Project Tenji: Python

Contents

Chapter 1

Getting Started

An algorithm for a given problem is a list of instructions which given an input produces a desired output.

1.1 Find square root

1.1.1 The problem

given a nonnegative number x (the input), approximate its square root.
Interested in an approximation g of $\sqrt{x}$ such that $g * g$ is close enough to x

1.1.2 The algorithm

Algorithm due to Heron of Alexandria, around 2000 years ago:

- input : x and tolerance parameter $\theta > 0$
- start with a guess $g > 0$
- if $g * g$ is close enough to x ($|g * g - x| \leq \theta$) stop.
- Else update g to the value : $(g + \dfrac{x}{g})/2$
- Repeat until $|g * g - x| \leq \theta$
- Output : g

1.2 Search for a number in a sorted list

1.2.1 The problem

Assume that you have a large list of n numbers sorted in non-decreasing order.
Given a number x, check if x is in the list: YES/NO answer.

1.2.2 Binary search algorithm

Taking advantage of the fact that the numbers are sorted:

- Compare x to the middle element in the list
- if equal:
  Stop and output :Yes
- if $x >$ middle:
  narrow down search to upper sub-list excluding middle
- if $x >$ middle:
  narrow down search to lower sub-list excluding middle
- Repeat until :
  x is found
  or the sub-list is empty, in which case output:No

1.3 Binary representation

Binary digit (bit) : 0 or 1
Byte: sequence of 8 bits
In general, using n bits, get 2^n combinations, thus for 8bits, we have $2^8 = 256$ combinations.

In Base 2 representation of integers we use 0 and 1:

$$x_k x_{k-1} \dots x_0 = x_k \times 10^k + x_{k-1} \times 10^{k-1} + \cdots + x_0 \times 10^0$$

Example: $1100 \rightarrow 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 12$

In Base 10 representation of integers we use 0,1,2,3,4,5,6,7,8,9

$$x_k x_{k-1} \dots x_0 = x_k \times 2^k + x_{k-1} \times 2^{k-1} + \cdots + x_0 \times 2^0$$

Chapter 2

Introduction to Python

2.1 Data Types

Python is an interpreter, it guesses the type of a variable from initialization.

2.1.1 int type

used to represent integers.

```
m = 4
n = 1231244536545356l2
p = -234
...
```

2.1.2 float type

Used to represent real number up to limited precision.
Represented using 8 bytes (64 bits) :

- 1 bit for the sign s
- 52 bits the significant part b
- remaining 11 bits for the exponent e to put the floating decimal point

$$(-1)^s \times 1.b \times 2^{\text{offset}-e}$$

```
f  =24.2345
k=-56.45322342
...
```

2.1.3 Bool type

Used to represent Boolean/logical values: True and False

```
a = True
b = (8>3)
c = False
...
```

2.1.4 String type str

It a sequence of characters
Use single quotations or double quotations

```
s = "ab d"
c = 'ab d'
```

Escape sequences:

Escape sequences	Meaning
\n	New line
\"	"
\'	'
\\	\

2.1.5 Casting between types

- To integer
  int(x) function: casts x to integer when possible.

```
int("12") #12
int(13.7) #13
int(True) # 1
```

- to float
  float() function: casts to float when possible

```
float("-14.3") # -14.3
```

- to string
  str() function: cast to string

```
str(-1432.2) # "-1432.2"
```

2.2 Operators

2.2.1 Arithmetic operators

Operators for the int and float types

- Addition (+) : $x + y$
- subtraction (-) : $x - y$
- Multiplication (*) : $x \times y$
- Division (/) : $\frac{x}{y}$
- Power (**) : x^y

All the above are binary operator: $x <$ operator $> y \rightarrow z$
We have also a unary operator minus(-): $-x$ is the negative of x
Except of division if x and y are integer then the result in an integer,in all other cases the results are a float

Operators specific to the int type

- Integer Division (//):
  x//y is the quotient of x/y if y $\neq$ 0, (5//2 is 2)
- Modulo (%): x%y is the remainder of x/y, (5% 2 is 1)

2.2.2 Relational operators

- Equality check (==)
- Not equal (!=)
- Less than (<)
- Greater than (>)
- Less than or equal (<=)
- Greater than or equal (>=)

Comparing two floats

```
x =0.1
y = x*x # 0.010000000000000002
x*x ==0.001# False
# use instead
abs(x*x -0.01) <= 1E-6
```

2.2.3 Logical operators

x	y	x and y
True	False	False
True	True	True
False	True	False
False	False	False

x	y	x or y
True	True	True
True	False	True
False	True	True
False	False	False

x	not x
True	False
False	True

2.2.4 String operators

- Concatenation
  using the + operator, (”abc” +”mnr” gives ”abcmnr”)
- Repetition
  using the * operator,(3*”abc” gives ”abcabcabc”)
- length: a build in function len(),(len(”abc”) gives 3)

2.2.5 Order of precedence

if we skip parentheses in an expression, the following order of precedence will be followed:

1	Power : **
2	Unary operator minus: -
3	Multiplication, division, modulo: *,/, // ,%
4	Addition,Subtraction:+,-
5	Relational: $<, >, \leq, \geq, ==, !=$
6	not
7	and
8	or

2.3 Variables and the assignment operator

In Python, a variable is just a name associated with an object.
In python, variables can change type:

```
    x =12.1 # float
    x = "abc" string
```

2.3.1 Naming variable

A variable can't start with a digit, can't contain a spaces or symbols
It is recommended to capitalize first letter of each word except for initial word (secondNumber) or separate by underscores (second_number)

2.4 Miscellaneous

2.4.1 Input

Use input() function which returns a string.
Then, use the functions int() or float() functions to cast if needed

```
c = float(input('Enter a float: '))
```

2.5 Output

Use the function print(), which takes one or more arguments:

```
print(expression1 ,expression2 ,...)
```

2.5.1 Modules

Here some useful scientific modules:

- math
- numpy : Numerical Python
- scipy: scientific tools for python
- matplotlib : 2D plotting library

2.5.2 Comments

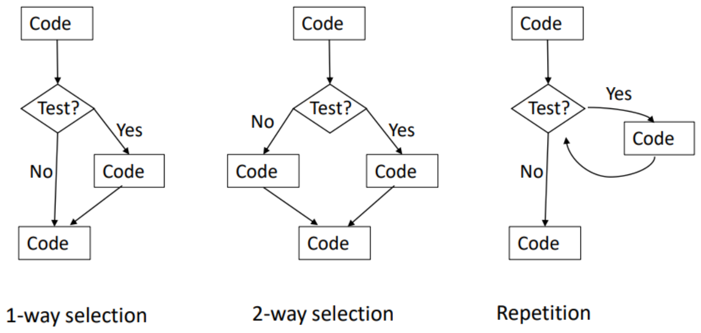
Comments are ignored by python :

- Starting a line with the hash symbol # comments....
- Docstring ””” comments.... ”””

Chapter 3

Selection and Repetition

Selection and repetition flow diagrams:



3.1 Selection

3.1.1 if Structure

Enables the program to branch depending on conditions.
Syntax :

```
if (Boolean expression):
    block of code
```

The block has one ore more statements.
If the Boolean expression evaluates to True, the block is executed. Otherwise it is bypassed.

3.1.2 if-else structure

```
if Boolean expression:
    Block 1 of code
else:
    Block 2 of code
```

If the Boolean expression evaluates to True, the first block is executed.
Otherwise, the second block is executed

3.1.3 Multi-way selection

The two following syntax are equivalent :

The First one:

```
if Boolean expression 1:
  Block 1 of code
elif Boolean expression 2:
  Block 2 of code
elif Boolean expression 3:
  Block 3 of code
...
else:
  Block k of code
```

The Second one:

```
if Boolean expression 1:
  Block 1 of code
else:
  if Boolean expression 2:
    Block 2 of code
  else :
    if Boolean expression 3:
      Block 3 of code
    ...
        else :
          Block k of code
```

3.2 Repetition: While loops and counters

3.2.1 while loop

Enables the program to repeat a task as long as a condition is satisfied.
Syntax:

```
while Boolean expression:
  block of code
```

As long as the Boolean expression evaluates to True, the loop body is executed.

3.2.2 Incrementing counters

Incrementing an integer variable can be done by

```
i+=num # equivalent to i = i+num
```

3.2.3 For loops

Used to simplify syntax of counter controlled while loop:

```
variable = start
while variable < stop:
  code block
  variable = variable + step
```

For-loop syntax:

```
for variable in range(start,stop,step):
  code block
```

The default step is 1 and the default value of start is 0.

3.2.4 break statement

break statement : terminates the loop in which it is contained, and transfer control to the code immediately following the loop.
In nested loops, a break statement in the inner loop only affects the inner loop.

