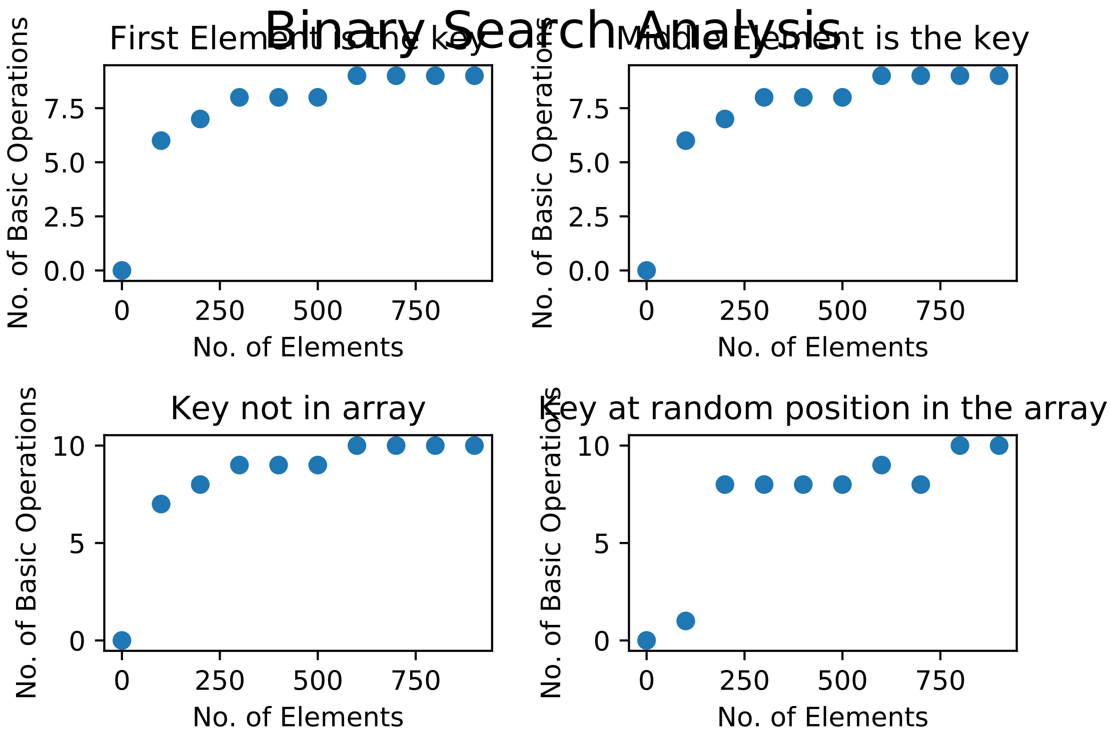
- Analysis is done by inputting sorted integer arrays with size staring from 100, and varying upto `maxpts` in the steps of 100, and counting the number of basic operations
- `maxpts` Upper bound on size of elements chosen for analysing efficiency

```
In [18]: bin_visualizer = SearchAnalyzer(BinarySearch)
```

```
<matplotlib.figure.Figure at 0x7ff57329b978>
```

```
In [19]: bin_visualizer.analyze(progress=False)
```

```
Please wait while analyzing Binary Search Algorithm
```



Note $\mathcal{O}(\log n)$ performance in all cases

25.6 `compare(algs)`

`algs` is a list of classes derived from `SearchingAlgorithm`. It performs tests and plots the bar graph comapring the number of basic operations performed by each algorithm.

25.7 Example File

You can see more examples at Github[17]

[17] https://github.com/OpenWeavers/openanalysis/blob/master/analysistest/searching.py

26 String Matching Analysis

Consider a string of finite length m Let it be T. Finding whether a string P of length n exsists in T is known as String Matching, Following is some of the comparision based String Matching Algorithms.

