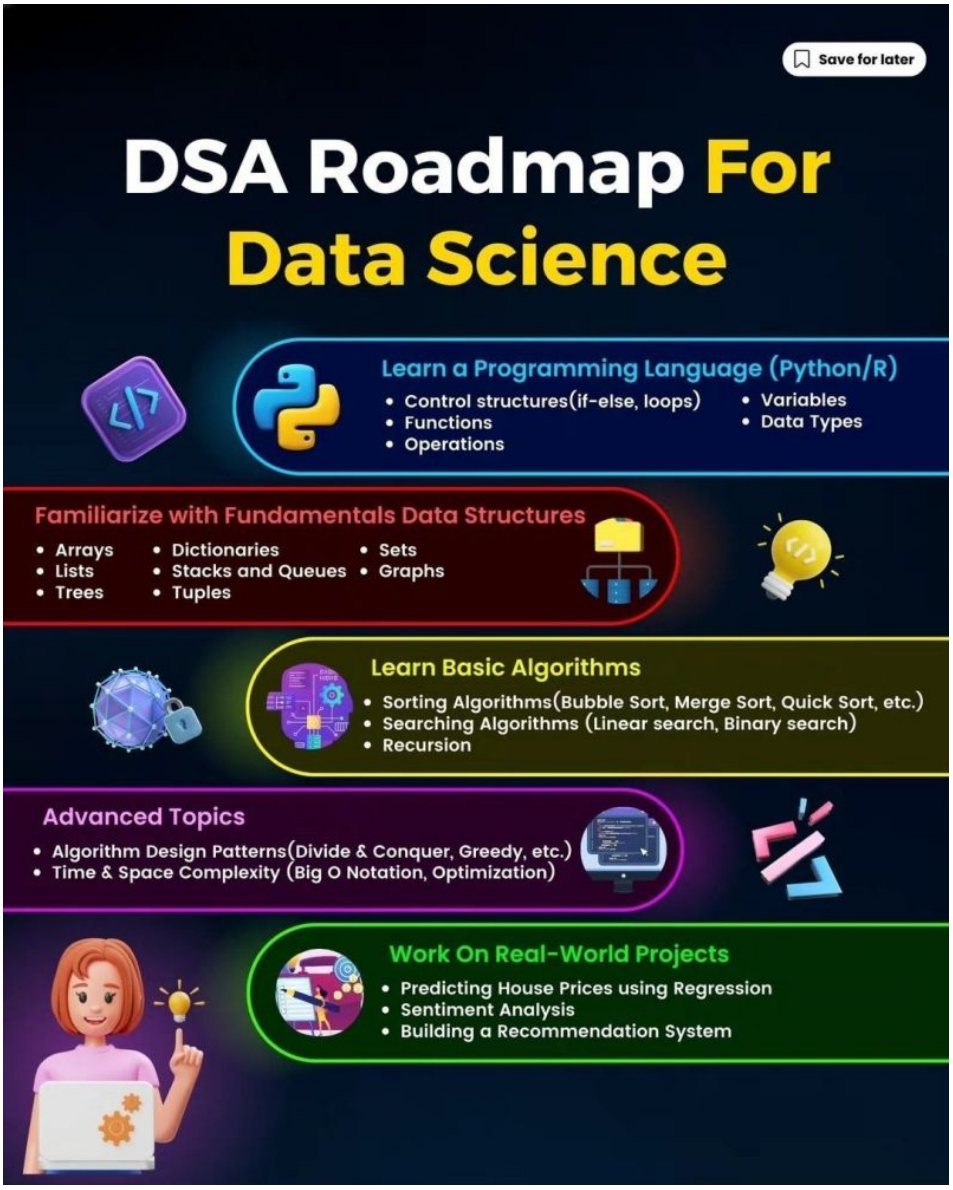
Python - H Bharath Bhat

➡️**Created by H Bharath Bhat**

▼ Contents

Algorithm

1. Algorithm: how to systematically perform a task
2. Programming language describes the step
3. Algorithms that manipulate information:
   - Compute numerical functions - f(x,y)
   - Reorganize data - arrange in ascending order
   - Optimization - find the shortest route



Greatest Common Divisor

gcd(m, n)

- gcd must be the common divisor which divides both the numbers
- for example: gcd(8, 12) = 4
- 1 divides every number

Steps

1. List out factors of m
2. List out factors of n
3. Report the largest number that appears in the list

```
def gcd(m, n):
    m_multiples = []
    n_multiples = []
    for i in range(1, m+1):
```

```
        if m%i == 0:
            m_multiples.append(i)
    for i in range(1, n+1):
        if n%i == 0:
            n_multiples.append(i)

    for i in range (0, len(m_multiples)):
        for j in range(0, len(n_multiples)):
            if m_multiples[i] == n_multiples[j]:
                gcd = m_multiples[i]
    return gcd

print(gcd(8,12))
print(gcd(18,25))


'''
Outputs
4
1
'''
```

- Program is a sequence of steps
- Some steps are repeated
- And some steps are executed conditionally

An algorithm for gcd(m, n)

1. Use fm, fn for list of m, n respectively
2. For each i from 1 to m, add(append) i to fm if i divides m
3. For each j from 1 to n, add(append) j to fm if j divides n
4. Use cf for list of common factors
5. For each f in fm, add f to cf if f also appears in fn
6. Return rightmost value in cf (largest value)

Better code

1. We scan from 1 to m to compute fm and again from 1 to n to compute fn
2. Instead we can scan only once
    a. For each i in 1 to max(m,n) add i to fm if i divides m and add i to fn if i divides n

Best code

1. Compare them to compute common factors cf, at one shot
    a. For each i in 1 to max(m,n), if i divides m and i also divides n, then add i to cf
2. Actually, any common factor must be less than min(m,n)
    a. For each i in 1 to min(m,n), if i divides m and i also divides n, then add i to cf

```
def gcd(m,n):
    multiples = []
    for i in range (1, min(m,n)+1):
        if m%i == 0 and n%i == 0:
            multiples.append(i)
    return max(multiples)

print(gcd(8,12))
print(gcd(18,25))

'''
Outputs
4
1
'''
```

Even more better

1. We do not require any list at all, a value can be stored in a variable

```
def gcd(m,n):
    for i in range (1, min(m,n)+1):
        if m%i == 0 and n%i == 0:
            gcd = i
    return gcd

print(gcd(8,12))
print(gcd(18,25))

'''
Outputs
4
1
'''
```

Better code than before

Can be made even more efficient by scanning backwards instead of from 1

Let i run from min(m,n) to 1

```
def gcd(m,n):
    i = min(m,n)
    while i > 0:
        if m%i == 0 and n%i == 0:
            return i
        else:
            i = i-1
print(gcd(8,12))
print(gcd(18,25))

'''
Outputs
```

```
4
1
'''
```

Euclid’s Algorithm

1. Suppose d divides both m and n, and m>n
2. Then m = a*d, n = b*d
3. So m-n = ad - bd = (a-b)*d
4. d divides m-n as well
5. So gcd(m,n) = gcd(n,m-n)

```
Consider gcd(m,n) with m>n
If n divides m, return n
Otherwise, compute gcd(n, m-n) and return that value
```

```
def gcd(m,n):
        # Assume m>=n [comment]
    if m<n:
        (m,n)=(n,m)
    if m%n == 0:
        return n
    else:
        diff = m-n
        return(gcd(max(n,diff),min(n,diff))) # Recursion
print(gcd(8,12))
print(gcd(18,25))

'''
Outputs
4
1
'''
```

- m-n must be at least 1, if it is 0 then m%n = 0, hence it returns the smaller value i.e n

Euclid’s Algorithm using while condition

```
def gcd(m,n):
    if m<n:
        (m,n)=(n,m)
    while m%n != 0:
        diff = m-n
        (m,n) = (max(n,diff),min(n,diff))
    return n
print(gcd(8,12))
print(gcd(18,25))

'''
Outputs
```

```
4
1
'''
```

Euclid's Extended Version

```
Consider gcd(m,n) with m>n
If n divides m, return n
Otherwise, let r = m%n
Return gcd(n,r)
```

```
def gcd(m,n):
    if m<n:
        (m,n)=(n,m)
    if m%n == 0:
        return n
    else:
        return gcd(n,m%n)  # m%n <n, always
print(gcd(8,12))
print(gcd(18,25))

'''
Outputs
4
1
'''
```

While Condition

```
while condition:
    step 1
    step 2
    step 3
```

1. Don't know in advance how many times we will repeat the steps
2. Should be careful to ensure the loop terminates, eventually the condition should become false

---

Python and its Environment

Compiler

Translates high level programming language to machine level instructions, generates "executable" code.

Interpreter

Itself is a program that runs and directly "understands" high level programming language.