3.3 Bisection method,Finding square-root

- Assume that $x \leq 1$
- We know that $\sqrt{x}$ is in the interval $[x, 1]$
- Compute mid = (x+1)/2
- Compare mid*mid with x
- if abs(mid*mid-x)$\leq$ epsilon,stop
- if mid*mid<x, narrow down search to the interval [mid,1]
- Else, narrow down search to the interval [x,mid]
- Repeat the above process until abs(mid*mid -x)$\leq$ epsilon

Chapter 4

List,tuples,and strings

4.1 List type

4.1.1 Motivation

Using what we know so far (scalar types, selection, and repetition), we can solve problems such as:

- given a sequence of numbers
- find sum
- average
- max

But fall short of basic problems such as: given a sequence of numbers entered by user:

- print them in reverse order
- check if they are distinct
- sort them

In all above three problems, we need to store all the sequence in memory and manipulate it,to do that we need lists.

4.1.2 List type in python

- List is a built-in type :mutable (can be modified) ordered sequence of values, where each value is identified by an index
- Initialization:

  ```
  L = [10,2,"ab",4.4]
  ```

- indexing:

  ```
  L[i] # is the i'th element of L
  ```

  the indexing start from zero

- length function

  ```
  len(L) returns the length of L
  ```

- Read and Write

```
v=L[1] # stores the value of L[1] in v
L[1]=7 # modifies the value of L[1]
```

- Homogenous lists : all elements are of the same type
- Non-homogenous lists: mixed types

4.2 Manipulating lists

4.2.1 Initializing

- if n is an integer, the statement

```
L = [value]*n
```

creates a length-n list L whose entries are all of equal to value.

- Manipulating lists using loops

```
n = len(L)
for i in range(n)
  # proccess L[i]
```

- The range -n,...,-1 is special:

```
L[-1] #is interpreted as L[n-1]
```

4.2.2 Input to list

- Method 1:

```
n = int(input("Enter number n of integers:"))
L = [0]*n
for i in range(n):
  L[i] = int(input("Enter integer:"))
print(L)
```

- Method 2:

```
st = input("Enter integers separated by spaces:")
L=st.split()
for i in range(len(L)):
  L[i] = int(L[i])
print(L)
```

4.2.3 The input reverse problem

```
st = input("Enter integers separated by spaces: ")
L = st.split()
for i in range(len(L)):
  L[i] = int(L[i])
n = len(L)
for i in range(n-1,-1,-1):
  print(L[i],end=' ')
```

4.3 Element distinctness problem

Given a sequence of integers entered by user, check whether or not they are distinct

```
st = input("Enter integers separated by spaces: ")
L = st.split()
for i in range(len(L)):
  L[i] = int(L[i])
n = len(L)

distinct = True
for i in range(n-1):
  for j in range(i+1,n):
    if L[i] ==L[j]:
      distinct = False
      break
  if not distinct:
    break
if(distinct):
  print("Elements are distinct")
else:
  print("Element not distinct")
```

4.4 List copy

- The assignment operator on lists produces an alias:

```
L2 = Lchanging L, changes L2
```

- to get clone of L, use list.copy() method:

```
L3 =L.copy
```

4.4.1 Reverse List

- WRONG SOLUTION:

```
L = [1,2,15,20,17]
print(L)
n = len(L)
L2 = L
for i in range(n):
  L[i] = L2[n-1-i]
print(L)

```

since L2 is just an alias of L

- We can use :

```
L2 = L.copy() instead of L2 =L
```

Or :

```
L = [1,2,15,20,17]
print(L)
n = len(L)
for i in range (n//2):
  #swap L[i] and L[n-1-i]
  temp = L[i]
  L[i] = L[n-1-i]
  L[n-1-i] = temp
print(L)

```

- Or simply :

```
L.reverse()
```

4.5 Tuples

- Tuples are immutable lists: once initialized cannot be modified, i.e., read-only
- Initialization: instead of brackets [] and commas as in lists, use parenthesis () and commas

4.5.1 Application

Swaping two variables :

```
(x,y) = (y,x)
```

Reverse problem:

```
for i in range(n//2):
  (L[i],L[n-1-i])=(L[n-1-i],L[i])
```

4.5.2 Deep copy

- For a tuple T without mutable objects, only need the assignment operator T2 = T since anyways we cannot change T (there is no copy() method for type tuple).
- For a list containing mutable objects, the assignment operator L2=L creates an alias, and L2 = L.copy() creates a clone whose mutable objects are aliases.
- Deep copy: clone all mutable objects all the way, i.e., recursively.

4.6 Sequences

Sequence is an ordered set of objects, like:

- lists
- tuples
- strings
- ranges

Strings are immutable, like tuples.
The indexing operators [i] can be used to access the i'th character of a string.

4.6.1 Iterating over sequences

L is a sequences

```
for x in L:
  code block to processes x

```

4.7 List comprehension

A concise way of initializing lists:

- creates a list L consisting of the value of expression on x, for all elements x in the sequence.

  ```
  L = [expression(x) for x in sequence]
  ```

- creates a list L consisting of the value of expression on x for all elements x in the sequence satisfying the condition.

  ```
  L = [expression(x) for x in sequence if condition]
  ```

Examples:

```
L = [0 for i in range(5)] ===> [0,0,0,0,0]
L = [i for i in range(5)] ===> [0,1,2,3,4]
L = [i*i for i in range(5)] ===> [0,1,4,9,16]

mixed=[1,2,'a',3,4.0]
L = [x**2 for x in mixed if type(x)==int] ===> [1,4,9]
```

Chapter 5

Functions

5.1 Introduction

A function has a name, input parameters (optional) , return value.
Why functions:

- Abstraction
- Code decomposition
- Code reuse

5.1.1 Defining new function

```
def functionName(formal parameters separated by commas):
  body of function
```

if the function returns a value, the function body contains a return statement:

```
return value
```

and it may contain return statement to stop the execution of the function: Calling the function:

```
functionName(actual parameters separated by commas)
```

5.2 Scope of variables and execution stack

Execution stack:

- At function call, the actual parameters are assigned to the formal parameters
- The return statement stops the execution of the function and assigns the returned value
- Each function defines a new namespace, also called a scope
- Formal parameters and local variables exist only within the scope of the function's definition

Scope of variables, Rules:

- Variables are just names which refer to actual objects
- A variable cannot be used before being defined, i.e., initialized
  Python guesses the type from initialization
- Initializing a variable in a function makes it a local variable in the function's scope

5.3 Functions handling lists,tuples,and strings

Keep in mind that a variable of type list is just name which refers to an object of type list.
Passing a list to a function and returning a list from a function are done by the assignment operator: aliases are passed and returned, which is efficient
Example:

```python
def f(L):
  if len(L) >=1:
    L2 = [L[0],L[0]]
    L[0] = 0
    L3 = L +L2
    return L3
  else:
    return L
A = [1,5,6] # it will change to [0,5,6]
B = f(A)  # [0,5,6,1,1]
```

For strings and tuples they cannot be changed when passed as input arguments to functions, unless the tuples contain mutable objects.

5.3.1 Python garbage collector

Garbage collector: a process which eventually wakes up to clean up objects in memory that are not directly or indirectly accessible by variables. after a function is called,it's local variable will be cleaned since they are not accessible in global scope.

5.4 Miscellaneous

5.4.1 Building your own modules

You can create a .py file that contain functions, and import those function by importing the .py file

```python
import myfunctions # for importing the file myfunctions.py in the same directory
```

5.4.2 Function defined in another function

```python
def f(x):
  def g(y):
    return y+1
  z = g(x)
  return z*z
```

Here we can't call g from outside the function f

5.4.3 Default parameters

You can set default value for your formal parameters,the default value will be in place if the call of function didn't assign a value to it.

```python
def printName(firstName, lastName, reverse = False):
  if reverse:
    print(lastName + ', '+firstName)
  else:
    print(firstName,LastName)
printName("Homer","Simpson")#output: Homer Simpson
printName("Homer","Simpson",True)#output: Simpson,Homer
```

5.4.4 Function without a return value

Assigning a function which doesn't return value to a variable sets the variable's type to the None type

5.5 Higher-order functions

A function is called a higher-order function if:

- it returns another function, or
- it takes another function as an input argument

5.5.1 Functions are objects

In python functions are objects, namely they maybe be:

- passed to other functions
- returned by other functions
- assigned to variables

5.5.2 Function taking another function as input argument

```
def linear(x):
  return x
def square(x):
  return x*x
def findsum(n,p):
  x=0
  for i in range(1,n+1):
    x = x+p(i)
  return x
print(findSum(10,linear)) # output 55
print(findSum(10,square))# output 385
```

5.6 Some methods associated with list and str types

5.6.1 Methods associated with data types

Types in python have methods associated with them.
T is a type and x is an object of type T.
we call those function by the member access operator " . "

- list.copy
- list.reverse
- x.f(input parameters) , f is a member function of the type T

5.6.2 List methods

- list.append(e):
  add object ”e” to the end of the list it’s the same as list = list + [e]
  it has no return value .
  Side effect:
  in the following example we get an infinite loop.

```
L = [1,20,3]
for e in L :
  L.append(e)
print(L)
```

  L.append has an advantage over L = L+[e], which always creates a new list

- List.extend(L): adds the items in L to the end of List
  Has the same effect on L as L = L+L2
  it has no return value.
  Same side effect as L.append
- L.count(e) returns the number of times that e occurs in L.
- L.insert(i,e) inserts the object e into L at index i.
- L.remove(e) deletes the first occurrence of e from L.
- L.index(e) returns the index of the first occurrence of e in L.
- L.pop(i) removes and returns the item at index i in L.
- L.sort() sorts the elements of L in ascending order

5.6.3 Methods associated with the string type

- str.split : return the list of words for str
  st.split(sep) will return the list of substring separated by sep
- List.join: return a string consisting of the strings in List
  sep.join(L) will joint the elements of the list separated by sep

Chapter 6

Files and exceptions

6.1 Files

Unlike lists , files are stored on drives.