- Brute Force String Matching Algorithm
- Horspool String Matching
- Boyer - Moore String Matching

Before looking at the analysis part, we shall examine the Language in built methods to sorting

26.1 The `in` operator and `str.index()`

We have already seen the `in` operator in several contexts. Let's see the working of `in` operator again

```
In [1]: x = 'this is some random text used for illustrative purposes'

In [2]: x

Out[2]: 'this is some random text used for illustrative purposes'

In [3]: 'this' in x

Out[3]: True

In [4]: 'not' in x

Out[4]: False

In [5]: x.index('is')

Out[5]: 2

In [6]: x.index('not')

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-a1f052cc5af7> in <module>()
----> 1 x.index('not')

ValueError: substring not found
```

26.2 Standard `import` statement

```
In [10]: from openanalysis.string_matching import
↪  StringMatchingAlgorithm,StringMatchingAnalyzer
         %matplotlib inline
```

```
%config InlineBackend.figure_formats={"svg", "pdf"}
```

`StringMatchingAlgorithm` is the base class providing the standards to implement sorting algorithms, `SearchVisualizer` visualizes and analyses the algorithm

26.3 `StringMatchingAlgorithm` class

Any String Matching Algorithm, which has to be implemented, has to be derived from this class. Now we shall see data members and member functions of this class.

26.3.1 Data Members

- `name` - Name of the Searching Algorithm
- `count` - Holds the number of basic operations performed

26.3.2 Member Functions

- `__init__(self, name):` - Initializes algorithm with a `name`
- `match(self, text, pattern)` _ The base String Matching function. Sets `count` to 0.

26.4 An example .... Horspool String Matching Algorithm

Now we shall implement the class `Horspool`

```
In [11]: class Horspool(StringMatchingAlgorithm):             # Must derive from
↪ StringMatchingAlgorithm
             def __init__(self):
                 StringMatchingAlgorithm.__init__(self, "Hosrpool String Matching")
                 self.shift_table = {}
                 self.pattern = ''

             def generate_shift_table(self, pattern):              # class is needed to include
↪ helper methods
                 self.pattern = pattern
                 for i in range(0, len(pattern) - 1):
                     self.shift_table[pattern[i]] = len(pattern) -i - 1

             def match(self, text: str, pattern: str):
                 StringMatchingAlgorithm.match(self, text, pattern)
                 self.generate_shift_table(pattern)
                 i = len(self.pattern) - 1
                 while i < len(text):
                     j = 0
                     while j < len(self.pattern) and text[i-j] ==
↪ self.pattern[len(self.pattern)-1-j]:
                         j += 1
                     self.count += j                                      # Increment count
↪ here
                     if j == len(self.pattern):
                         return i-len(self.pattern)+1
                     if text[i] in self.shift_table:
                         i += self.shift_table[text[i]]
                     else:
                         i += len(self.pattern)
                 return -1
```

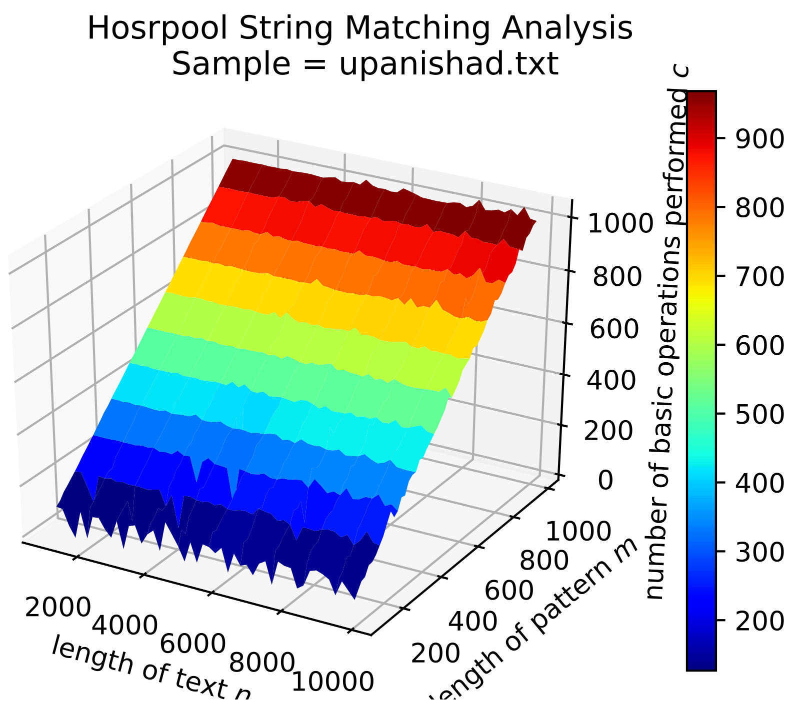
26.5 `StringMatchingAnalyzer` class

This class provides the visualization and analysis methods. Let's see its methods in detail