Python is basically an interpreted language

Load the Python interpreter

Send Python commands to the interpreter to be executed

Easy to interactively explore language features

Can load complex programs from files

>>>from filename import *

Running a Python program

Resources

- https://docs.python.org/3/tutorial/index.html
- Dive into Python3, Mark Pilgrim: https://www.diveintopython3.net
- Think Python, 2nd Edition, Allen B. Downey: https://greenteapress.com/wp/think-python-2e/

Typical Python Program

1. Interpreter executes statements from top to bottom
2. Function definitions are “digested” for future use

```
def function01(x,y):
        return x*y
def function02(m,n):
        return m-n
def function03(a,b):
        return a+b

statement_01
statement_02
    :
    :
    :
statement_n
```

3. Actual program starts from statement_01
4. Python allows free mixing of function definitions and statements

Basics of Python

Assignment Statement

- Assign a value to a name (variable)

```
i = 5
j = 2*i
j = j + 5 # statement that updates a value
```

- Left hand side is a name
- Right hand side is an expression
  - Operations in expression depend on type of value

Names, values and types

- Values have types
  - Type determines what operations are legal
- Names inherit their type from their current value
  - Type of a name is not fixed
  - Unlike C/C++/Java where each name is declared in advance with its type.
- Names can be assigned values of different values as the program evolves

```
i = 5 # i is int
i = 7*1 # i is still int
j = i/3 # j is float, creates float
...
i = 2*j # i is now float
```

- type(e) returns type of e

```
i = 5
j = 5.55
print(type(i))
print(type(j))

'''
Outputs
<class 'int'>
<class 'float'>
'''
```

- Not good style to assign values of mixed types to same names

Numeric Values

Numbers come in two flavors:

1. int - integers

   Ex: 178, -87, 0, 76

2. float - fractional numbers

   Ex: 0.1, 3.14, -0.01

- Internally, a value is stored as a finite sequence of 0's and 1's (binary digits, or bits)
- For an int, this sequence is read off as a binary number
- For a float, this sequence breaks up into a mantissa and exponent

- Like “scientific” notation: 0.602 × 10^(24)

Operations on Numbers

- Normal arithmetic operations: +, -, *, /
  - Divide (/) always produces a float
  - Quotient and remainder: // and %
    - 9//5 is 1; 9%5 is 4
- Exponentiation: **
  - 3**4 is 81
- Other operations on Numbers
  - log(), sqrt(), sin(), ...
  - Built on Python, but not available by default
  - Must include math library
    - from math import *

Boolean Values: bool

- True, False
- Logical Operators: not, and, or
  - not True is False, not False is True
  - x and y is True if both of x, y are True
  - x or y is True if at least one of x, y is True
- Frequent ways: Comparisons
  - x == y, a ≠ b, z < 17*5, n>m, 19 ≥ 44*d

```
x = 5
y = 5
z = 8
print(x == y)
print(x == z)
print(x != z)

'''
Outputs
True
False
True
'''
```

- Combine using Logical Operators
  - n >0 and m%n == 0:

```
x = 5
y = 5
z = 8
print(x == y and x == z)
print(x == y or x == z)
print(x == y and x != z)
```

```
print(x == y or x != z)

'''
Outputs
False
True
True
True
'''
```

- Example

```
def divides(m, n):
    if m%n == 0:
        return True
    else:
        return False

def even(n):
    return divides(n,2)

def odd(n):
    return not divides(n,2)

print(divides(12,4))
print(divides(12,5))
print(even(4))
print(even(5))
print(odd(4))
print(odd(5))

'''
Outputs
True
False
True
False
False
True
'''
```

Strings

- Type string, str, a sequence of characters
  - A single character is a string of length 1
  - No separate type char

- Enclose in quotes - single, double, or even triple
- Backslash can also be used to print a single quote inside a string enclosed within a single quote

```
print("Bharath's")
print('Bharath\'s')
print("'Bharath'")
name = '''Bharath is a "gangster's brother"'''
```

```
print(name)

'''
Outputs
Bharath's
Bharath's
'Bharath'
Bharath is a "gangster's brother"
'''
```

- Positions 0, 1, 2, 3,....n-1 for a string of length n
  - Eg: s = ”hello”

0	1	2	3	4
h	e	l	l	o
-5	-4	-3	-2	-1

```
name = "bharath's"
for i in range (0,len(name)):
    print(i, name[i])
print('\n')
for i in range (-len(name), 0):
    print(i, name[i])

'''
Outputs
0 b
1 h
2 a
3 r
4 a
5 t
6 h
7 '
8 s


-9 b
-8 h
-7 a
-6 r
-5 a
-4 t
-3 h
-2 '
-1 s
'''
```

Operations on Strings

- Combine two strings: concatenation, operator +
- Concatenation (+) does not put any spaces in between by default

```
print('Bharath'+'is my name')
print('Bharath '+'is my name')
```

```
print('Bharath','is my name')
print(len("bharath"))

'''
Outputs
Bharathis my name
Bharath is my name
Bharath is my name
7
'''
```

Extracting a Substring

- A slice is a “segment” of a string

```
s = "hello"
print(s[1:4])
print(s[1:5])
print(s[2:])
print(s[:4])
print(s[-5:])
print(s[-4:-1])
print(s[:-1])

'''
Outputs
ell
ello
llo
hell
hello
ell
hell
'''
```

Modifying Strings

- Cannot update a string “in place”
  - s = “hello”, want to change to “help!”
  - s[3] = p - throws error

```
In [51]: s[3]="p"
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[51], line 1
----> 1 s[3]="p"

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

- Instead, use slices and concatenation
  - s = s[0:3] + “p!”
- Strings are immutable values

```
s = s[:3]+"p!"
print(s)

'''
Output
help!
'''
```

Lists

- Sequences of values
  - factors = [1, 2, 3, 4]
  - names = [’bharath’, ‘bhavya’, ‘sharath’]
  - Type need not be uniform
    - mixed = [1, 'bharath', 2, 'sharath’]
  - Extract values by position, slice, like str
    - factors[3] is 4, mixed[0:2] is [2, ‘sharath’]
  - Length is given by len()
    - len(names) is 3
- For str, both a single position and a slice return strings

```
s = "hello"
print(s[0]) # returns a string itself
print(s[0:1]) # returns a string itself

'''
h
h
'''
```

- For lists, a single position returns a value, a slice returns a list

```
mixed = [1, 'bharath', 2, "sharath"]
for i in range(0, len(mixed)):
    print(mixed[i])
print(mixed[1:])
print(mixed[:])

'''
Outputs
1
bharath
2
sharath
['bharath', 2, 'sharath']
[1, 'bharath', 2, 'sharath']
'''
```

Nested List

- Lists can contain other lists too

```
mixed.append([2,3])
print(mixed)
mixed.insert(2, ['hrushik', 'ramya']) # list.insert(index, element)
print(mixed)

'''
Outputs
[1, 'bharath', 2, 'sharath', [2, 3]]
[1, 'bharath', ['hrushik', 'ramya'], 2, 'sharath', [2, 3]]
'''
```

```
for i in range(0, len(mixed)):
    print(mixed[i])

'''
Outputs
1
bharath
['hrushik', 'ramya']
2
sharath
[2, 3]
'''
```

```
print(mixed[2][0])
print(mixed[2][0][1])
print(mixed[2][0][3:])

'''
Outputs
hrushik
r
shik
'''
```

- Unlike strings, lists can be updated in place

```
mixed[2][0] = 'hrasika'
print(mixed)
mixed[2] = ['bengaluru', 'chennai', 'mumbai']
print(mixed)

'''
Outputs
[1, 'bharath', ['hrasika', 'ramya'], 2, 'sharath', [2, 3]]
[1, 'bharath', ['bengaluru', 'chennai', 'mumbai'], 2, 'sharath', [2, 3]]
'''
```

- Lists are mutable, unlike strings

```
sample = [0, 1, 2, 3]
sample[1] = 2
print(sample)

'''
Outputs
[0, 2, 2, 3]
'''
```

List Operations

- We can add elements to a list using append() and extend()
- Remove elements using remove() or pop()
- Sort and reverse lists using sort() and reverse()

```
sample = [3, 1, 4, 1, 5, 9]
sample.append(2)
sample.extend([6, 5])
sample.remove(3) # removes the first occurence in the list
sample.pop()
sample.sort()
sample.reverse()
sample[2] = 'n'
sample.extend([3,4,5]) # list must always be passed in case of extend method
print(sample)

'''
Outputs
sample:                      [3, 1, 4, 1, 5, 9]
sample.append(2):          [3, 1, 4, 1, 5, 9, 2]
sample.extend([6, 5]):   [3, 1, 4, 1, 5, 9, 2, 6, 5]
sample.remove(3):          [1, 4, 1, 5, 9, 2, 6, 5]
sample.pop():                [1, 4, 1, 5, 9, 2, 6]
sample.sort():             [1, 1, 2, 4, 5, 6, 9]
sample.reverse():          [9, 6, 5, 4, 2, 1, 1]
sample[2] = 'n'            [9, 6, 'n', 4, 2, 1, 1]
sample.extend([3,4,5])   [9, 6, 'n', 4, 2, 1, 1, 3, 4, 5]
'''
```

- list1.append(value) - extends the list by a single element passed in the parantheses
- list1.extend([list]) - extends list by a list of values
- remove() removes only the first occurrence in the list
- Safely removes ‘n’ from list - sample

```
if 'n' in sample:
    sample.remove('n')
print(sample)

'''
Output
[9, 6, 4, 2, 1, 1, 3, 4, 5]
'''
```

- Remove all occurrences of element in the list

```
while 4 in sample:
    sample.remove(4)
print(sample)

'''
Output
[9, 6, 2, 1, 1, 3, 5]
'''
```

- **Other Functions**
- index() method returns the index of the leftmost occurrence of the element

```
sample.sort()
print(sample)
print(sample.index(5))

'''
Outputs
[1, 1, 2, 3, 5, 6, 9]
4
'''
```

- What happens when we assign names?

```
x = 5
y = x
x = 7
print("x: ",x)
print("y: ",y)

'''
Outputs
x:  7
y:  5
'''
```

- Does assignment copy the value or make both names point to the same value.
- For immutable value, we can assume that assignment makes a fresh copy of a value.
  - Values apply for all types
- Updating one value does not effect the copy
- For mutable values, assignment does not make a fresh copy.