6.1.1 Manipulating files in python

General structure

```
nameHandle = open(fileName,mode)
nameHandle.close()#close the file
```

- fileName: string containing the path of the file
- mode: $\begin{cases} \text{r: reading} \\ \text{w: writing} \\ \text{a: appending} \end{cases}$

Reading a file

- In one shot

```
nameHandle = open(fileName,'r')
s = nameHandle.read() # read the whole file into single string s
nameHandle.close()
```

- line by line

```
nameHandle=open(fileName,'r')
for line in nameHandle:
  ...
nameHandle.close()
```

Writing to a file

```
nameHandle = open(fileName, 'w')
nameHandle.write(s)#now the file consists of the string s
#we can use the write method as many times as needed to append additional strings
nameHandle.close()
```

Note that if we open an existing file in the 'w' mode, its content will be disregarded, if we want to write instead of overwriting we must use the mode 'a' instead of 'w'.
Note that if the file doesn't exist it will be created.

6.1.2 Iterable versus subscriptable objects

If we can iterate over object (for something in object ...) then the object is iterable.
when we can use an indexing operator with the object (object[i]) then the object is subscriptable.
The file object is iterable and not subscriptable.

6.2 Exceptions and assertions

Exception are something that does not conform to the norm.
we can handle exceptions.
Common exceptions:

- TypeError (e/"abc")
- IndexError (L=["a,",","b,"] and try to accessL[2])
- ValueError (int("abc"))
- ZeroDevisionError
- FileNotFoundError

6.2.1 Handling exceptions

Try-except statement

Instead of program crashing, Code Block B will execute if an exception is raised in Code Block A. For any error:

```
try:
  Code Block A
except:
  Code Block B
```

Or :

```
try:
  Code Block A
except Error_1:
  Code Block B_1
except Error_k:#will execute only if Error_k was raised in Block A
  Code Block B_k
```

```
except: #will execute if an exception other than all the above was raised in Block
    A
  Code Block B
```

6.2.2 Assertions

The assert statement raises an AssertionError if the boolean expression was False.

```
assert Boolean expression,"error message"
```

Chapter 7

Miscellaneous: plotting, randomness, and Monte Carlo simulation

7.1 Plotting

- Importing the plotting module :

```
import matplotlib.pyplot as plt
```

- Let X and Y be lists of numbers of the same length, representing x and y coordinates.
  To plot Y as a function of X,use

```
plt.plot(X,Y,color)
```

  color is a string taking values such as ”k” for black, ”r” for red, ”b” for blue ...
  The plot function plots the points (X[i],Y[i]), for i = 0 ,...,len(X)-1
  connected by lines of colors color.

- to include labels :

```
plt.xlabel("x label text")
plt.ylabel("y label text")
```

- to include a title, use

```
plt.title("title text")
```

- to show the figure

```
plt.show()
```

- to clear a figure

```
plt.clf()
```

- to plot on a new or existing figure whose index is i, use

```
plt.figure(i)
plt.close(i)#to close figure i
```

- To plot on the same figure multiple graphs with tiled axes, use

```
plt.subplot(m,n,i)
```

  - m: number of rows
  - n: number of columns
  - i: graph index

- When plotting multiple functions on the same figure, it helps to include a legend to label the functions you can add a label

```
plt.plot(X,Y,label="myLabel")
```

  it appears when you invoke :

```
plt.legend()
```

- If the y-values contain very large and very small numbers, use log scale on the y-axis:

```
plt.yscale('log')
```

7.2 Generating random numbers

- Import the numerical python module numpy

```
import numpy.random as rand
```

- To generages a uniformly random number in the real interval [x,y]

```
rand.uniform(x,y)#float
rand.randint(x,y)#int
```

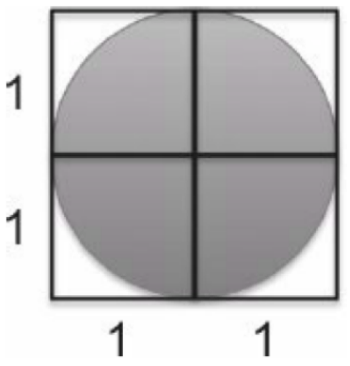
7.3 Monte Carlo Simulation

Monte Carlo simulation is a technique used to approximate the probability of an event by random sampling multiple times, and averaging the results.

7.3.1 Approximating π

- Area of unit circle : $\pi \times 1^2 = \pi$
- Area of unit square : $2 \times 2 = 4$
- $\dfrac{\text{area of unit circle}}{\text{area of unit sqaure}} = \dfrac{\pi}{4}$
- Thus the probability p that a random point of the unit square belongs to the unit circle is $\dfrac{\pi}{4}$

  Which mean : $\boxed{\pi = 4 \times p}$



Now to approximate π:

- Choose n point (x,y) where x and y between -1 and 1
- Find the number m of points in the unit circle
- Return $\dfrac{4m}{n}$
- For large n, get an approximation of π

Chapter 8

Program efficiency,binary Search, insertion sort

8.1 Asymptotic Analysis: Theta notation

8.1.1 Linear search

Consider the linear search

```
c1     def linearSearch(L,e):
c2         n = len(L)
c3         i = 0
c4         while i<n:
c5             if L[i] ==e:
c6                 return i
c7             i = i+1
c8         return -1
```

Let $T(n)$ is the wors case running time of linearSearch on a size-n list
The worst case if e not in L
Thus :
$T(n) = c_1 + c_2 + (c_4 + c_5 + c_7) \times n + c_4 + c_8 = (\text{constant}) \times n + (\text{negligable term compared to } n)$

8.1.2 Asymptotic Analysis

Its a solution to measure the running of algorithm.
we look at the growth of $T(n)$ as the input size $n \to \infty$
The keys are:

- ignore constants
- ignore low order terms

Theta notation

Here it comme the Theta notation as following:

- $5n + 17 \implies \theta(n)$
- $6n^2 + 18n + 5 \implies \theta(n^2)$
- $3\log(n) + 7 \implies \theta(\log(n))$

- $10 \implies \theta 1$

Definition:
Let $f(n)$ and $g(n)$ be function defined on the nonnegative integers.

We say that $f(n) = \theta(g(n))$ if: $\boxed{\lim_{n\to\infty} \frac{f(n)}{g(n)} = a}$ a strictly positive constant.

More generally even if the limit doesn't exist, we say that $f(n) = \theta(g(n))$ if $f(n)$ can be sandwiched between two positive constant multiples of $g(n)$.

$$\boxed{0 \leq c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)}$$

8.1.3 Working with Theta

Useful properties

- $f(n) = \theta(g(n))$ and $g(n) = \theta(h(n)) \implies f(n) = \theta(h(n))$
- $\theta(g(n)) + \theta(g'(n)) = \theta(g(n) + g'(n))$
- $\theta(g(n)) \times \theta(g'(n)) = \theta(g(n) \times g'(n))$

Linear search running time

Back to the linear search running time we can use theta notation as following: $T(n) = \theta(n)$ steps
For the best case running time : $\theta(1)$ (if L[0] ==e)
In case of searching for two elements (two sequential loops), the theta notation as following:
$\theta(n) + \theta(n) = \theta(n)$
Nesting loops costs more

Naive distinct element algorithm

```python
def naiveDistinctElements(L):
  n = len(L)
  for i in range(n):
    for j in range(n):
      if i!=j and L[i] ==L[j]:
        return False
  return True
```

Theta notation : $\theta(1) + n \times (\theta(n)) = \theta(n^2)$ steps

Better Distinct Elements algorithm

```python
def distinctElements(L):
  n = len(L)
  for i in range(n):
    for j in range(i+1,n):
      if L[i] ==L[j]:
        return False
  return True
```

Number of tests is reduced by half, but it is still quadratic

Square tests

By a naive square test : $\theta(\sqrt{n})$ arithmetic operations
By bisection:

```python
def isSquareBisection(n):
  if n<0: return False
  elif n==0:return True
  else:
    low =1
    high =n
    while low <=high:
      mid = (low+high)//2
      if mid*mid ==n :
        return True
      elif mid*mid<n:
        low = mid +1
      else:
        high = mid-1
    return False
```

it take $\theta(log(n))$ arithmetic operations.
because after each iteration, the lenght of the search interval is reduced by at least half

Imporatant Note

$\theta(n)$ arithmetic operations $\neq \theta(n)$ steps since:
for large n, multiplication operation in factorial algorithm costs is more than $\theta(1)$ step.