- `__init__(self, matching)`: Initializes visualizer with a String Matching Algorithm.
  - `searcher` is a class, which is derived from `StringMatchingAlgorithm`
- `analyze(self,progress = True)`:
  - Plots the number of basic operations performed
  - Both Text length and Pattern Length are varied
  - Samples are chosen randomly from pre defined large data
  - `progress` indicates whether Progress Bar has to be shown or not

```
In [13]: StringMatchingAnalyzer(Horspool).analyze(progress=False)
```

```
Please wait while analysing Hosrpool String Matching algorithm
```



Note $\mathcal{O}(n)$ performance of algorithm

26.6 Example File

You can see more examples at Github[18]

[18] https://github.com/OpenWeavers/openanalysis/blob/master/analysistest/string_matching.py

27

Data Structures

Data structures are a concrete implementation of the specification provided by one or more particular abstract data types (ADT), which specify the operations that can be performed on a data structure and the computational complexity of those operations.

Different kinds of data structures are suited for different kinds of applications, and some are highly specialized to specific tasks. For example, relational databases commonly use B-tree indexes for data retrieval, while compiler implementations usually use hash tables to look up identifiers.

Usually, efficient data structures are key to designing efficient algorithms.

27.1 Standard `import` statement

```
In [1]: from openanalysis.data_structures import DataStructureBase, DataStructureVisualization
        import gi.repository.Gtk as gtk    # for displaying GUI dialogs
```

`DataStructureBase` is the base class for implementing data structures

`DataStructureVisualization` is the class that visualizes data structures in GUI

27.2 `DataStructureBase` class

Any data structure, which is to be implemented, has to be derived from this class. Now we shall see data members and member functions of this class:

27.2.1 Data Members

- `name` - Name of the DS
- `file_path` - Path to store output of DS operations

27.2.2 Member Functions

- `__init__(self, name, file_path)` - Initializes DS with a `name` and a `file_path` to store the output
- `insert(self, item)` - Inserts `item` into the DS

- `delete(Self, item)` - Deletes `item` from the DS, if `item` is not present in the DS, throws a `ValueError`
- `find(self, item)` - Finds the `item` in the DS returns `True` if found, else returns `False`similar to `__contains__(self, item)`
- `get_root(self)` - Returns the root (for graph and tree DS)
- `get_graph(self, rt)` - Gets the `dict` representation between the parent and children (for graph and tree DS)
- `draw(self, nth=None)` - Draws the output to visualize the operations performed on the DS `nth` is used to pass an `item` to visualize a find operation

27.3 `DataStructureVisualization` class

This class is used for visualizing data structures in a GUI (using GTK+ 3). Now we shall see data members and member functions of this class:

27.3.1 Data Members

- `ds` - Any DS, which is an instance of DataStructureBase

27.3.2 Member Functions

- `__init__(self, ds)` - Initializes `ds` with an instance of DS that is to be visualized
- `run(self)` - Opens a GUI window to visualize the DS operations