```
list1 = [1,2,3,4]
list2 = list1
list1[2] = 'n'
print(list1)
print(list2)

'''
Outputs
[1, 2, 'n', 4]
[1, 2, 'n', 4]
'''
```

- What is list2[2] now?
  - list2[2] is also ‘n’
- list1 and list2 are the same names for the same list.

Copying list

- How can we make a copy of a list?
- A slice creates a new (sub) list from the old one

```
l[:k] is l[0:k], l[k:] is l[k:len(l)]
l[:] == l[0:len(l)]
```

- Omitting both end points gives a full slice

```
lista = [5, 6, 7, 8, 9]
listb = lista[:]
lista[2] = 'm'
print(lista)
print(listb)

'''
Outputs
[5, 6, 'm', 8, 9]
[5, 6, 7, 8, 9]
'''
```

- Concatenation produces a new list

```
list1 = [1,3,5,7]
list2 = list1
list1 = list1[0:2] + [7] + list1[3:]
print(list1, list2)

'''
Output
[1, 3, 7, 7] [1, 3, 5, 7]
'''
```

Digression on equality

```
lista = [5, 6, 7, 8, 9]
listb = lista
listc = listb
listc[2] = 'n'
print(lista,listb, listc)

'''
Outputs
[5, 6, 'n', 8, 9] [5, 6, 'n', 8, 9] [5, 6, 'n', 8, 9]
'''
```

```
lista = [5, 6, 7, 8, 9]
listb = [5, 6, 7, 8, 9]
listc = listb
print(lista == listb)
print(lista == listc)
print(listc == listb)
print(lista is listb)
print(lista is listc)
print(listc is listb)
listc[2] = 'n'
print(lista,listb, listc)

'''
Outputs
True
True
True
False
False
True
[5, 6, 7, 8, 9] [5, 6, 'n', 8, 9] [5, 6, 'n', 8, 9]
'''
```

Concatenation of Lists

```
listd = lista + listb
listd + [8]
print(listd)

'''
Outputs
[5, 6, 7, 8, 9, 5, 6, 'n', 8, 9]
'''
```

- Concatenation '+' always produces a new list

```
list1 = [1, 2, 3, 4]
list2 = list1
list1 = list1 + [9]
print(list1, list2)

'''
Outputs
[1, 2, 3, 4, 9] [1, 2, 3, 4]
'''
```

Tuples

- Simultaneous Assignments in rounded brackets

```
(age, name, primes) = (23, "Bharath", [2,3,5])
print(age)
print(name)
print(primes)
```

```
'''
Outputs
23
Bharath
[2, 3, 5]
'''
```

- Can assign a 'tuple' of values to a name

```
point = (3.5, 4.8)
date = (18,4,2024)
print(point)
print(date)

'''
Outputs
(3.5, 4.8)
(18,4,2024)
'''
```

- Extract positions, slices

```
xCoOrdinate = point[0]
monthYear = date[1:]
print(xCoOrdinate, monthYear)
'''
Outputs
3.5 (4, 2024)
'''
```

- **Tuple is immutable,** unlike Lists

```
In [14]: date[1] = 8

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[14], line 1
----> 1 date[1] = 8

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

- l = [13, 46, 0, 25, 72]
- View l as a function associating values to a positions
  - l = {0,1,.., 4} → integers
  - l[0] = 13, l[4] = 72
- 0, 1, 2, .., 4 are keys

Dictionaries

- Allow keys other than range(0, n)
- Key could be a string
  - test1["Dhawan"] = 84
- In python we call it dictionary
  - In other languages it is called **associative array**
  - Any immutable value is known as key

    - Can update dictionaries in place - mutable, like lists
- Empty dictionary is { }
    - Initialization: test1 = { }
    - test1 = [ ] // list
    - test1= ( ) // tuple
- Keys can be immutable values
    - int , float, bool, string, tuple
- Keys cannot be:
    - list, dictionary
- Can nest dictionary

```
score = {}
score["test1"] = {}
score["test2"] = {}
score["test1"]["dhawan"] = 84
score["test2"]["dhawan"] = 27
score["test1"]["kohli"] = 100
print(score)

'''
Outputs
{'test1': {'dhawan': 84, 'kohli': 100}, 'test2': {'dhawan': 27}}
'''
```

Operating on dictionaries

- score.keys() returns sequence of keys of dictionary score

```
for k in score.keys():
    print(k)

'''
Outputs
test1
test2
'''
```

- score.keys() is not in any predictable order

```
score = {}
score["test2"] = {}
score["test1"] = {}
score["test2"]["dhawan"] = 84
score["test1"]["dhawan"] = 27
score["test2"]["kohli"] = 100
print(score)
for k in score.keys():
    print(k)
print("\n")
for k in sorted(score.keys()):
    print(k)
print(list(score.keys()))

'''
```

```
Outputs
{'test2': {'dhawan': 84, 'kohli': 100}, 'test1': {'dhawan': 27}}
test2
test1


test1
test2
['test2', 'test1']
'''
```

- sorted(l) returns sorted copy of l, l.sort() sorts l in place
- score.keys() is not a list - use list(score.keys())
- Similarly, score.values() is sequence of values in score

```
for s in score.values():
    print(s)
'''
Outputs
{'dhawan': 84, 'kohli': 100}
{'dhawan': 27}
'''
```

- Test for key using in, like in membership

/code

Dictionary v/s lists

- Assigning to an unknown key inserts an entry
  - d = { }
  - d[0] = 7 # No problem, d == {0: 7}
- Unlike a list
  - l = []
  - l[0] = 7 # index error

Execution of typical python program

```
def function01(x,y):
        return x*y
def function02(m,n):
        return m-n
def function03(a,b):
        return a+b

statement_01
statement_02
    :
    :
    :
statement_n
```

- Interpreter executes the program from top to bottom
- Function definitions are 'digested' for future use only.
- Actual computation starts from statement_01.

Control Flow

- Need to vary computation steps as values change
- Control flow determines order in which statements are executed
  - Conditional Executions
  - Repeated Executions - loops
  - Functional Definitions

Conditional Execution

If Statement

1. **If statement**

```
if m%n != 0:
    (m, n) = (n, m%n)
```

- Second statement is executed only if the condition m%n ≠ 0 is True
- Indentation demarcates body of if - must be uniform

```
if condition:
    statement_01 # executes conditionally
    statement_02 # executes conditionally
statement_03   # executes unconditionally
```

2. **Alternative Execution**

```
if m%n != 0:
    (m, n) = (n, m%n)
else:
    gcd = n
```

- else is optional

Shortcuts for conditions

- Numeric value 0 is treated as false
- Empty sequence “”, [] is treated as false
- Everything else is True

Multiway Branching, elif

```
if x == 1:
    y = f1(x)
else:
    if x == 2:
        y = f2(x)
    else:
        if x == 3:
            y = f3(x)
```

```
        else:
            y = f4(x)
```

- To avoid this we use else if

```
if x == 1:
    y = f1(x)
elif x == 2:
    y = f2(x)
elif x == 3:
    y = f3(x)
else:
    y = f4(x)
```

Loops: Repeated actions

for loop

- Repeat something a fixed number of times

```
list = [1, 2, 3, 4]
for i in list:
    y = y*i
    z = z+1
```

- Repeating n times:
  - range(0, n) generates sequence 0, 1, 2,....., n-1
  - range(i, j) generates sequence i, i+1, i+2,..., j-1

```
def flist(n):
    flist = []
    for i in range(1, n+1):
        if n%i == 0:
            flist = flist + [i]
    return flist

print(flist(9))

'''
Outputs
[1, 3, 9]
'''
```

- **More about range**
  - range(i, j) produces a sequence from i, i+1, i+2, ..., j-1
  - ranje(j) automatically ranges from 0; 0, 1, 2, ..., j-1
  - range(i, j, k) increments from k; i, i+k, ..., i+nk
    - Stops with n such that i+nk < j ≤ i+(n+1)k
  - Count Down: Make n negative
    - range(i, j, -1), i>j, produces i, i-1, i-2, ..., j+1
  - If k is positive and i≥j, it will generate empty sequence

- If k is negative and i≤j
- If k is negative, stop before j
  - range(12, 1, -3) produces 12, 9, 6, 3
- Compare the following
  - for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
  - for i in range(0, 10):
- Can convert range() into list using list()

```
list(range(0,5))

'''
Output
[0, 1, 2, 3, 4]
'''
```

while conditional loop

- Often we don't know the number of repetitions in advance
- while is executed if the condition evaluates to True
- After each iteration check the condition again
- Repeat based on condition

```
while condition:
    statement_01
    ...
    statement_02
```

- **for** and **while:**
  - primeUptoNo()
    - Know we have to scan from 1 to n, use **for**
  - nprimes()
    - Range to scan not known in advance, use **while**

Break and Continue statements

1. **break statement**

- Exits the loop

```
def findpos(l, v):
    pos, i= -1, 0
    for x in l:
        if x == v:
            pos = i
            break
        i = i+1
    return pos
print(findpos(sample, 5))
print(findpos(sample, 9))
```

```
'''
Outputs
4
6
'''

def findpos(l, v):
    pos = -1
    for i in range(len(l)):
        if l[i] == v:
            pos = i
            break
    # If pos not in the loop, pos is -1
    return pos
# break, if v is found
# terminate loop normally - v is not found
print(findpos(sample, 5))
print(findpos(sample, 9))
print(findpos(sample, 4))

'''
Outputs
4
6
-1
'''
```

- In python, loop can also have else: else signals normal termination (incase of break)

```
def findpos(l, v):
    for i in range(len(l)):
        if l[i] == v:
            pos = i
            break
    else:
        pos = -1 # No break, v not in l
    return pos

print(findpos(sample, 5))
print(findpos(sample, 9))
print(findpos(sample, 4))

'''
Outputs
4
6
-1
'''
```

2. **continue statement**

- Returns back to the start of the loop

```
while True:
    name = input('Who are you: ')
    if name != 'bharath':
        continue
```

```
    password = input('Hello Bharath, Whats the password: ')
    if password == 'bharath28':
        break
print('Access granted')

'''
Output
Who are you: bh
Who are you: bharath
Hello Bharath, Whats the password: bha
Who are you: bharath
Hello Bharath, Whats the password: bharath28
Access granted
'''
```

Function Definition

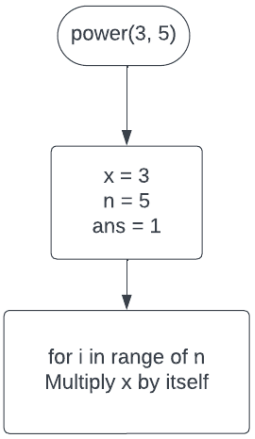
```
def f(a, b, c):
    statement_01
    statement_02
    ...
    return (v)
```

- Function name(arguments/parameters)
- Body is intended
- return() statement exists and returns a value

Passing values to functions

- Argument value is substituted for name

```
def power(x, n):
    ans = 1
    for i in range(0, n):
        ans = ans*x
    return ans
print(power(3, 5))

'''
Outputs
243
'''
```



- Same rules apply for mutable, immutable values
  - Immutable values will not be affected at calling point
  - Mutable values will be affected

```
def update(l, i, v):
    if i >= 0 and i < len(l):
        l[i] = v
        return True
    else:
        return False

ns = [3,11,12]
z = 8
print(update(ns,2,z),ns) # ns is [3, 11, 8]
print(update(ns,4,z),ns) # z remains 8

'''
Outputs
True [3, 11, 8]
False [3, 11, 8]
'''
```