8.2 Other asymptotic notations

- Theta : $f(n) = \theta(g(n))$
  $f(n)$ is asymptotically like $g(n)$
- Big O: $f(n) = O(g(n))$
  $f(n)$ is asymptotically like $g(n)$ or weaker than $g(n)$
  There exist $c > 0$ and $n_0 > 0$ such that for all $n > n_0$,$0 \leq f(n) \leq c \times g(n)$
- Little o: $f(n) = o(g(n))$
  $f(n)$ is asymptotically weaker than $g(n)$
  $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

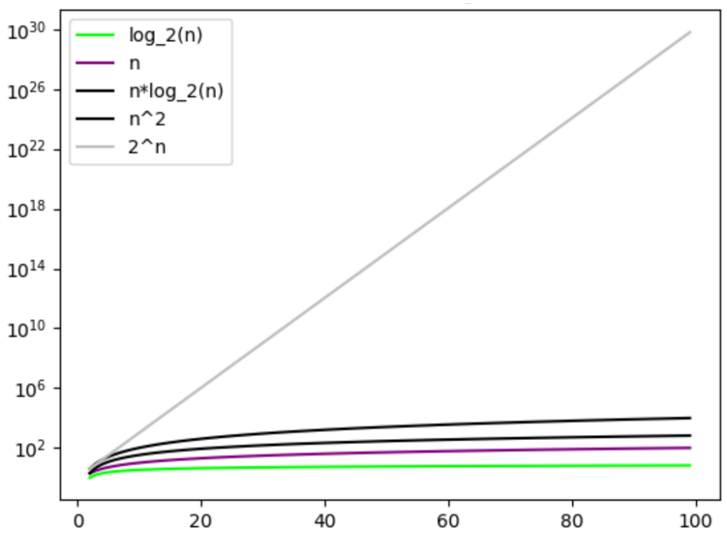
$f(n) = O(g(n))$ and $g(n) = O(f(n)) \longleftrightarrow f(n) = \theta(g(n))$

8.2.1 Worst case running time

- $T(n) = \theta(g(n))$ mean that:
  The worst case running time grows like $g(n)$.
- $T(n) = O(g(n))$ mean that:
  The worst case running time grows like $g(n)$ or is weaker than $g(n)$.
- $T(n) = o(g(n))$ mean that:
  The algorithm is asymptotically much faster than $g(n)$

8.2.2 Common growth rates

- $\theta(1)$ called constant running time.
- $\theta(log(n))$ called logarithmic running time.
- $\theta(n)$ called linear running time.
- $\theta(nlog(n))$ is called log-linear running time.
- $\theta(n^2)$ is called quadratic running time.
- $\theta(n^k), k > 0$ constant, is called polynomial running time.
- $\theta(c^n), c > 1$ constant, is called exponential running time.



8.3 Binary Search

Well linear search takes linear time, we expect algorithm faster than linear search.

8.3.1 The idea

In a sorted list (in non-decreasing order), we want to find if x is in L and return its index if it exist.

- Same as the bisection method
- Compare x with the middle element of L.
- if >, ignore the lower half of L including middle element of L since L is sorted.
- if <,ignore the upper half of L including middle element since L is sorted.
- if =, we are done (x is an element of L).
- Repeat.

```python
def binarySearch(L,x):
  n = len(L)
  low =0
  high = n-1
  while low <=high:
    mid = (low+high)//2
    if L[mid] == x:
      return mid
    elif L[mid]<x:
      low = mid+1
    else:
      high = mid-1
  return -1
```

The worst case running of binary search on a list of size n is $\theta(log(n))$ steps, less than $\theta(n)$.

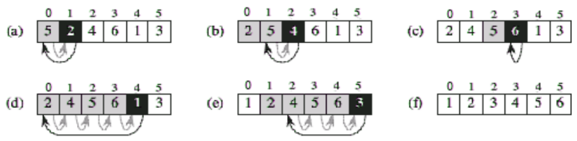
8.4 Sorting problem

The selection sort algorithm take $\theta(n^2)$ time.
Now the insertion Sort, which also takes $\theta(n^2)$time.

8.4.1 Insertion Sort

Idea:

- First element ok.
- Compare the second element with the first and insert in the correct place.
- Compare the third element with the second,and if needed with the first to insert it in the correct place.
- And so till the end



The code:

```python
def insertionSort(L):
  n = len(L)
  for j in range(1,n):
    key = L[j]
    i = j -1
    while i >=0 and L[i] > key:
      L[i+1] = L[i]
      i = i -1
    L[i+1] = key
```

Comparision slection versus insertion sort

	Selection Sort	Insertion Sort
Worst case running time	$\theta(n^2)$	$\theta(n^2)$
Best case running time	$\theta(n^2)$	$\theta(n)$
Number of write operations on list	$\theta(n)$	$\theta(n^2)$ Worst case

8.5 Time analysis of some list operations and methods

8.5.1 Some operations and methods

Equality Check: L1==L2	$\theta(len(L1) + len(L2))$
Concatenation: L = L1+L2	$\theta(len(L) + len(L2))$
Membership: e in L	$\theta(len(L))$
Slicing: L[i:j]	$\theta(j - i)$
L.count(e)	$\theta(len(L))$
L.index(e)	$\theta(len(L))$
L.reverse()	$\theta(len(L))$

8.5.2 List.append

In the worst case $L.append(e)$ operations takes $\theta(len(L))$ time:
if not enough contiguous cells are available, the whole list is copied to new place in memory and resized.
The point is that when copied to a new place in memory, the list is resized to twice its size to allow for efficient append in the next iterations.
That mean copy will only happen when the list length is a power of 2:1,2,4,8,16...
Thus total cost is: $\boxed{\theta(n)}$.
While L = L+[e] method cost:$\theta(n^2)$ be cause for each iteration the cost of $L = L + [e]$ is $\theta(i)$ by creating new list.

8.5.3 list.sort

list.sort takes $\theta(nlog(n))$ time to sort a size-n list, much faster than selection sort and insertion sort,which take $\theta(n^2)$.

Chapter 9

Recursion

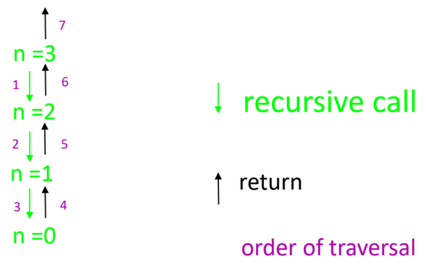
9.1 Introduction: recursive factorial and recursion stack

A recursive definition defines a structure in terms of a smaller version of itself.
To stop the recursion, every recursive definition must have at least one base case.
The general case must eventually reduce the definition into the base case.

$$\text{example: Factorial:recursive definition} \begin{cases} 0! = 1 : n = 0 \text{ is the base case} \\ n! = (n-1)! \times n \text{ if } n \geq 1\text{: the general case} \end{cases}$$

Code:

```python
def recursiveFactorial(n):
    if n == 0: return 1
    else:
        y = recursiveFactorial(n-1)
        return n*y
print(recursiveFactorial(3))
```



9.1.1 Recursion stack

The recursion stack is the execution stack when recursion is involved
Each recursive call to a recursive function has its own:

- parameters
- local variables
- return value
- control (knows where to return when done)

Remember that :
when a function call completes control returns to the calling function.
execution in the calling function resumes from the point immediately following the call.

9.1.2 Iterative Factorial

```python
def iterativeFactorial(n):
    x =1
    for i in range(1,n+1):
        x = x*i
    return x
print("iterativeFactorial(5):",iterativeFactorial(5))
```

9.1.3 Recursive versus iterative factorial

Recursive:
Time : $\theta(n)$ arithmetic operations
Space : $\theta(N)$ integers
iterative implementation is better.