27.4 An example . . . . . Binary Search Tree

Now we shall implement the class `BinarySearchTree`

```
In [2]: class BinarySearchTree(DataStructureBase):                          # Derived from
↪   DataStructureBase

            class Node:                                                     # Class for creating
↪   a node
                def __init__(self, data):
                    self.left = None
                    self.right = None
                    self.data = data

                def __str__(self):
                    return str(self.data)

            def __init__(self):
                DataStructureBase.__init__(self, "Binary Search Tree", "t.png")       #
↪   Initializing with name and path
                self.root = None
                self.count = 0

            def get_root(self):                                              # Returns root node
↪   of the tree
                return self.root

            def insert(self, item):                                          # Inserts item into
↪   the tree
```

```
            newNode = BinarySearchTree.Node(item)
            insNode = self.root
            parent = None
            while insNode is not None:
                parent = insNode
                if insNode.data > newNode.data:
                    insNode = insNode.left
                else:
                    insNode = insNode.right
            if parent is None:
                self.root = newNode
            else:
                if parent.data > newNode.data:
                    parent.left = newNode
                else:
                    parent.right = newNode
            self.count += 1

        def find(self, item):                                       # Finds if item is
↪   present in tree or not
            node = self.root
            while node is not None:
                if item < node.data:
                    node = node.left
                elif item > node.data:
                    node = node.right
                else:
                    return True
            return False

        def min_value_node(self):                                   # Returns the
↪   minimum value node
            current = self.root
            while current.left is not None:
                current = current.left
            return current

        def delete(self, item):                                     # Deletes item from
↪   tree if present
                                                                    # else shows Value
↪   Error
            if item not in self:
                dialog = gtk.MessageDialog(None, 0, gtk.MessageType.ERROR,
                                           gtk.ButtonsType.CANCEL, "Value not found ERROR")
                dialog.format_secondary_text(
                    "Element not found in the %s" % self.name)
                dialog.run()
                dialog.destroy()
            else:
                self.count -= 1
                if self.root.data == item and (self.root.left is None or self.root.right is
↪   None):
                    if self.root.left is None and self.root.right is None:
                        self.root = None
                    elif self.root.data == item and self.root.left is None:
                        self.root = self.root.right
                    elif self.root.data == item and self.root.right is None:
                        self.root = self.root.left
                    return self.root
                if item < self.root.data:
                    temp = self.root
                    self.root = self.root.left
                    temp.left = self.delete(item)
```

```
                    self.root = temp
                elif item > self.root.data:
                    temp = self.root
                    self.root = self.root.right
                    temp.right = self.delete(item)
                    self.root = temp
                else:
                    if self.root.left is None:
                        return self.root.right
                    elif self.root.right is None:
                        return self.root.left
                    temp = self.root
                    self.root = self.root.right
                    min_node = self.min_value_node()
                    temp.data = min_node.data
                    temp.right = self.delete(min_node.data)
                    self.root = temp
                return self.root

        def get_graph(self, rt):                                        # Populates
↪   self.graph with elements depending
                                                                        # upon the
↪   parent-children relation
            if rt is None:
                return
            self.graph[rt.data] = {}
            if rt.left is not None:
                self.graph[rt.data][rt.left.data] = {'child_status': 'left'}
                self.get_graph(rt.left)
            if rt.right is not None:
                self.graph[rt.data][rt.right.data] = {'child_status': 'right'}
                self.get_graph(rt.right)
```

Now, this program can be executed as follows:

```
In [3]: DataStructureVisualization(BinarySearchTree).run()
```

```
In [4]: import io
        import base64
        from IPython.display import HTML

        video = io.open('../res/bst.mp4', 'r+b').read()
        encoded = base64.b64encode(video)
        HTML(data='''<video alt="test" width="500" height="350" controls>
                        <source src="data:video/mp4;base64,{0}" type="video/mp4" />
                     </video>'''.format(encoded.decode('ascii')))
```

```
Out[4]: <IPython.core.display.HTML object>
```

27.5 Example File

You can see more examples at Github[19]

[19] https://github.com/OpenWeavers/openanalysis/blob/master/analysistest/data_structures.py

28

Tree Growth based Graph Algorithms

These class of algorithms takes a Graph as input, and generates Tree, which consists of some of edges of input Graph, which are selected according to particular criteria. Some examples are

- DFS
- BFS
- Minimum Spanning Tree Problem (Prim's and Kruskal's Algorithm)
- Single Source Shortest Path Problem (Dijkstra's Algorithm)

28.1 Standard `import` statement

```
In [6]: import openanalysis.tree_growth as TreeGrowth
```

28.2 Implementation Notes

- The algorithm should be implemented as a method
- The algorithm works on a `networkx` graph
- All algorithms start building the tree from a given source, But if source is not given, select source as the first node of Graph

```
def algorithm_name(G,source = None):
    if source is None:
        source = G.nodes()[0]
    # do other work now
```

- As soon as node `v` is visited from node `u`, `yield` the tuple containing them

```
# Assume that visiting is done
yield (u,v)
```