- Return value may be ignored
- If there is no return(), function ends when last statement is reached

Scope of names in a function

- Names within a function have local scope

```
def name(x):
    n = 17
    return x

n = 7
v = name(28)
print(n, v) # Name x inside the function is seperate from x outside

'''
Outputs
```

```
7 28
'''
```

Defining Functions

- A function must be defined before it is invoked
- A python program is executed from top to bottom

Recursive Functions

- A function can call itself - recursion

```
def factorial(n):
    if n<=0:
        return 1
    else:
        val = n*factorial(n-1)
        return val

print(factorial(5))

'''
Outputs
120
'''
```

Passing Arguments by Name

```
def power(x,n):
    ans = 1
    for i in range(0,n):
        ans = ans*x
    return ans
print(power(n=5, x=4)) # 4^5 (4*4*4*4*4)

'''
Outputs
1024
'''
```

Default Arguments

- Recall int(s) that converts string to integer
  - int(”76”) genertaes an 76
  - But, int(”A5”) generates an error
- Actually int(s,b) takes two arguments, string s and base b
  - b has default value 10

```
print(int("A5",16))
print(int("AB5",16))
# Output 165 (10*16^1+5)
```

```
# Output 2741 (10*16^2 + 11*16^1 + 5) → 2560+176+5

print(int('10100',2))
# Output 20 (1*2^4 + 1*2^2) → 16+4
```

```
def f(a, b, c=14, d=22):
    ....
```

- f(13,12) is interpreted as f(13, 12, 14, 22)
- f(13, 12, 16) is interpreted as f(13, 12, 16, 22)
- Default values are identified by position, must come at the end
    - Order of the arguments matter

Function definitions

- def associates a function body with a name
- Flexible, like other value assignments to name
- Definition can be conditional

```
a=9
b=8
if a>b:
    def sub(a,b):
        res = a-b
        return res
else:
    def sub(a,b):
        res = b-a
        return res
print(sub(a,b))

a=8
b=9
if a>b:
    def sub(a,b):
        res = a-b
        return res
else:
    def sub(a,b):
        res = b-a
        return res
print(sub(a,b))


'''
Outputs
1
1
'''
```

- **Can assign a function to a new name**

```
def f(a,b,c):
    ...
```

```
g=f
```

- Now g is another name for f
- Apply f to x n times

```
def apply(f, x, n):
    res = x
    for i in range(n):
        res = f(res)
    return res

def square(x):
    return x*x

apply(square, 5, 2)

# Output: (5^2)^2
```

Examples

- **Prime Finder**: Finds whether the number is prime or not.

```
def primeFinder(n):
    if n == 1:
        return 'non-prime'
    else:
        factorlist = []
        for i in range(1,n+1):
            if n%i == 0:
                factorlist.append(i)
        if len(factorlist) > 2:
            return 'non-prime'
        else:
            return 'prime'

print(primeFinder(9))
print(primeFinder(31))
print(primeFinder(69))
print(primeFinder(11))
print(primeFinder(121))
print(primeFinder(1))

'''
Output
non-prime
prime
non-prime
prime
non-prime
non-prime
'''
```

- **Prime upto n:** List all primes below a given number

```
def primeUptoNo(n):
    primelist = []
    for i in range(1,n+1):
        if primeFinder(i) == 'prime':
            primelist.append(i)
    return primelist

print(primeUptoNo(13))
print(primeUptoNo(8))
print(primeUptoNo(30))

'''
Outputs
[2, 3, 5, 7, 11, 13]
[2, 3, 5, 7]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
'''
```

- **First n prime numbers:** List the first n prime numbers

```
def nprimes(n):
    primelist = []
    count = 0
    i = 1
    while count < n:
        if primeFinder(i) == 'prime':
            count += 1
            primelist.append(i)
        i=i+1
    return primelist

print(nprimes(9),len(nprimes(9)))
print(nprimes(20),len(nprimes(20)))

'''
Outputs
[2, 3, 5, 7, 11, 13, 17, 19, 23] 9
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71] 20
'''
```

Arrays and lists

Arrays

- Single block of memory, elements of uniform type
  - Typically size of sequence is fixed in advance
- Indexing is fast
  - Access seq[i] in constant time for any i.
  - Compute offset from start of memory block
- Inserting between seq[i] and seg[i+1] is expensive

- Contraction is expensive

Lists

- Values scattered in memory
  - Each element points to the next - ‘linked’ list
  - Flexible size
- Follow i links to access seq[i]
  - Cost proportional to i
- Inserting or deleting an element in an element is easy.

Operations

- Exchange seq[i] and seg[j]
  - Constant time in array, linear time in lists
- Delete seq[i] or insert v after seq[i]
  - Constant time in Lists (if we are already at seq[i])
  - Linear time in array - because of shifting
- Algorithms on one data structure may not transfer to another
  - Binary Search

Operating on Lists

- Update an entire list
- map(f, l) applies f to each element of l
- Output of map(f, l) is not a list
  - Use list(map(f, l)) to get a list
  - Use list(map(f, l)) to get a list
  - Can be used directly inside a loop

map function

```
def f(x):
    return x*x

l = [2,3,6,7]

num = map(f, l)
print(list(num))

# Output [4, 9, 36, 49]
```

- Extracting list of primes from list numberlist

```
def isprime(n):
    if n < 2:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
```

```
    return True

numberlist = list(range(0,100))
primelist = []
for i in numberlist:
    if isprime(i):
        primelist.append(i)
print(primelist)

'''
Output
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89,
'''
```

filter function

- filter(p, l) checks p for each element of l
- Output is sublist of values that satisfy p

List Comprehension

- Squares of even numbers below 100

```
def square(x):
    return x*x
def iseven(x):
    if x%2 == 0:
        return True
    else:
        return False
x=2
value=[square(x) for i in range(100) if iseven(x)]
print(value)

# Output: [4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4
```

Binary Search

- Is a value present in a collection seq
- Does the structure of seq matter
    - Array vs list
- Does the organization of the information matter?
    - Values sorted/unsorted

- **The unsorted case**

```
def search(seq, v):
    for i in seq:
        if i == v:
            return True

    return False
```

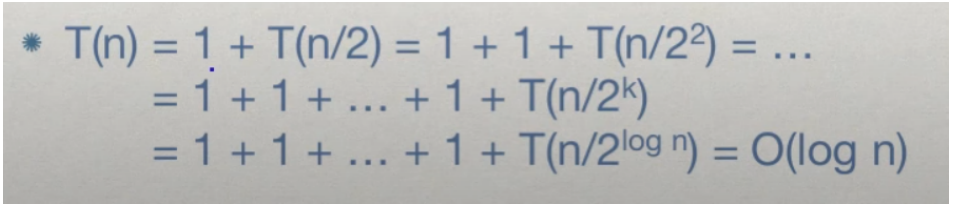
```
listx=[2,1,6,5,7,8,9,4,5,4]
print(listx)
print(search(listx, 9))
print(search(listx, 6))
print(search(listx, 10))

'''
Output
[2, 1, 6, 5, 7, 8, 9, 4, 5, 4]
True
True
False
'''
```

- **Worst Case:** 1. v is not in the list 2. v is the last value in the list
- Need to scan the entire sequence
  - Time proportional to length of the sequence
- Does not matter if seq is array or list

- **The sorted case**
- Compare v with midpoint of seq
- If seq[mid] == v: the value is found
- If v < midpoint, search left half of seq
- If v > midpoint, search right half of seq

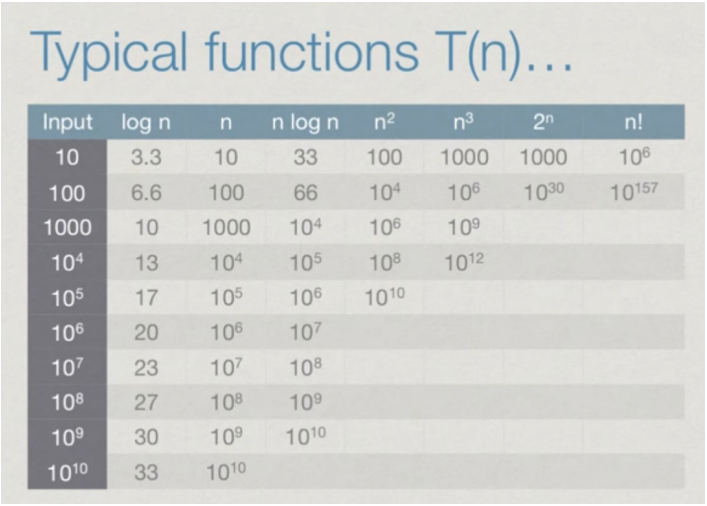
```
def binarySearch(seq, value, l, r):
    seq.sort()
    if r-l == 0:
        return False
    mid = (l+r)//2
    if value == seq[mid]:
        return True
    if value < seq[mid]:
        return binarySearch(seq, value, l, mid)
    else:
        return binarySearch(seq, value, mid+1, r)

listx=[2,1,6,5,7,8,9,4,5,4]
print(listx)
print(binarySearch(listx, 9, 0, len(listx)))
print(binarySearch(listx, 6, 0, len(listx)))
print(binarySearch(listx, 10, 0, len(listx)))

'''
Outputs
[2, 1, 6, 5, 7, 8, 9, 4, 5, 4]
True
True
False
'''
```

- How long will it take
  - Each step halves the interval to search
  - For an interval of size 0, the answer is immediate
- T(n): time to search in a sequence of size n
  - T(0) = 1
  - T(n) = 1+T(n/2); 1 is for finding the mid
  - Unwinding the recurrence



Efficiency

- Usually report worst case behavior
  - When value is not found it is the worst case
  - Worst case is easier to calculate than average case or other more reasonable measures
- Interested in broad relationship between input size and running time
- O() notation
  - Interested in broad relationship between input and running time.
  - Write T(n) = O(n), T(n) = O(n · logn), ... to indicate this
    - Linear scan is O(n) for arrays and lists