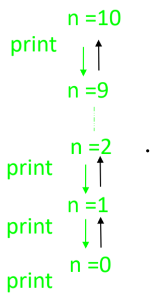
Iterative:
Time : $\theta(n)$ arithmetic operations
Space : $\theta(1)$ integers

9.2 Tracking recursion, indirect recursion, infinite recursion

9.2.1 Order of printing

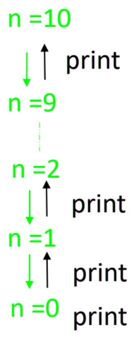
Order 1 :

```python
def f(n):
  print(n,end="")
  if n >=1:
     f(n-1)
f(10)
```



Order 2 :

```python
def g(n):
  if n >=1:
     f(n-1)
  print(n,end="")
g(10)
```



9.2.2 Infinite recursion

Infinite recursion: every function call results in a recursive function call.
in theory it executes forever but the computer executes until it runs out of memory.
if not intended, it is usually the result of missing base case, or the problem does not reduce to the base case.

9.2.3 Indirect recursion

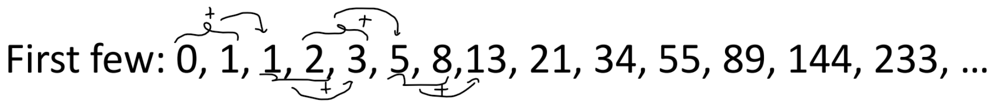
- Direct recursion: a function calls itself
- Indirect recursion: a function f calls other functions that eventually end up calling f

9.3 Two-way recursion: Fibonacci numbers

Two-way recursion: recursive function which calls itself twice, the key concept is recursion tree.

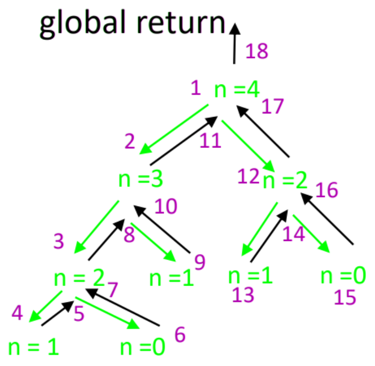
9.3.1 Fibonacci numbers

- Base case:
  - $F_0 = 0$
  - $F_1 = 1$
- General case:
  $F_n = F_{n-1} + F_{n-2}$ if $n \geq 2$


First few: 0, 1, 1, 2, 3, 5, 8,13, 21, 34, 55, 89, 144, 233, …

9.3.2 Fibonacci numbers: recursive function

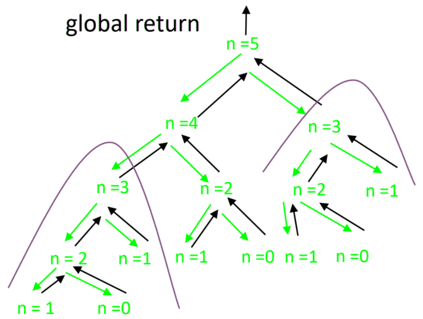
```python
def recFib(n):
    assert type(n) == int and n >=0, "bad input!"
    # the two base cases
    if n ==0 or n==1: return n
    else:
        prev1 = recFib(n-1)
        prev2 = recFib(n-2)
        return prev1 + prev2 # combine the
    results
print(recFib(5))
```



Step:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Stack content:	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
Stack content:		3	3	3	3	3	3	3	3	3		2	2	2	2	2		
Stack content:			2	2	2	2	2		1				1		0			
Stack content:				1		0												

Recursion tree captures the recursion process of multi-way recursion.

- Each node corresponds to a recursive call.
- Root: initial call
- Leaves: base cases
- Down arrows: recursive call
- Up arrow: return



Recursive function not efficient:

- It solves the same subproblem multiple times
- For instance, F_3 is computed twice and F_2 three times

recursive Fibonacci time = $2^{\theta(n)}$ arithmetic operations
recursive Fibonacci space = $\theta(n)$

9.3.3 Fibonacci numbers: non-recursive:list-based

```python
def fibListBased(n):
    assert type(n) == int and n>=0,"Bad input!"
    L = [0]*(n+1)
    if n ==0 or n==1: return n
    else:
        L[0] =0
        L[1] =1
        for i in range(2,n+1):
            L[i] = L[i-1]+L[i-2]
        return L[n]
```

Time $\theta(n)$, space$\theta(n)$

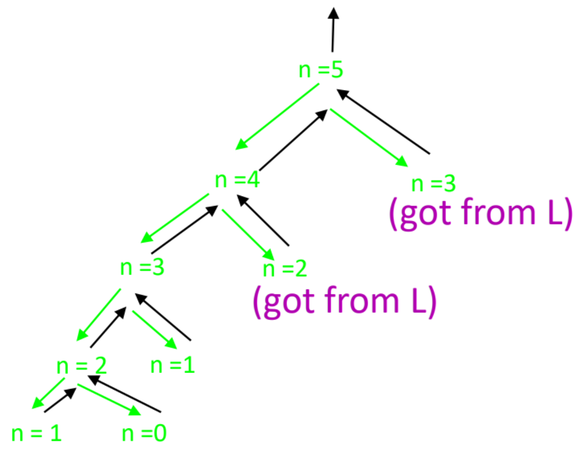
9.3.4 Fibonacci numbers: recursive: memoized

Memoization: to avoid resolving previously solved problems, help recursion with look-up list L[0...n]

- Initially all entries in list are -1
- Before resolving check list
- if -1, solve and save in list
- Else, return value in list

Memoization is an overkill for Fibonacci, but very useful for other problems

```python
def FibMemoized(n):
    assert type(n) ==int and n>=0,"Bad input!"
    def recFibMemoized(n,L):
        if L[n] != -1:
            return L[n]
        elif n==0 or n==1:
            L[n] =n
        else:
            L[n] = recFibMemoized(n-1,L)+recFibMemoized(n-2,L)
        return L[n]
    L = [-1]*(n+1)
    return recFibMemoized(n,L)
```



Time

$\theta(n)$ arithmetic operations , Space: $\theta(n)$ integers

9.3.5 Fibonacci numbers: non-recursive O(1) space

Key: enough to keep track of the last two Fibonacci numbers

```python
def fib(n):
    assert type(n) ==int and n>=0,"Bad input!"
    if n==0 or n==1: return n
    previous2 = 0
    previous1 = 1
    for i in range(2,n+1):
        current = previous1+previous2
        previous2 = previous1
        previous = current
    return current
```

Time $\theta(n)$, space$\theta(1)$

9.3.6 Better than $\theta(n)$ arithmetic operations

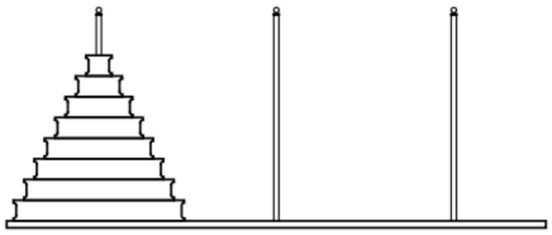
key: matrix identity: for $n \geq 1$

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

9.4 Two-way recursion: tower of Hanoi

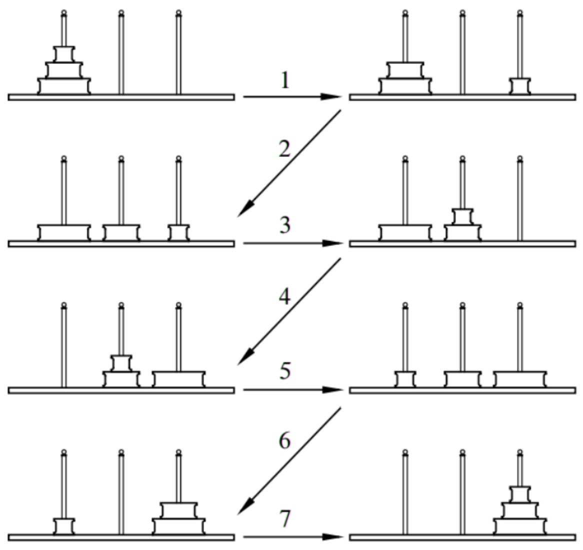
9.4.1 Tower of Hanoi problem

- Given 3 needles and n disks of increasing size.
- The n disks are originally stacked on needle 1 in increasing size with largest at the bottom.
- Target is to move the disks to needle 3 and place in the same same order.
- Constraints:
  - Move one disk at a time from one needle to another
  - A disk can not be placed on top of a smaller disk.