- To make your life easy, some data structures which comes handy while working with Graphs are included in `OpenAnalysis.base_data_structures`

28.3 Example - Dijkstra's Algorithm

Dijkstra's Algorithm finds minimum spanning tree of a graph in greedy manner. The algorithm is given below

ALGORITHM *Dijkstra*(G, s)

//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph $G = \langle V, E\rangle$ with nonnegative weights
// and its vertex s
//Output: The length d_v of a shortest path from s to v
// and its penultimate vertex p_v for every vertex v in V
Initialize(Q) //initialize priority queue to empty
for every vertex v in V
  $d_v \leftarrow \infty$; $p_v \leftarrow$ **null**
  Insert(Q, v, d_v) //initialize vertex priority in the priority queue
$d_s \leftarrow 0$; *Decrease*(Q, s, d_s) //update priority of s with d_s
$V_T \leftarrow \varnothing$
for $i \leftarrow 0$ **to** $|V| - 1$ **do**
  $u^* \leftarrow$ *DeleteMin*(Q) //delete the minimum priority element
  $V_T \leftarrow V_T \cup \{u^*\}$
  for every vertex u in $V - V_T$ that is adjacent to u^* **do**
    if $d_{u^*} + w(u^*, u) < d_u$
      $d_u \leftarrow d_{u^*} + w(u^*, u)$; $p_u \leftarrow u^*$
      Decrease(Q, u, d_u)

Fig. 28.1: Dijkstra's Algorithm

28.4 Implementation

Since we need a Priority Queue here, Let's `import` it

```
In [7]: from openanalysis.base_data_structures import PriorityQueue
```

Now, Let's implement the algorithm

```
In [8]: def dijkstra(G, source=None):                   # This signature is must
            if source is None: source = G.nodes()[0]   # selecting root as source
            V = G.nodes()
            dist, prev = {}, {}
            Q = PriorityQueue()
            for v in V:
                dist[v] = float("inf")
                prev[v] = None
                Q.add_task(task=v, priority=dist[v])
            dist[source] = 0
            Q.update_task(task=source, new_priority=dist[source])
            visited = set()
            for i in range(0, len(G.nodes())):
                u_star = Q.remove_min()
                if prev[u_star] is not None:
```

```
                    yield (u_star, prev[u_star])    # yield the edge as soon as we visit the
↪   nodes
                visited.add(u_star)
                for u in G.neighbors(u_star):
                    if u not in visited and dist[u_star] + G.edge[u][u_star]['weight'] <
↪   dist[u]:
                        dist[u] = dist[u_star] + G.edge[u][u_star]['weight']
                        prev[u] = u_star
                        Q.update_task(u, dist[u])
```

Note how implementation looks similiar to the algorithm, except the `if` block, which is used to `yield` the edges.