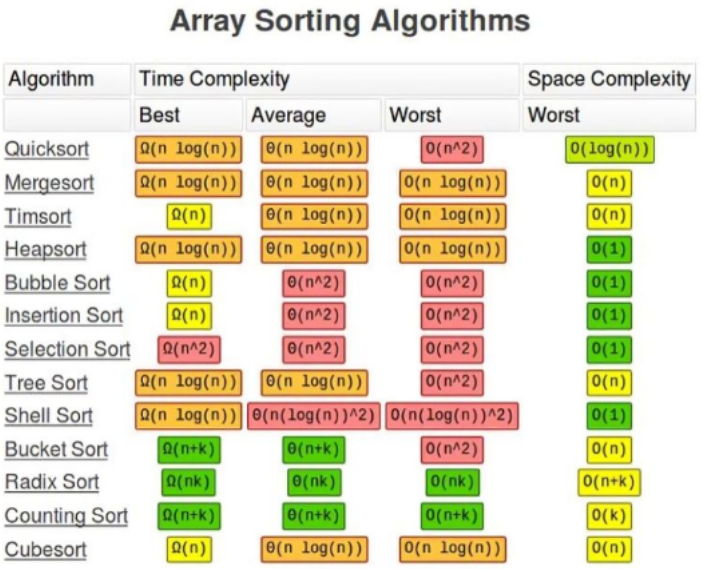
- Python can do about 10^7 basic steps in a second.
- Command in Linux to check time execution of the python program.
  - Reports real, user, sys time.
  - User time must be considered.

```
time python3 filename.py
```

- Theoretically $T(n) = O(n^k)$ is considered efficient
  - Polynomial Time
- In practice even $T(n) = O(n^2)$ has very limited effective range
  - Inputs larger than size 5000 take very long

Sorting

- Searching for a value
  - Unsorted Array - linear scan, O(n)
  - Sorted Array - binary search, O(log n)
- Other advantages of sorting
  - Finding median value: midpoint of sorted list
  - Checking for duplicates
  - Building a frequency table of values



Selection Sort

- Select the next element in sorted order (ascending/descending).
- Move it into its correct place in the final sorted list.

74	32	89	55	21	64

- Old list ↑ | New list ↓

21					
21	32				
21	32	55			
21	32	55	64		
21	32	55	64	74	
21	32	55	64	74	89

- New list will be formed in the above case.

- Avoid using a second list
  - Swap minimum element with value in first position
  - Swap second minimum element to second element.

74	32	89	55	21	64
21	32	89	55	74	64
21	32	89	55	74	64
21	32	55	89	74	64
21	32	55	74	89	64
21	32	55	74	64	89

- Instead of making a new list, we have systematically moved the smallest element to the start of the section we are looking at.

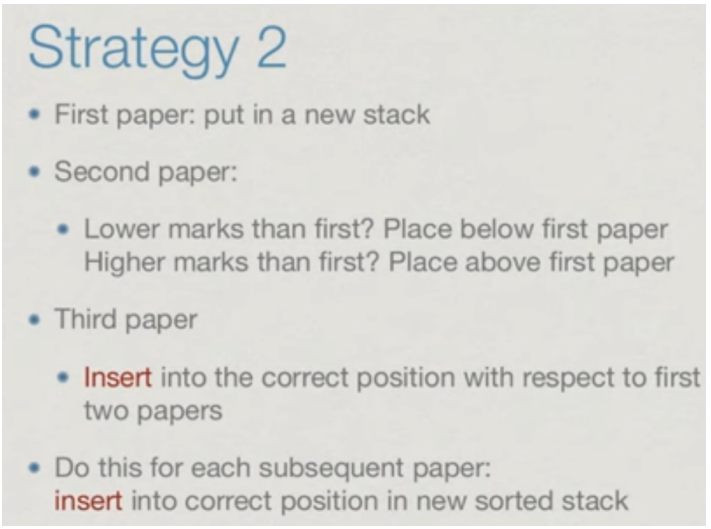
```
def selectionSort(l):
    for start in range(len(l)):
        minpos = start
        for i in range(start, len(l)):
            if l[i] < l[minpos]:
                minpos = i
        (l[start], l[minpos]) = (l[minpos], l[start])
    return l

listx=[2,1,6,5,7,8,9,4,5,4]
print(listx)
print(selectionSort(listx))

'''
Outputs
[2, 1, 6, 5, 7, 8, 9, 4, 5, 4]
[1, 2, 4, 4, 5, 5, 6, 7, 8, 9]
'''
```

Analysis of Selection sort

- Finding minimum in unsorted segment of length k requires one scan, k steps.
- In each iteration, segment to be scanned reduces by 1

- for the first slice + for the second slice of list + ... + for the last slice of the list (1)

Insertion Sort



- Start building a sorted sequence with one element
- Pick up next unsorted element and insert it into its correct place in the already sorted sequence.

74	32	89	55	21	64

74					
32	74				
32	74	89			
32	55	74	89		
21	32	55	74	89	
21	32	55	64	74	89

```
def insertionSort(l):
    for sliceEnd in range(len(l)):
        pos = sliceEnd
        while pos > 0 and l[pos] < l[pos-1]:
            (l[pos], l[pos-1]) = (l[pos-1], l[pos])
            pos = pos -1
    return l
listx=[2,1,6,5,7,8,9,4,5,4]
print(listx)
print(insertionSort(listx))

'''
Outputs
[2, 1, 6, 5, 7, 8, 9, 4, 5, 4]
[1, 2, 4, 4, 5, 5, 6, 7, 8, 9]
'''
```

Analysis of Insertion Sort

- Inserting a new value in sorted segment of length requires up to k steps in the worst case
- In each iteration, sorted segment in which to insert increased by 1

$$\text{D} = O(n^2)$$

Recursive Computation

- Inductive definitions naturally give rise to recursive programs

- **Factorial**

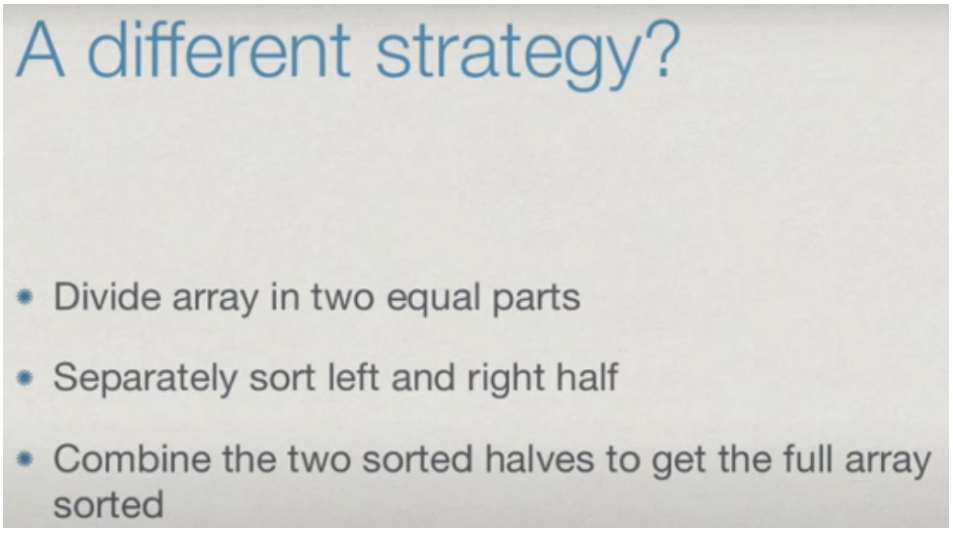
```
def factorial(n):
    if n == 0:
        return 1
    else:
        return(n * factorial(n-1))

print(factorial(5))

'''
Outputs
120
'''
```

Inductive Definitions for lists

- List can be decomposed as
  - First (or Last) element
  - Remaining list with one less element
- Define list functions inductively
  - Base case: empty list or list size 1
  - Inductive step: f(l) in terms of smaller sublists of l

Merge Sort

- Sort A[0 : n//2]
- Sort A[n//2 : n]
- Merge sorted halves into B[0 : n]
- Sorting the halves: Recursively, using same strategy



List

74	32	89	55	21	64

Halved List - 01

Halved List - 02

74	32	89	55	21	64

Sorted - Halved List - 01

32	74	89

Sorted - Halved List - 02

21	55	64

Merge both the sorted list

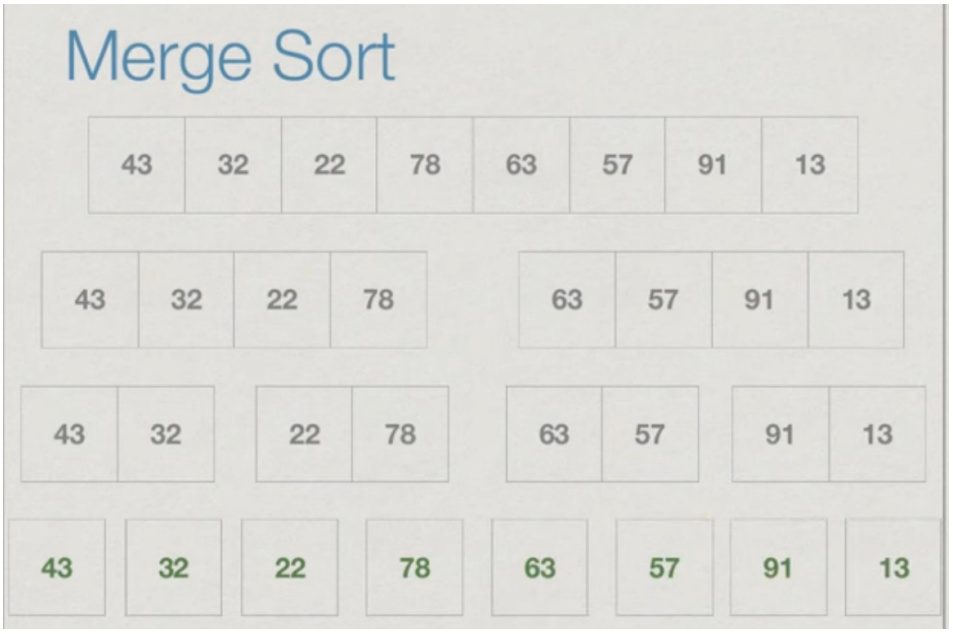
21					
21	32				
21	32	55			
21	32	55	64		
21	32	55	64	74	
21	32	55	64	74	89