Solution

For n =3



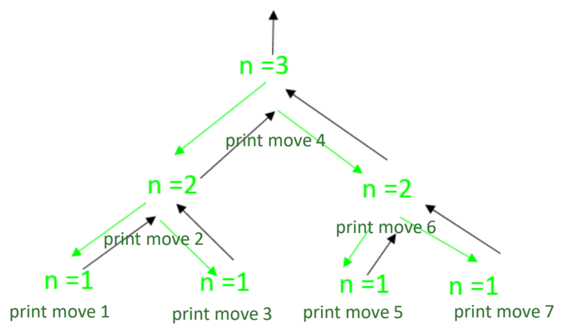
To move n disks from needle 1 to needle 3:

- 1. if $n \geq 2$, move top $n-1$ disks recursively from needle 1 to needle 2, using needle 3 as an intermediate needle.
- 2.Move disk from needle 1 to needle 3
- 3. if $n \geq 2$,move top $n-1$ disks recursively from needle 2 to needle 3, using needle 1 (which is now empty) as an intermediate needle.

Base case : $n = 1$ do second step only.

Code:

```python
def moveDisks(n,start=1,destination=3,
    intermediate=2):
    if n>=2:
        moveDisks(n-1,start,intermediate,
    destination)
    print("Move disk",n,"from",start," to ",
    destination)
    if n >=2:
        moveDisks(n-1,intermediate,destination,
    start)
moveDisks(3)
```



Time : $\theta(2^n)$,sapce $\theta(n)$

9.5 Recursive binary search

9.5.1 Recursive viewpoint

To find the index of x in a sorted list L:

- Compare middle element of L with x
- If =, return index of middle element
- If <, recur on the upper half of L
- If >, recur on the lower half of L
- If recursive call is on empty list, return -1

Base case: two base cases 2 and 5

9.5.2 Implement the recursiveBinarySearch function

Don't pass lower half or upper half as slices to the function as slicing makes copies and the this takes linear time
Instead, pass the indices low and high in addition to L (just an alias) and x, which takes constant time:

```python
def search(L,x):
    def recBinarySearch(L,x,low,high):
        if low>high:return -1
        mid = (low+high)//2
        if L[mid] ==x:
            return mid
        elif L[mid]<x:
            return recBinarySearch(L,x,mid+1,high)
        else:
            return recBinarySearch(L,x,low,mid-1)
    return recBinarySearch(L,x,0,len(L)-1)
```

9.5.3 Iterative versus recursive binary search

Iterative binary search :	Recursive binary search
Time $\theta(log(n))$ steps	Time $\theta(log(n))$ steps
Space $\theta(1)$	Space $\theta(log(n))$

iterative best.

Chapter 10

Data structures digression

Data structures are used to store and manage data.
Dynamic sets are data structures that support certain operations, also called queries, such as search, insert, and delete.

10.1 Lists of lists

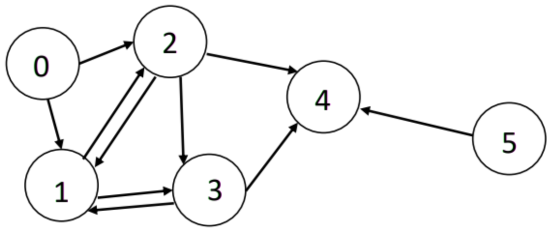
A list L of lists is a list whose entries are objects of type list.

10.1.1 Representing graphs using lists of lists

Adjacency list representation of graph:

Adj[i] is a list containing the nodes adjacent to node i:

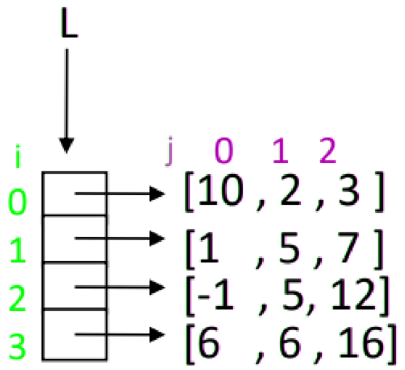
```
Adj = [[1,2],[2,3],[1,3,4],[1,4],[],[4]]
```



Tables , Matrices:

In general, we can model an $m \times n$ table/matrix using a length-m list L,where L[i] is a length-n list, for i=0,...,m-1

- m = number of rows = len(L)
- n = number of columns = row length =len(L[0])
- Thus L[i][j] correspond to cell (i,j) in table



10.1.2 Initializing 2-dimensional list

Given m and n and a scalar val.
an $m \times n$ 2-dimensional list L whose entries are all equal to val can be created:

- using * operator:

```
L = [0]*m
for i in range(m):
  L[i] = [val]*n
```

- Compact method using list comprehension

```
L = [[val for j in range(n)] for i in range(m)]
```

10.1.3 Printing matrices

Printing matrices:

```
import numpy
M = [[10,2,3],[1,-500,7],[1,5,12],[6,6,16]]
print(M)
print(numpy.matrix(M))
_______________________________
OUTPUT:
[[10,2,3],[1,-500,7],[1,5,12],[6,6,16]]
[[10   2   3]
[1   -500   7]
[1   5   12]
[6   6   16]]
```

Printing a boolean matrices

```
M = [[True,False,False],
[False,True,True],
[True,False,True],]
print(numpy.matrix(M))
print(numpy.matrix(M,int))
_______________________________
OUTPUT:
[[True  False  False]
[False  True  True]
[True  False  True]]

[[1  0  0]
[0  1  1]
[1  0  1]]
```

10.2 Applications of 2-dimensional lists

10.2.1 Check if given matrix is symmetric

A matrix M is called symmetric if:

- 1. number n of columns = number m of rows
- 2. M[i][j] = M[j][i]

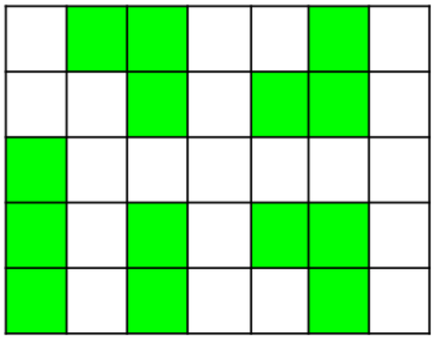
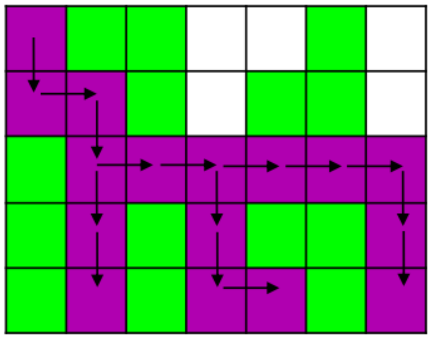
```python
def checkIfSymmetric(M):
    m = len(M)
    n = len(M[0])
    if m!=n: return False
    for i in range(n):
        for j in range(i+1,n):
            if M[i][j] !=M[j][i]
                return False
    return True
```

10.2.2 Maze reachability: right-down moves

$m \times n$ table with True and False entries:

- True indicates an open square
- False indicates a blocked square

Once in an open square, you can possibly move to two squares:
the square on the right if it is open and the square below if open.


.


```python
def mazeReachableCells(M):
  m = len(M)
  n = len(M[0])
  R = [[False for j in range(n)] for i in range(m)]
  R[0][0] = M[0][0]
  for i in range(m):
    for j in range(n):
      if R[i][j]:
        if i<m-1 and M[i+1][j]:
          R[i+1][j] = True
        if j<n-1 and M[i][j+1]:
          R[i][j+1] = True
  return R
```

10.3 Dictionaries

10.3.1 Dictionaries

Dictionary (dict) is a built in type in python.
While lists are indexed by integers, dictionaries are indexed by keys.
Syntax to definite a new dictionary :

```
D = \{ key:value,anotherKey:anotherValue,... \}
```

Use subscript operator to access value given key: D[key] Entries in a dictionary are unordered

10.3.2 Dictionary operations

As a data structure, a dictionary supports the four operations/queries:

- Search for key: key in D: return True or False
- Access value given key (read/write): D[key] In read mode, gives an error "KeyError: key" if key is not in D
- Delete entry given key: del D[key] Gives an error "KeyError: key" if key is not in D
- Add new entry (newKey/val pair): D[newKey] = val It overwrites old value if newKey exists in dictionary

Dictionaries are mutable.
Their values can be mutable objects,but not their keys: keys must be immutable.
All 4 dict operations add, access, search, and delete take O(1) time on the average

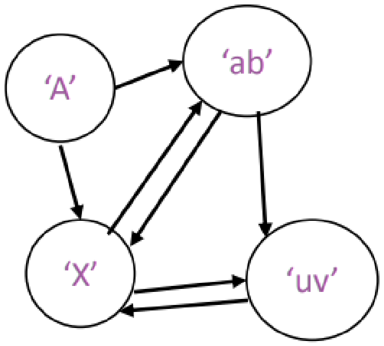
10.3.3 Iterating over keys in dictionaries

```
for key in D:
  #proccess D[key]
```