28.5 Visualizing the Algorithm

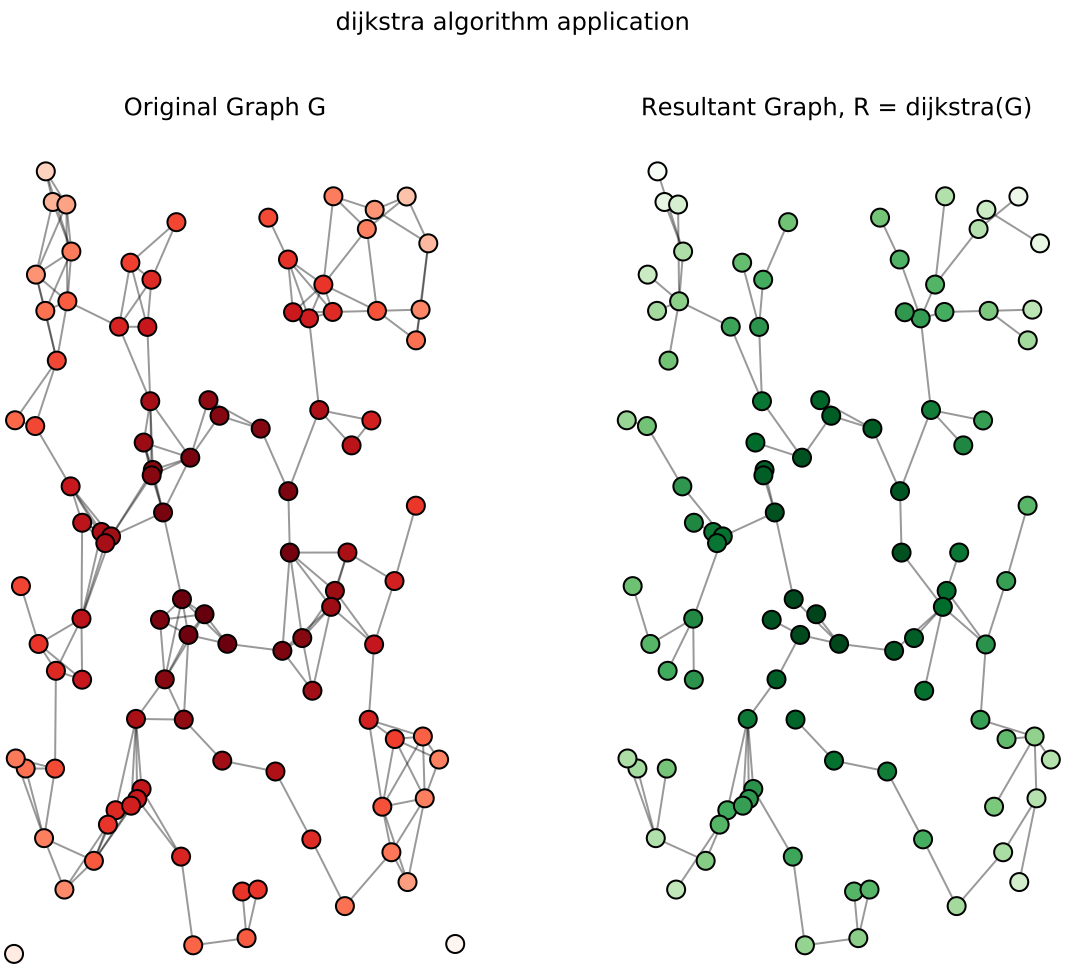
- `apply_to_graph(fun)`: Creates Random Geometric Graph of 100 nodes and applies `fun` on it to build the tree. After building the tree, it shows original graph and the tree side by side
- `tree_growth_visualizer(fun)`: Creates Random Geometric Graph of 100 nodes and applies `fun` on it to build the tree. Saves the animation of building the tree in `output/` folder

28.6 Random Geometric Graph

Random Geometric Graph is created using two parameters. Number of nodes n, and radiuus r. n points are chosen randomly on plane. The edge between 2 nodes is created if and only if the distance between 2 nodes is less than r

```
import networkx as nx
G = nx.random_geometric_graph(100,2.3) # n,r
pos = nx.get_node_attribute('pos')
```

```
In [10]: TreeGrowth.apply_to_graph(dijkstra)
```



After executing

```
TreeGrowth.tree_growth_visualizer(dijkstra)
```

go to `output/` directory to see `mp4` files

28.7 Example File

You can see more examples at Github[20]

[20] https://github.com/OpenWeavers/openanalysis/blob/master/analysistest/tree_growth.py

29 Dynamic Programming based Graph Algorithms

These class of algorithms takes a Graph as input, using it's adjacency matrix , generates result matrix. Some examples are

- Transitive clousure of graph
- All pair shortest path problem

29.1 Standard `import` statement

```
In [1]: from openanalysis.matrix_animator import MatrixAnimator
        import numpy as np       # Needed to work with arrays
```

29.2 Implementation Notes

- The algorithm should be implemented as a method
- The algorithm works on a `networkx` graph
- Obtain the adjacency matrix as follws

```
def algorithm_name(G):
    import networkx as nx
    M = nx.to_numpy_matrix(G)
    # do other work now
```

- If Graph is weighted, matrix elements are weights. Default weight for an edge is 1. If an edge doesn't exsist, its weight will be treated as 0. When working with weighted graphs, You have to **MANUALLY** set those weigthts to infinity.

```
m, n = M.shape
for i in range(0, n):
    for j in range(0, n):
        if i != j and D[i, j] == 0:
           M[i, j] = float('inf')
```