Sorting

32	74	89	74	89	89	74	89
21	55	64	64	55	64	64	

74	89		89		

- **Example 02**



Procedure

- Combine two sorted lists A and B into C
  - If A is empty, copy B into C
  - If B is empty, copy A into C
  - Otherwise, compare first element of A and B and move the smaller of the two into C
  - Repeat until all elements in A and B have been moved

```
def merge(A, B):
    (C, m,n) = ([],len(A), len(B))
    (i,j) = (0,0)
    while i+j < m+n:
        if i == m:
            C.append(B[j])
            j = j+1
        elif j == n:
            C.append(A[i])
            i = i+1
        elif A[i] <= B[j]:
            C.append(A[i])
            i = i+1
        elif A[i] > B[j]:
            C.append(B[j])
            j = j+1
    return C
a = list(range(0, 50, 2)) # Even numbers from 0 to 50
b = list(range(1, 40, 2)) # Odd numbers from 1 to 40
print(a,b)
print(merge(a,b))
print(len(merge(a,b)))

'''
Outputs
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 4
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25
45
'''
```

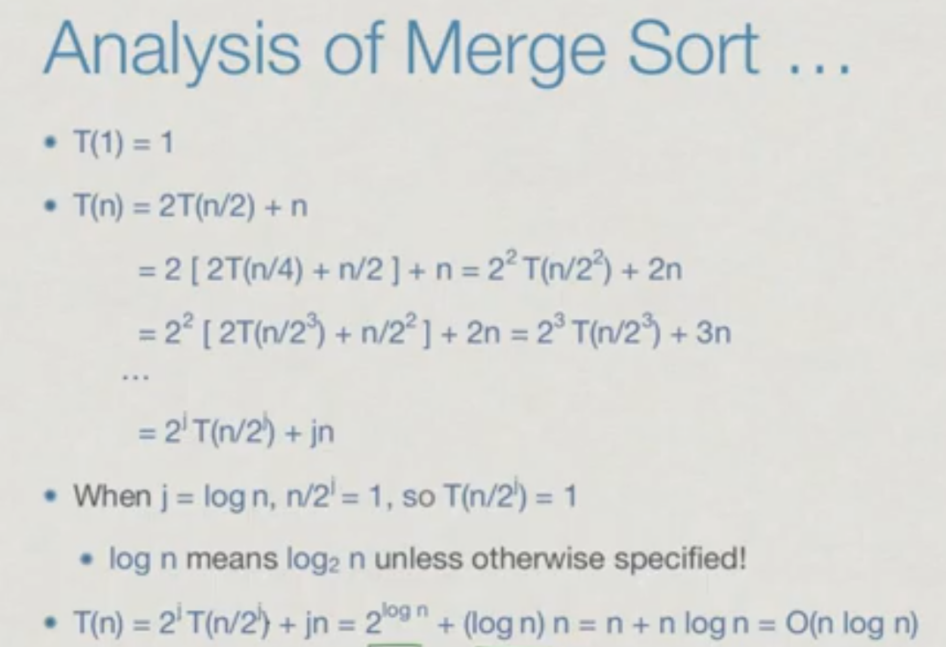
```
def mergeSort(a, left, right):
    if right - left <= 1:
        return a[left : right]
    if right - left > 1:
        mid = (left+right)//2
        l = mergeSort(a, left, mid)
        r = mergeSort(a, mid, right)
    return merge(l,r)
a = list(range(1, 100, 2)) + list(range(0, 100, 2))
print(a)
print(mergeSort(a, 0, len(a)))

'''
Outputs
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 4
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25
''''
```

Analysis of Merge

- Merge A of size m, B of size n into C
- In each iteration, we add one element to C
  - Size of C is m+n
  - $m+n \leq 2 \max(m, n)$
- Hence $O(\max(m, n)) = O(n)$ if $m = n$

Analysis of Merge Sort

- To sort A[0 : n] into B[0 : n]
- If n is 1, nothing to be done
- Otherwise
  - Sort A[0 : n//2] into l (left)
  - Sort A[n : n//2] into l (left)
  - Merge a nd r into B
- T(n): time taken by merge sort on input size n
  - Assume, for simplicity, that $n = 2^k$
- T(n) = 2T(n/2) + n
  - Two subproblems of size n/2
  - Merging solutions requires time $O(n/2 + n/2) = O(n)$
- Solve the recurrence by unwinding



Merge Sort: Shortcomings

- Merging A and B creates a new array C
  - No obvious way to efficiently merge in place
- Extra storage can be costly
- Inherently recursive
  - Recursive call return are expensive

Quick Sort

Alternative approach to Merge Sort

- Extra space is required to merge
- Merging happens because elements in left half must right and vice versa
- Can we divide so that everything to the left is smaller than everything to the right
  - No need to merge

Divide and conquer without merging

- Suppose the median value in A is m
- Move all values ≤ m to left half of A
  - Right half has values > m
  - This shifting can be done in place, in time $O(n)$
- Recursively sort left and right halves
- A is now sorted, no need to merge now

$$T(n) = 2T(n/2) + n = O(n \cdot logn)$$

- Find median: Sort and pick up middle element
- But our aim is to sort
- Instead, pick up some value in A - pivot value
  - Split A with respect to this pivot element

Quick Sort: Algorithm

- Choose a pivot element: typically the first element in the array
- Partition A into lower and upper parts with respect to pivot
- Move pivot between lower and the upper partition
- Sort the partition recursively

43	32	22	78	63	57	91	13

Red: Pivot Element

Yellow: Elements lesser than pivot element, goes to the left of pivot element

Green: Elements greater than pivot element, goes to the right of pivot element

13	32	22	43	63	57	91	78

- Sort the left and right of the pivot element

13	22	32	43	57	63	78	91

Quick Sort: Partitioning

43	32	22	78	63	57	91	13

| ≤43 ↑ >43 ↑

- When you identify 13 as smaller number, swap both 13 and 78

43	32	22	13	63	57	91	78

| ≤43 ↑ >43 ↑

- Swap the pivot element (43) with the last number of the left partition (13)

13	32	22	43	63	57	91	78

| ≤pivot | Pivot | >Pivot |

```
def quickSort(A, l, r):
    if r-l <= 1:
        return()
    yellow = l+1

    for green in range(l+1, r):
```

```
        if A[green] <=  A[l]:
            (A[yellow], A[green]) = (A[green], A[yellow])
            yellow = yellow+1

    (A[l], A[yellow-1]) = (A[yellow-1], A[l])

    quickSort(A, l, yellow-l)
    quickSort(A, yellow, r)
    return A

a = list(range(100, 1, -2)) + list(range(1,100,2))
print(a)
print(quickSort(a, 0, len(a)))

'''
Outputs
[100, 98, 96, 94, 92, 90, 88, 86, 84, 82, 80, 78, 76, 74, 72, 70, 68, 66, 64, 62, 60, 58, 56,
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 2
'''
```

Analysis of Quick Sort

Worst Case

- Pivot is maximum or minimum
  - One of the partition is empty
  - Other has size n-1

$$T(n) = T(n-1) + n = T(n-2) + T(n-1) + 1 = 1 + 2 + ... + n = O(n^2)$$

- Already sorted array is worst case input

But ...

- Average case is $O(n \cdot logn)$
- All permutations of n values, each equally likely
- Average running time across all permutations
- Sorting is rare example where average case can be computed

Quicksort: Randomization

- Worst case arises because of fixed choice of pivot
  - We chose the first element
  - For any fixed strategy (last element, midpoint), can work backwards to construct $O(n^2)$ worst case
- Instead, choose pivot element randomly
  - Pick any index in range(0, n) with uniform probability
- Expected running time is again $O(n \cdot logn)$

Stable Sorting

- Sorting on multiple criteria
- Assume students are listed in alphabetical order
- Now sort students by marks
  - After sorting, are students with equal marks still in alphabetical order?
- Stability is crucial in applications like spreadsheets

    - Sorting column B should not disturb previous sort on column A
- Swap operation during partitioning disturbs the original order
- Merge sort is stable if we merge carefully
    - Do not allow elements from right to overtake elements from left
    - Favour left list when breaking ties

Quicksort in Practice

- Very fast
- Spreadsheets
- Built in sort functions in programming languages

Exception Handling

Common Errors

- y = x/z, but z has value 0
- y = int(s), but string s is not a valid integer
- y = 5*x, but x does not has a value
- y = l[i], but i is not a valid index for list l
- Try to read from a file, but the file does not exist

- Some errors can be anticipated, if anticipated, it can be considered as an exceptional case
- Contingency Plan - exception handling

Exception Handling

- If something goes wrong, provide “corrective action”
- Corrective action depends on the error type
    - File not found: display a message and ask user to retype the filename
    - List index out of bounds - provide diagnostic information to help debug the error
- Need mechanism to internally trap exceptions
- An untapped exception will abort the program

Types of Errors

Syntax Errors

- Most common error, invalid Python code
- SyntaxError: invalid syntax

```
In [5]: l = [5; 2]
          Cell In[5], line 1
            l = [5; 2]
                  ^
        SyntaxError: invalid syntax
```

- Nothing much can be done with these

Run Time Errors

- Error while code is executing (run-time errors)

- Name used before value is defined
  - NameError: name ‘q’ is not defined

```
In [7]: print(q)

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[7], line 1
----> 1 print(q)

NameError: name 'q' is not defined
```

- Division by zero in arithmetic expression
  - ZeroDiviionError: division by zero

```
In [8]: q = 0
        print(3/q)

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[8], line 2
      1 q = 0
----> 2 print(3/q)

ZeroDivisionError: division by zero
```

- Invalid list Index
  - IndexError: list assignment index out of range

```
In [11]: l = [1,2,3,4]

         for i in range(len(l)+1):
             print(l[i])

1
2
3
4

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[11], line 4
      1 l = [1,2,3,4]
      3 for i in range(len(l)+1):
----> 4     print(l[i])

IndexError: list index out of range
```