10.3.4 Copy, clear, update, equality check

- Copy
  - Assignment operator: D2 = D creates an alias D2 of D
  - Copy method dict.copy: D2 = D.copy() : depth-1 only
  - Deep copy: copy.deepcopy(D) returns a deep copy (need to import copy)
- Clear:
  To clear all the entries of a dictionary: D.clear()
  - Not the same as D =
  - If D has an alias, D.clear() also clears the alias but D = doesn't
- Add:
  To add dictionary D2 to dictionary D: D.update(D2)
- equality check:
  To check if two dictionaries D1 and D2 are equal D1 ==D2

10.4 Applications of dictionaries



Using Adj to represent graph:

- key = node label
- value = list of adjacent nodes

Adj[s] is a list containing the nodes adjacent to node s:
Adj = 'A':['X','ab'],'X':['ab','uv'],'ab':['X','uv'],'uv':['X']

10.4.1 Frequency of words in a file

```python
def freqOfWordsInFile(fileName):
    nameHandle = open(fileName, 'r')
    s = nameHandle.read() # read file into a string s
    nameHandle.close()
    L = s.split() # store words in list L
    D = {} # initialize empty dictionary D : keys = words, values= frequencies
    for w in L:  # loop over strings in list L
        if w  not in D: # search for w in D
            D[w] = 1 # if not found: add w as a new word with frequency  1
        else:
            D[w] +=1 # increment frequency of w if found
    return D

D = freqOfWordsInFile("findFreqInFile.py")
for w in D:
    print("Word:",w,"    Frequency:",D[w])
```

10.4.2 2-SUM

```python
def twoSumUsingDictionary(L,t): # O(n) expected time
    D = {}
    for i in range(len(L)):
        if L[i] not in D:
            D[L[i]] = i
    for x in L:
        if t-x in D:
            return (D[x],D[t-x])
    return (-1,-1)
```

10.5 Stacks

We have already seen an example of a stack: the recursion stack In general, a stack S is a data structure, which as a dynamic set supports operations:

- Push object x to stack S
- Pop object from stack S and return it: removed object is the last
- is stack empty

We can use a list S as a stack :

- Push:

```
S.append(x)
```

- Pop :

```
S.pop()
```

- Is stack empty

```
len(S) == 0
```

Chapter 11

Classes and object oriented programming

11.1 Intro

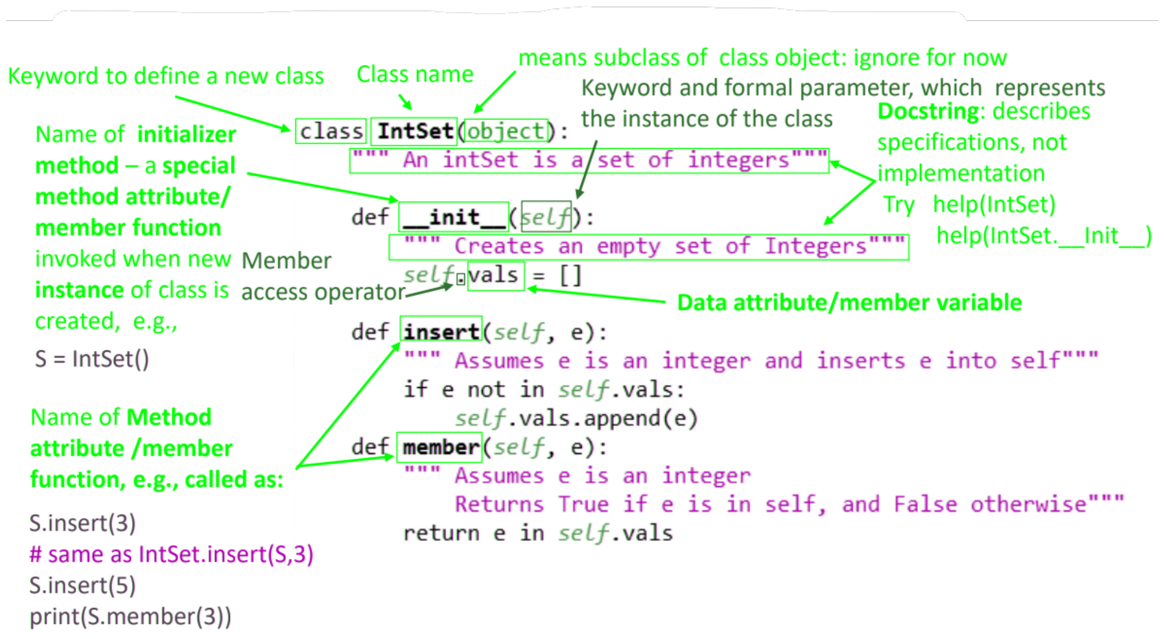
Objects are collections of data and the methods.
Every object has a type, of which the object is an instance.

Abstract data type (ADT) is a type of objects with data and associated methods also called operations.

11.2 User defined classes

Syntax to define a class: class keyword:

```python
class className:
  def method1(..):
    ....
  def method2(..):
    ....
```



```python
class IntSet(object):
    """ An intSet is a set of integers"""
    def __init__(self):
        """ Creates an empty set of Integers"""
        self.vals = []
    def insert(self, e):
        """ Assumes e is an integer and inserts e into self"""
        if e not in self.vals:
            self.vals.append(e)
    def member(self, e):
        """ Assumes e is an integer
            Returns True if e is in self, and False otherwise"""
        return e in self.vals
```

```python
S = IntSet()
S.insert(3)
# same as IntSet.insert(S,3)
S.insert(5)
print(S.member(3))
```

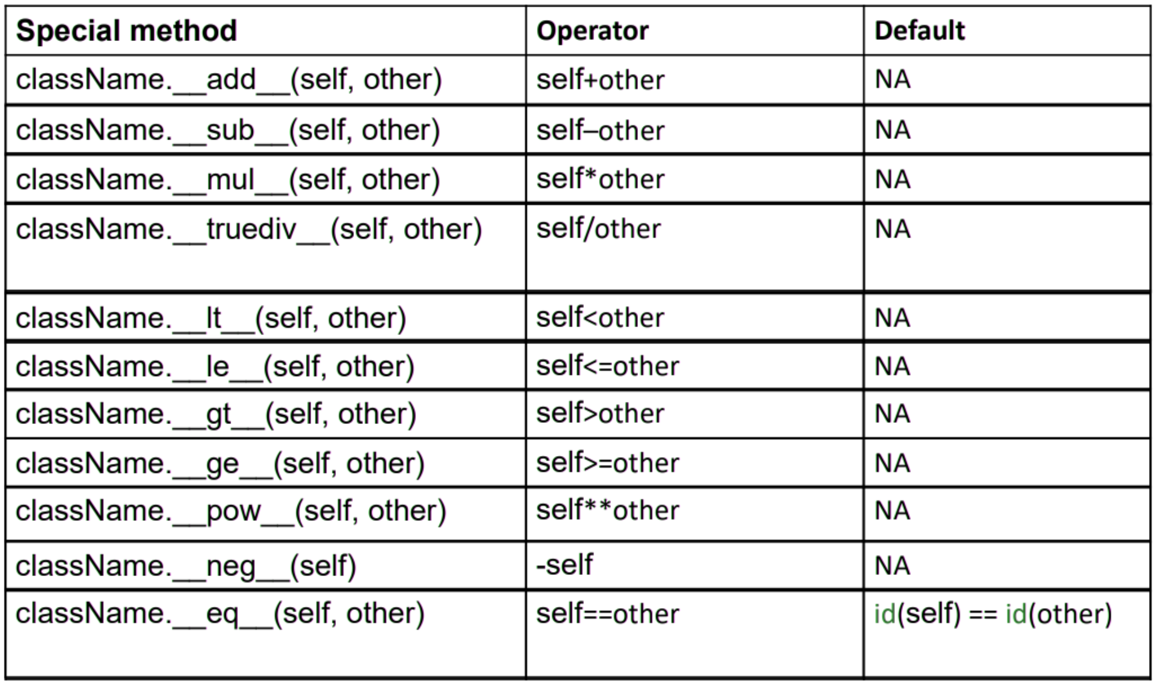
Keyword to define a new class
Class name
means subclass of class object: ignore for now
Keyword and formal parameter, which represents the instance of the class
Docstring: describes specifications, not implementation
Try help(IntSet)
help(IntSet.__Init__)
Name of initializer method – a special method attribute/ member function invoked when new instance of class is created, e.g.,
Member access operator
Data attribute/member variable
Name of Method attribute /member function, e.g., called as:

11.3 OOP machinery

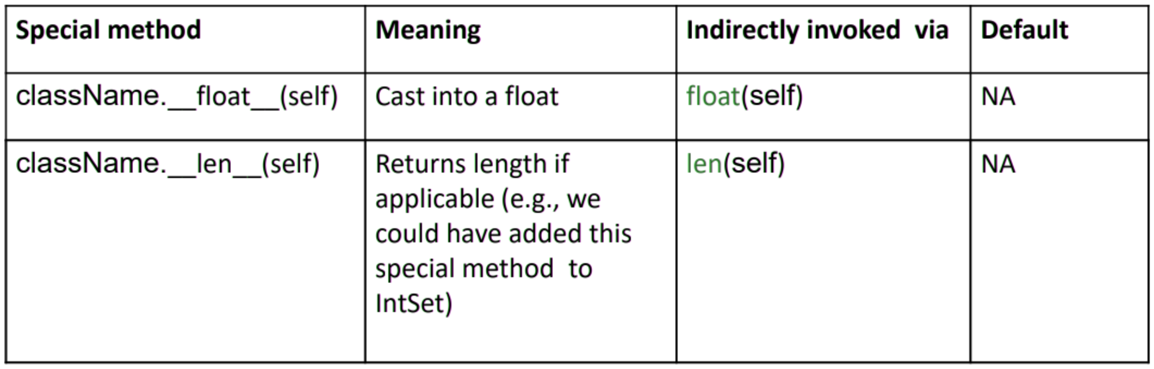
11.3.1 Terminologies

- Dot operator: used to reference attributes
- Attributes/member:
  - data attribute
  - method attribute
- self keyword: if f is a method attribute of a class and S is an instance of the class,calling S.f results in assigning S to self.
- Special methods: (start and end with double underscore):
  - __init__(self,arg1,arg2,...):
    intializer method, invoked when a new instance of the class is created
  - __str__(self):
    string representation method invoked by print and str functions

11.3.2 Other special methods



Special method	Operator	Default
className.__add__(self, other)	self+other	NA
className.__sub__(self, other)	self–other	NA
className.__mul__(self, other)	self*other	NA
className.__truediv__(self, other)	self/other	NA
className.__lt__(self, other)	self<other	NA
className.__le__(self, other)	self<=other	NA
className.__gt__(self, other)	self>other	NA
className.__ge__(self, other)	self>=other	NA
className.__pow__(self, other)	self**other	NA
className.__neg__(self)	-self	NA
className.__eq__(self, other)	self==other	id(self) == id(other)



Special method	Meaning	Indirectly invoked via	Default
className.__float__(self)	Cast into a float	float(self)	NA
className.__len__(self)	Returns length if applicable (e.g., we could have added this special method to IntSet)	len(self)	NA

11.4 OOP concepts

- - Getters: methods which do not modify data attributes.
  - Setters: methods which modify data attributes.
- Encapsulation: It means bundling together data attributes and associated methods attributes
- Information hiding:
  Information hiding allows a programmer to change the implementation of the class (e.g., to improve efficiency) without breaking the code that uses the class based on specifications.

11.5 Inheritance

Inheritance enables a programmer to define a class in terms of another class.
Instead of defining a new class from scratch, a class could be defined as subclass by inheriting certain attributes from a base class.
Inheritance gives a hierarchy of types.

Syntax to define classB as subclass of classA,i.e.,derive classB from classA:

```
class classB(classA):
  def ....
```

The subclass Inherits all attributes (data and methods) of base class.
We can override, i.e., replace, method attributes of the base class in the subclass.
We can include new attributes in the subclass.

11.6 Class variables

Syntax to define a class variable var :

```
class className:
  var = initial value
  def method ...
     ...
```

to access var: use className.var

Chapter 12

Stack class derived from list

12.1 Stack

A stack S is a data structure, which as a dynamic set supports operations:

- Push
- Pop
- is empty?

```
class Stack(list):
    """ class Stack derived from list"""
    def push(self,value):
        self.append(value)
    def top(self):
        assert not self.isEmpty(), "Stack Empty!"
        return self[len(self)-1]
    def isEmpty(self):
        return (len(self)==0)

```

Pop and Push operations take O(1) amortized time

12.2 Queue

Queues are like stacks except that the removed object is the first in instead of last in
A queue Q is a data structure, which as a dynamic set supports operations:

- Enqueue
- Dequeue
- is queue empty

12.2.1 Non-efficient implementation

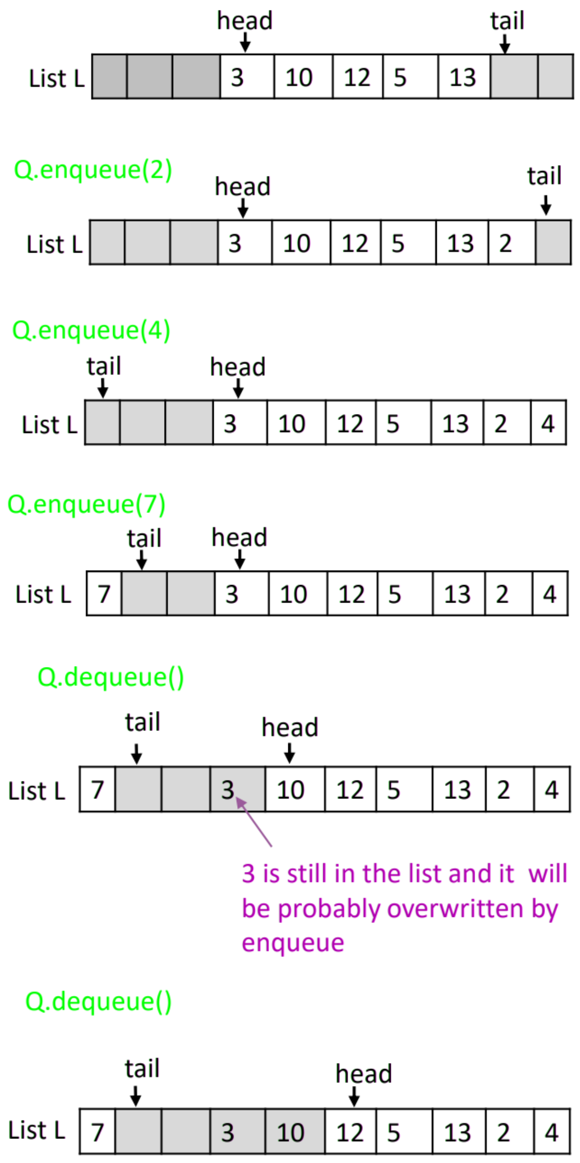
Similarly to a stack, could implement it using a list Q:

- Enqueue: Q.append(x)
- Dequeue: Q.pop(0)
- is empty : len(Q) ==0

The issue is Q.pop(0) takes O(len(Q)) time

12.2.2 Wrap-around

- Use Data attributes:
  - maxSize
  - List L of length maxSize
  - size: number of elements in queue
  - tail: index in L
  - head: index in L
- __init__ method:
  - takes maxSize as input argument and initializes L
- enqueue method:
  - check if full
  - insert element at the tail
  - increment tail in a circular fashion
  - increment size
- dequeue method:
  - check if empty
  - read value at head
  - increment head in a circular fashion
  - return value
- Other methods :
  - peakHead(return value at head)
  - isFull,isEmpty
  - __str__,__len__(return size)



```
class Queue(object):
    """ Queue with given max size """
    def __init__(self, maxSize=10):
        """ takes maxSize whose  default value is 10 """
        self.L = [None]*maxSize
        self.size = 0
        self.maxSize = maxSize
        self.tail = 0
        self.head = 0
    def enqueue(self,value):
        """ adds value to queue, raises exception if full """
        assert not self.isFull(), "Queue Full"
        self.L[self.tail]=value
        if self.tail<self.maxSize-1:
            self.tail+=1
        else:
            self.tail = 0
        self.size+=1
    def dequeue(self):
        """ removes and returns first value in, raises exception if empty"""
        assert not self.isEmpty(), "Queue Empty"
        val = self.L[self.head]
        if self.head<self.maxSize-1:
            self.head+=1
        else:
            self.head=0
        self.size-=1
        return val
    def peakHead(self):
        """ returns value at head, raises exception if empty"""
        assert not self.isEmpty(), "Queue Empty"
        return self.L[self.head]
    def isFull(self):
        """ returns True if full"""
        return self.size==self.maxSize
    def isEmpty(self):
        """ returns True if empty"""
        return self.size==0
    def __str__(self):
        """represent elements in queue as a string:
           separated by commas and between brackets"""
        s = "["
        index  = self.head
        for count in range(self.size):
            s = s+ str(self.L[index])+","
            if index<self.maxSize-1:
                index+=1
            else:
                index=0
        return s[:-1]+"]"
        # -1 to remove the trailing comma
    def __len__(self):
        return self.size
```

Drawbacks and improvement

- One drawback: need to set maxSize at initialization and live with it.
  Solution:
  - Double list size if maxSize reached
- Another drawback which has a similar solution is that too much space remains reserved if queue grows a lot and then shrinks a lot.