- After each change in matrix, yield matrix, `yield` copy of current version of matrix, along with a tuple containing current 3 co-ordinates at which change is caused

```
yield np.array(D), (i, j, k)
```

29.3 Example Warshall- Floyd Algorithm

Warshal-Floyd Algorithm computes All Pair Shortest Paths of a Graph using its adjacency matrix

Now, Let's implement the algorithm

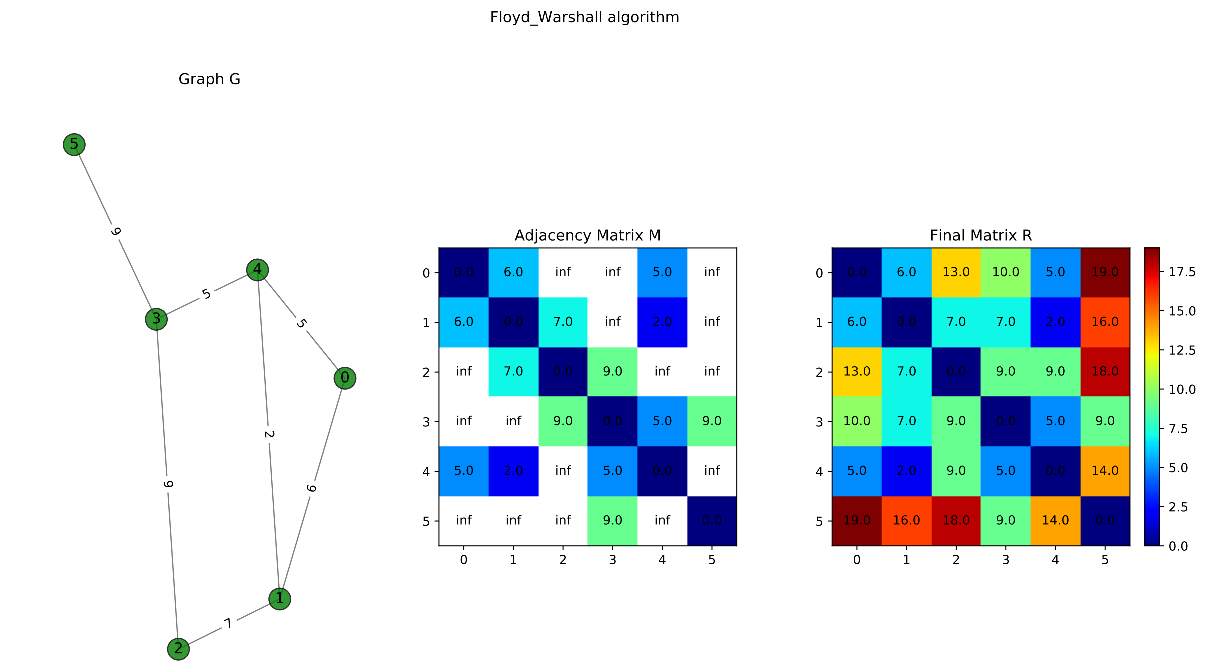
```
In [2]: def Floyd_Warshall(G):                  # Must have signature like this
            D = nx.to_numpy_matrix(G)           # Obtaining Adj. matrix
            m, n = D.shape
            for i in range(0, n):               # Making non-diagonal zeros to infinity, as it is
↪   a Weighted Graph
                for j in range(0, n):
                    if i != j and D[i, j] == 0:
                        D[i, j] = float('inf')
            yield np.array(D), (0, 0, 0)        # Starting yield
            count = 0
            for k in range(0, n):
                for i in range(0, n):
                    for j in range(0, n):
                        if D[i, j] > D[i, k] + D[k, j]:
                            yield np.array(D), (i, j, k)  # yield as array changes
                            D[i, j] = D[i, k] + D[k, j]
                        count += 1
            yield np.array(D), (0, 0, 0)
```