Terminologies

- Raise an exception
  - Run time error → signal error type, with diagnostic information
    - NameError: name ‘x’ is not defined
  - Handle an exception
    - Anticipate and take corrective action based on error type
- Unhandled exception aborts execution

Handling exceptions

- By using try block: code where error may occur

Zero Division Error

```
def spam(divideby):
    return 42/divideby
try:
    print(spam(9))
    print(spam(6))
    print(spam(0))
except ZeroDivisionError:
    print("error: invalid argument")

'''
Outputs
4.666666666666667
7.0
error: invalid argument
'''
```

Index Error

- except block: what to do if Index Error occurs

```
def listPrint(l,n):
    sq = []
    for i in range(n):
        sq.append(l[i]*l[i])
    return sq

l = [1,2,3,4]
try:
    print(listPrint(l,4))
    print(listPrint(l,5))
except IndexError:
    print("Index Error is found: ",l)

'''
Outputs
[1, 4, 9, 16]
Index Error is found:  [1, 2, 3, 4]
'''
```

Name Error

```
x = 108
try:
    print(x)
    print(y)
except NameError:
    print("Entered variable is missing (hence pritning x): ",x)
'''
Outputs
108
Entered variable is missing (hence pritning x):  108
'''
```

- Common code to handle multiple errors

```
def spam(divideby):
    return 42/divideby
```

```
x = 7

try:
    print(spam(x))
    print(spam(y))
    print(spam(0))
except (NameError, ZeroDivisionError):
    print("Error found")

'''
Outputs
6.0
Error found
'''

def spam(divideby):
    return 42/divideby

x = 7

try:
    print(spam(x))
    #print(spam(y))
    print(spam(0))
except (NameError, ZeroDivisionError):
    print("Error found")
'''
Outputs
6.0
Error found
'''
```

except: Catch all exceptions

- Execution is sequential, if any error is found, it'll skip the next error and goes to the default statement

else:

- Execute if try terminates normally, no errors

```
def spam(divideby):
    return 42/divideby

x = 7

try:
    print(spam(x))
except:
    print("Error found")
else:
    print("done and dusted")
'''
Outputs
6.0
done and dusted
'''
```

- Add a new entry to this dictionary

Using Exception Handling in a Positive Manner

Traditional Manner

```
scores = {'Dhawan':[3,22], 'kohli':[200,3]}
if 'b' in scores.keys():
    scores['b'].append('s')
else:
    scores['b'] = ['s']
print(scores)

'''
Outputs
{'Dhawan': [3, 22], 'kohli': [200, 3], 'b': ['s']}
'''
```

Using exceptions

```
scores = {'Dhawan':[3,22], 'kohli':[200,3]}
try:
    scores['b'].append('s')
except KeyError:
    scores['b'] = ['s']
print(scores)

'''
Outputs
{'Dhawan': [3, 22], 'kohli': [200, 3], 'b': ['s']}
'''
```

Interacting with the user

Inputs

- Program needs to interact with the user
  - Receive Input
  - Display Output
- Standard input and output
  - Input from keyboard
  - Output to screen

```
userdata = input()
print(userdata)
userdata = input("enter a number: ")
print(type(userdata))

'''
Outputs
78
78
enter a number: 78
```

```
<class 'str'>
'''
```

- Input read by the program is always a string, convert that into an integer

```
userdata = int(input("enter a number: "))
print(type(userdata))

'''
Outputs
enter a number: 78
<class 'int'>
'''
```

- But, when a user inputs any other value except integer, the program generates an error

```
In [47]: userdata = int(input("enter a number: "))
         print(type(userdata))

         enter a number: 178

         ---------------------------------------------------------------------------
         ValueError                                Traceback (most recent call last)
         Cell In[47], line 1
         ----> 1 userdata = int(input("enter a number: "))
               2 print(type(userdata))

         ValueError: invalid literal for int() with base 10: '178'
```

- Exception handling can be used

```
while True:
    try:
        userdata = int(input("enter a number: "))
        print(type(userdata))
    except ValueError:
        print("Not a number try again")
    else:
        print("Break")
        break
'''
Outputs
enter a number: bharath28
Not a number try again
enter a number: 28k
Not a number try again
enter a number: 28
<class 'int'>
Break
'''
```

Outputs

Printing on screen

- Print values of names, separated by spaces

```
a = 'Hi'
b = "bruh"
print(a,b,"lol-string")
```

```
# Output: Hi bruh lol-string /
```

Fine tuning print()

- By default, print() appends a new line character '\n' to whatever it printed
  - Each print() appears a new line
- Specify what to append with argument using end="..."

```
print("Bharath is a legend.")
print("yes, I know")
print("Bharath is a legend.",end=" ")
print("yes, I know")
print("Bharath is a legend.",end=".")
print("yes, I know")
print("Bharath is a legend.",end=" surely ")
print("yes, I know")

'''
Outputs
Bharath is a legend.
yes, I know
Bharath is a legend. yes, I know
Bharath is a legend..yes, I know
Bharath is a legend. surely yes, I know
'''
```

- Items are separated by space by default
- Specify separator with argument sep=""
  - Specifies spaces only mentioned and not by default

```
x = "is"
y = "well-known"
print("bharath",x,"a",y,"legend.")
print("bharath",x,"a ",y," legend.", sep="")

'''
Outputs
bharath is a well-known legend.
bharathisa well-known legend.
'''
```

Formatting Print

- Specify width to align text
- Align text within width - left, right, center
- How many digits before/after decimal point

```
name = 'Bharath'
print(name.rjust(20))
print(name.ljust(20))
print(name.rjust(20,'*'))
print(name.ljust(20,'*'))
print(name.center(20,'*'))
```

```
'''
Outputs
            Bharath
Bharath
************Bharath
Bharath************
******Bharath*******
'''
```

Dealing with Files

- Used to read large volumes of data in files located in disks

Disk Buffers

- Disk data is read/written in large blocks
- “Buffer” is a temporary parking place for disk data
- Memory ↔ Buffer ↔ Disk

- Whenever you need to read/write a data it done from or to the buffer, which reads or writes to the disk.

Reading/writing disk data

- Open a file - create file handle to file on disk
  - Like setting up a buffer for the file
- Read and write operations are to file handle
- Close a file
  - Write out buffer to disk (flush)
  - Disconnect the handle

Opening a file

```
fh = open('profile.txt', 'r')
```

- If file is not found, a new file will be created in the same folder
- First argument to open is file name
  - Can give a full path too
- Second Argument is mode for opening a file

Key Letter	Function	Description
r	read	opens a file for reading only
w	write	creates an empty to write to
a	append	append to an existing file

- Reads entire file into name as a single string

```
contents = fh.read()
print(contents)

'''
```

```
File contains/output
Hey there,
This is Bharath Bhat
'''
```

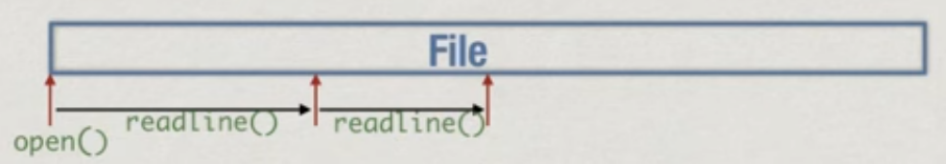
- Reads one line into name - lines end with '\n'
  - String includes the '\n', unlike input()

```
content = fh.readlines()
print(content)

# Output: ['Hey there,\n', 'This is Bharath Bhat']
```

- Reads entire file as list of strings
  - Each string is one line, ending with '\n'

Reading of a file

- Reading is a sequential operation
  - When file is opened, point to position 0, the start
  - Each successive readline() moves forward



- fh.seek(n) - moves pointer to position n
- fh.read(n) - reads a fixed number of characters

```
block = fh.read(12)
print(block)

'''
Output:
Hey there,
T

out of
Hey there,
This is Bharath Bhat
'''
```

End of file:

- When reading incrementally, important to know when file has ended
- The following both signal end of file
  - fh.read() returns empty string " "
  - fh.readline() returns empty string " "

Write to a file

- Write string s to file

- Returns numbers of characters written
- Include ‘\n’ explicitly to go to a new line

```
fh.write(s)
```

```
s = "bharath is a legend"
fh = open('profile.txt','w')
fh.write(s)
# Returns 19

fh = open('profile.txt', 'r')
contents = fh.read()
print(contents)
# Output: bharath is a legend
```

- Write a list of lines l to a file
  - Must include ‘\n’ explicitly for each string

```
s = "bharath is a legendary person \nHe lives in Bengaluru"
fh = open('profile.txt','w')
fh.writelines(s)
fh = open('profile.txt', 'r')
contents = fh.read()
print(contents)

'''
Output
bharath is a legendary person
He lives in Bengaluru
'''
```

Closing a file

```
fh.close()
```

- Flushes output buffer and decouples file handle
  - All pending writes copies to disk

Write a Half Adder Verilog code using python file handling

```
code ='''
module half_adder(input a, b, output sum, carry);

    assign sum = a^b;
    assign carry = a&b;

endmodule
'''
adder = open('half_adder.v','w')
adder.write(code) # 113 characters are added to the file
adder.close()

adder = open('half_adder.v', 'r')
adder_content = adder.read()
```

```
fh.close()

print(adder_content)

'''
Output

module half_adder(input a, b, output sum, carry);

    assign sum = a^b;
    assign carry = a&b;

endmodule
'''
```

- Flushes output buffer and decouples file handle
  - All pending writes copied to disk
  - fh.flush()
- Manually forces write to disk

Processing file line by line

```
adder = open('half_adder.v', 'r')
adder_copy = open('full_adder.v','w')
for line in adder.readlines():
    adder_copy.write(line)

# Above 2 lines can be replaced with the below 2 lines
# contents = adder.readlines()
# adder_copy_01.writelines(contents)

adder.close()
adder_copy.close()

adder_copy = open('full_adder.v','r')
content = adder_copy.read()
print(content)

'''
Outputs

module half_adder(input a, b, output sum, carry);

    assign sum = a^b;
    assign carry = a&b;

endmodule
'''
```

- Lines in the file named ‘adder.v’ was copied to the file named ‘adder_copy.v’

Strip new line character

- Get rid of trailing ‘\n’

```
adder = open('half_adder.v', 'r')
contents = adder.readlines()
for line in contents:
    s = line[:-1]
print(s)
# Output: endmodule
```

- Instead, use rstrip() to remove trailing whitespaces. (lstrip(), strip() can also be used)

```
for line in contents:
    s = line.rstrip()
print(s)
# Output: endmodule
```

String Processing

- String processing functions make it easy to analyze and transform contents
  - Search and replace text
  - Export spreadsheets as text file (csv: comma separated value format) and process columns

Strip Whitespaces

s.rstrip(): removes trailing (right) whitespaces

s.lshift(): removes leading (left) whitespaces

s.strip(): removes whitespaces both the leading and trailing whitespaces of the string

```
name = "    H Bharath Bhat      "
print(name.rstrip())
print(name.lstrip())
print(name.strip())

'''
Outputs
    H Bharath Bhat
H Bharath Bhat
H Bharath Bhat
'''
```

Searching for text

s.find(value)

- Returns first position in s where the given pattern occurs, -1 if no occurrence of the pattern in the string

```
name.find('Bharath') # Output: 7
name.find('Bhat')    # Output: 15
name.find('Bhatt')   # Output: -1
```

s.find(pattern, start, end)

```
name.find('Bha',1,20)  # Output: 7
name.find('Bha',13,20) # Output: 15
```

- Search for pattern in slice s[start : end]
  - s.index(pattern), s.index(pattern, l, r)
- Like find, but raise ValueError if pattern not found

```
s = 'brown fox grey dog brown fox'
print(s.find('brown'))
print(s.find('brown',5,len(s)))
print(s.find('cat'))
print(s.index('brown'))
print(s.index('brown',5,len(s)))

'''
Outputs
0
19
-1
0
19
'''
```

Search and Replace

s.replace(fromstr, tostr)

- Returns copy of s with each occurrence of fromstr replaced by tostr

s.replace(fromstr, tostr, n)

- Replace at most first n copies
- Note that s itself is unchanged - strings are immutable

```
s.replace('brown','black')   # Output: black fox grey dog black fox
s.replace('brown','black',1) # Output: black fox grey dog brown fox
```

Splitting a string

- Export spreadsheet as 'comma separated value' text file
- Want to extract columns from a line of text
- Split the line into chunks between commas
  - Can split any using any separator string

```
columns = s.split(' ') # Output: ['brown', 'fox', 'grey', 'dog', 'brown', 'fox']
```

- Split into at most n chunks

```
columns = s.split(' ',2) # Output: ['brown', 'fox', 'grey dog brown fox']
```

```
csvline = '6,7,8'
print(csvline.split(','))   # Output: ['6', '7', '8']
print(csvline.split(',',1)) # Output: ['6', '7,8']
csvline = '6#?7#?8'
print(csvline.split('#?'))  # Output: ['6', '7', '8']
```

Joining Strings

- Recombine a list of strings using a separator

```
# Example 01
csvline = ','.join(columns) # Output: brown,fox,grey,dog,brown,fox

# Example 02
date = '24'
month = 'may'
year = '2002'

birthdate = '-'.join([date, month, year])
print(birthdate)
# Output: 24-may-2002
```

Converting Case

- Convert uppercase to lowercase and lowercase to uppercase
- s.capitalize() - return a new string with first letter uppercase, rest lower
- s.lower() - converts all uppercase to lowercase
- s.upper() - converts all lowercase to uppercase
- s.title() - converts all the starting letter of the words to uppercase and rest to lowercase
- s.swapcase() - swaps lowercases with uppercases and vise versa

```
name = 'bharath Bhat'
print(name.capitalize()) # Output: Bharath bhat
print(name.lower())      # Output: bharath bhat
print(name.upper())      # Output: BHARATH BHAT
print(name.title())      # Output: Bharath Bhat
print(name.swapcase())   # Output: BHARATH bHAT
```

Resizing Strings

- s.center(n) - returns string of length n with s centered, rest blank
- s.center(n, '*') - fill the rest with * instead of blanks
- s.ljust(n), s.rjust(n), s.rjust(n, ':'), ...
    - Similar, but left/right justify s in returned string

```
print(name.center(50)) #                    bharath Bhat                    |
print(name.center(50,'-')) # --------------------bharath Bhat-------------------
print(name.rjust(40,'→')) # →→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→→bharath Bhat
print(name.ljust(40,'←')) # bharath Bhat←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←←
```

Other Functions

- Check the nature of characters in a string

```
name = 'Bharath Bhat'
password = 'bharath28'

print(name.islower()) # False
print(name.isupper()) # False

print(name.isalpha()) # False because of space
print(password.isalnum()) # True
namee = 'BharathBhat'
print(namee.isalpha()) # True
number = '7892104186' # it must be a string only
print(number.isdecimal()) # True
print(name.istitle()) # True
```

String format() method

Example

```
print('First: {0}, second: {1}'.format(47,11))
print('second: {1}, First: {0}'.format(47,11))

'''
Outputs
First: 47, second: 11
second: 11, First: 47
'''
```

- Replace arguments by position in message string
- Can also replace arguments by name

```
print('one: {f}, two: {s}'.format(f=47,s=11))
print('one: {f}, two: {s}'.format(s=47,f=11))

'''
Outputs
one: 47, two: 11
one: 11, two: 47
'''
```

Formatting

```
print('value:{0:3d}'.format(4)) # value:  4
```

- 3d describes how to display the value 4
- d is a code specifies that 4 should be treated as an integer value
- 3 is the width of the area to show 4

```
print('value:{0:6.2f}'.format(47.523)) # value: 47.52
```

- 6.2f describes how to display the value 47.523
- f is a code specifies that 47.523 should be treated as a floating point value
- 6 - width of the area to show 47.523
- 2 - number of digits to show after decimal point

Doing Nothing in Python

- Blocks such as except:, else:, ... cannot be empty
- Use pass for a null statement

```
while True:
    try:
        userdata = int(input("enter a number: "))
        print(type(userdata))
    except ValueError:
        print("Not a number try again")
    else:
        print("Break")
        break
'''
Outputs
enter a number: bharath28
Not a number try again
enter a number: 28k
Not a number try again
enter a number: 28
<class 'int'>
Break
'''
```

```
while True:
    try:
        userdata = int(input("enter a number: "))
        print(type(userdata))
    except ValueError:
        pass
    else:
        print("Break")
        break
'''
Outputs
enter a number: bh4
enter a number: gh
enter a number: 45
```

```
<class 'int'>
Break
'''
```

- pass can be used to fill the empty cases in if, if-elif, case statements

The value None

- None is a special value used to denote 'nothing'
- Use it to initialize a name and later check if it has been assigned a valid value

Removing a list entry

- Removes the specified index number and automatically contracts the list and shifts elements in l[n+1:] left

```
l = [1,4,3,4,5,6]
del(l[4])
# l = [1, 4, 3, 4, 6]
```

- Note that value at the specified index is removed
- Makes the value undefined

- Also works for dictionaries
- del(d[k]) removes the key k and its associated value

Global Variables

Scope of a Name

- Scope of name is the portion of code where it is available to read and update
- By default, in python, scope is local to functions
  - But actually, only if we update the name inside the function

```
def f():
    y = x
    print(y)

x = 7
f()

# Output: 7
```

```
def f():
    y = x
    print(y)
    x = 22

x = 7
f()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Cell In[2], line 7
      4     x = 22
      6 x = 7
----> 7 f()

Cell In[2], line 2, in f()
      1 def f():
----> 2     y = x
      3     print(y)
      4     x = 22

UnboundLocalError: local variable 'x' referenced before assignment
```

- If x is not found in f(), Python looks at enclosing function for global x
- If x is updated in f(), it becomes a local name

Global Variables

- Actually this applies to immutable values

```
def f():
    y = x[0]
    print(y)
    x[0] = 22

x = [7]
f()

# Output: 7
```

- Global names that point to mutable values can be updated within a function
- Declare a name to be global: refers that x in function refers to the same x in the main function too

Nest Function Definitions

- Can define local 'helper' functions
- g() and h() are only visible to f()

```
def f():
    def g(a):
        return a+1

    def h(b):
        return 2*b

    global x
    y = g(x) + h(x)
    print(y)
    x = 22

x = 7
f()

# Output: 22
```

- If we look up x, y inside g() or h() it will first look in f(), then outside
- Can also declare names global inside g(), h()
- Intermediate scope declaration: nonlocal

Data Structures

- Algorithms + Data Structures → Program Acc to Niklaus Wirth
- Array/lists - sequences of values
- Dictionaries - key-value pairs
- In-built data types

Sets in python

- List with braces, duplicates automatically removed

```
colours = {'red','black','green','red'}
print(colours)
# Output: {'red', 'black', 'green'}
```

- Create an empty set

```
colours = set()
```

- Note, not colours = { } - empty dictionary
- Set membership

```
'black' in colours # True
```

- Convert a list into a set

```
numbers = set([1,2,4,5,1,6,7]) # {1, 2, 4, 5, 6, 7}
print(numbers)
letters = set('banana') # {'b', 'a', 'n'}
print(letters)
```

Set operations

```
odd = set([1,3,5,7,9,11])
prime = set([2,3,5,7,11])

# Union
print(odd | prime) # {1, 2, 3, 5, 7, 9, 11}

# Intersection
print(odd & prime) # {11, 3, 5, 7}

# Set difference
print(odd - prime) # {1, 9}
print(prime - odd) # {2}

# Exclusive or
print(odd ^ prime) # {1, 2, 9}
```

Stacks

- Stack is a last-in, first out list (ex: Chairs)
  - push(s, x) - add x to stack s
  - pop(s) - return most recently added element
- Maintain stack as list, push and pop from the right
  - push(s, x) is s.append(x)
  - s.pop() - python built-in, returns last element
- Stacks are natural to keep track of recursive function calls

Queues

- First-in, first-out sequences
  - add(q, x) - adds x to rear of the queue q
  - remove(q) - removes element at head of q
- Using python lists, left is rear, right is front
  - add(q, x) is q.insert(0, x)
  - l.insert(j, x), insert x before position j
- remove(q) is q.pop()