29.4 Visualizing the Algorithm - `MatrixAnimator` class

- `__init__(self, fn, G):`
  - `fn` : A function yielding matrix along with 3-tuple
  - `G` : Graph on which `fn` has to be applied and visualized
- `animate(self, save=False):`
  - `save` is `True` implies animation is saved in `output/` folder
- `apply_to_graph(self, show_graph=True):`
  - applies `self.fn` to `self.G` and displays the result
  - `show_graph` is `True` implies Graph is shown along with adjacency matrix and final matrix

Here we shall create a matrix from `numpy` array, and assign random weights to its edges. Then we apply our function to graph

```
In [3]: import networkx as nx
        M = nx.from_numpy_matrix(
            np.matrix(
                [[0, 1, 0, 0, 1, 0],
                 [1, 0, 1, 0, 1, 0],
                 [0, 1, 0, 1, 0, 0],
                 [0, 0, 1, 0, 1, 1],
                 [1, 1, 0, 1, 0, 0],
                 [0, 0, 0, 1, 0, 0]]
            ))
        import random
        for u, v in M.edges():
            M.edge[u][v]['weight'] = random.randint(1, 10)
        animator = MatrixAnimator(Floyd_Warshall, M)
        animator.apply_to_graph()
```



After executing

```
animator.animate(save=True)
```

go to `output/` directory to see the `mp4` files

29.5 Example File

You can see more examples at Github[21]

[21] https://github.com/OpenWeavers/openanalysis/blob/master/analysistest/matrix_dp.py

Part IV

API Referance

30

openanalysis.base_data_structures module

`class openanalysis.base_data_structures.UnionFind`

Union-find data structure.

Each unionFind instance X maintains a family of disjoint sets of hashable objects, supporting the following two methods:

- X[item] returns a name for the set containing the given item. Each set is named by an arbitrarily-chosen one of its members; as long as the set remains unchanged it will keep the same name. If the item is not yet part of a set in X, a new singleton set is created for it.
- X.union(item1, item2, …) merges the sets containing each item into a single larger set. If any item is not yet part of a set in X, it is added to X as one of the members of the merged set.

`union(`**objects*`)`

Find the sets containing the objects and merge them all.

`class openanalysis.base_data_structures.PriorityQueue`

A simple Priority Queue Implementation for usage in algorithms. Internally uses heapq to maintain min-heap and tasks are added as tuples (priority,task) to queue. To make the order of tasks with same priority clear, count of element insertion is added to the tuple, making it as (priority,count,task), which means that tasks are first ordered by priority then by count

`add_task(`*task*, *priority*`)`

Add a task to priority queue

Parameters

- **`task`** – task to be added to queue
- **`priority`** – priority of the task, must be orderable

`remove(`*task*`)`

Removes the tasks from Queue Currently it takes O(n) time to find , and O(log n) to remove, making it O(n) further improvements can be done

Parameters **`task`** – task to removed from the Queue

`remove_min()`

Removes the minimum element of heap

Returns task with less priority

`update_task(`*task*, *new_priority*`)`

Updates the priority of exsisting task in Queue Updation is implemented as deletion and insertion, takes O(n) time further improvements can be done

Parameters

- `task` – task to be updated
- `new_priority` – new value of priority

31 openanalysis.data_structures module

`class openanalysis.data_structures.DataStructureBase(`*name*, *file_path*`)`
Base class for implementing Data Structures

`delete(`*item*`)`
Delete the item from Data Structure While removing, delete item from self.graph and modify the edges if necessary :param item: item to be deleted

`draw(`*nth=None*`)`

`find(`*item*`)`
Finds the item in Data Structure :param item: item to be searched :return: True if item in self else False also can implement __contains__(self,item)

`get_graph(`*rt*`)`

`get_root()`
Return the root for drawing purpose :return:

`insert(`*item*`)`
Insert item to Data Structure While inserting, add a edge from parent to child in self.graph :param item: item to be added

`class openanalysis.data_structures.DataStructureVisualization(`*ds*`)`
Class for visualizing data structures in GUI Using GTK+ 3

`action_clicked_cb(`*button*`)`

`on_stage_destroy(`*x*`)`

`run()`

32

openanalysis.matrix_animator module

33

openanalysis.searching module

34

openanalysis.sorting module

35

openanalysis.string_matching module

36

openanalysis.tree_growth module

Python Module Index

o

Index