

# notes

July 1, 2022

These notes are written for Operational Research Course to demonstrate how linear programming problem can be mapped into Python programming language.

## 0.1 Python Introduction

Python is general purpose programming language. It is interpreted and compiled. It is used for web development, data science, scientific computing, and many other fields.

- It is case sensitive.
- It is a dynamic language and do not need a type declaration.
- It is a high level language and can be used to write programs that are easier to read and write.
- Python work on different platforms such as Windows, Linux, Mac, and Raspberry Pi and so on.

```
[47]: print("Hello, World.")  
      print("Second statement", end="\t")  
      print("Third statement")
```

Hello, World.

Second statement

Third statement

**Python Comments** Comments can be added to explain the Python, or to make the code more readable. It is also added to add copyright information, function documentation, and so on.

```
[3]: # This is a comment  
      print("Hello World")  
  
      # Python does not support multiline comments so we can use this way  
      # Another ocmment
```

Hello World

**Python Variable** Variables are containers/places for storing data values.

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits, and underscores.
- Names must begin with a letter
- Names can also begin with underscore(\_)
- Names are case sensitive (y and Y are different variables)

- Python has no command for declaring a variable.
- String variables can be declared either by using single or double quotes

## Data type in Python

- Text Type: `str`
- Numeric Types: `int`, `float`, `complex`
- Sequence Types: `list`, `tuple`, `range`
- Mapping Type: `dict`
- Set Types: `set`, `frozenset`
- Boolean Type: `bool`
- Binary Types: `bytes`, `bytearray`, `memoryview`
- None Type: `None`

```
[18]: name = "AlphaSoftHub" # String
isMultiNational = False # True for true. False for false
NTN = 12345 # Integer, python does not support constants but for human
      ↳ readability we can use that way to define a constant
capital = 1.2345 # Float
none = None # None type.
print(name, type(name), len(name))
print(isMultiNational, type(isMultiNational))
print(NTN, type(NTN))
print(capital, type(capital))
print(none, type(none))
```

```
AlphaSoftHub <class 'str'> 12
False <class 'bool'>
12345 <class 'int'>
1.2345 <class 'float'>
None <class 'NoneType'>
```

### 0.1.1 Python Operators

- Arithmetic operators Arithmetic operators are used to perform arithmetic operations on numbers.
  - + addition
  - - subtraction
  - \* Multiplication
  - / Division
  - % Modulus (Find Reaminder)
  - \*\* Exponentiation
  - // Floor division
- Assignment operators Assignment operators assign a value to a variable.
  - = Assignment
  - += Addition assignment
  - -= Subtraction assignment
  - \*= Multiplication assignment
  - /= Division assignment

- %= Modulus assignment
- \*\*= Exponentiation assignment
- //= Floor division assignment
- &= Bitwise AND assignment
- |= Bitwise OR assignment
- ^= Bitwise XOR assignment
- <<= Left shift assignment
- >>= Right shift assignment
- Comparison operators Comparison operators compare two values and return a Boolean.
  - == Equal to
  - != Not equal to
  - < Less than
  - > Greater than
  - <= Less than or equal to
  - >= Greater than or equal to
- Logical operators Logical operators are used to combine conditional statements.
  - and Logical AND
  - or Logical OR
  - not Logical NOT
- Identity operators Identity operators are used to test if two objects are the same object.
  - is Is (Same as ==)
  - is not Is not (Same as !=)
- Membership operators Membership operators are used to test if a value is present in a sequence.
  - in Membership (Same as ==)
  - not in Not membership (Same as !=)
- Bitwise operators Bitwise operators are used to compare the binary representation of two numbers.
  - & Bitwise AND
  - | Bitwise OR
  - ^ Bitwise XOR
  - ~ Bitwise NOT
  - << Left shift
  - >> Right shift

**Python Sequences** In Python programming, sequences are a generic term for an ordered set which means that the order in which we input the items will be the same when we access them.

- Lists.
- Tuple.
- Sets. ##### List Lists are used to store multiple values of any type in a single variable.
- List items are indexed, the first item has index [0], the second item has index [1] etc. ##### Tuples The only different between list and tuples are they are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.
- Tuples are written with round brackets.
- Tuples are immutable. ##### Sets These are same as tuple but set can not have duplicate values as name suggest.
- A set is a collection which is unordered, unchangeable\*, and unindexed.

Note Set items are unchangeable, but you can remove items and add new items..

```
[25]: names = ['Alice', 'Bob', 'Alex']
data = ['Item 1', 2, 'item 2', False, 2]

data[3] = "False changed to this"
data.append("New item added")

print("List")
print(names, "first element", names[0])
print(data, 'last element', data[3])
print("length", len(names), len(data))
print(type(names))

# Tuples
t = ("item 1", "item 2", 3, 4, 2)

print("Tuples")
print(t, type(t), len(t))
```

List

```
['Alice', 'Bob', 'Alex'] first element Alice
['Item 1', 2, 'item 2', 'False changed to this', 2, 'New item added'] last
element False changed to this
length 3 6
<class 'list'>
```

Tuples

```
('item 1', 'item 2', 3, 4, 2) <class 'tuple'> 5
```

```
[26]: t[2] = 1
```

```
-----
TypeError                                Traceback (most recent call last)
g:\Umer\Projects\python\linear-programming\notes.ipynb Cell 9' in <cell line: 1>()
----> <a href='vscode-notebook-cell:/g%3A/Umer/Projects/python/linear-programming/notes.ipynb#ch0000014?line=0'>1</a> t[2] = 1

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

```
[32]: s = set((1, 2, 3, 4, 5, 4))
print(s, len(s), type(s))
```

```
{1, 2, 3, 4, 5} 5 <class 'set'>
```

### 0.1.2 Python Dictionaries

Dictionaries are used to store data values in key:value pairs, like JSON format.

```
[48]: name = "test"
      value = "test result"
      data = {
          "class": "Operational Research",
          "room": 111,
          "lab": True,
          name: value
      }

      print(data, len(data), type(data), data['class'])
```

```
{'class': 'Operational Research', 'room': 111, 'lab': True, 'test': 'test result'} 4 <class 'dict'> Operational Research
```

```
[34]: users = [
      {
          "name": "Alex",
          "username": "alex012"
      },
      {
          "name": "Bob",
          "username": "bob012",
      }
  ]

  print(users)
```

```
[{'name': 'Alex', 'username': 'alex012'}, {'name': 'Bob', 'username': 'bob012'}]
```

### 0.1.3 Python If-Else

- If else structure are used to condition on values.

Note Python do not have **switch-case** statement.

```
[35]: a = 200
      b = 33
      if b > a:
          print("b is greater than a")
      elif a == b:
          print("a and b are equal")
      else:
          print("a is greater than b")
```

```
a is greater than b
```

### 0.1.4 Python Loops

Python has two primitive loop commands: - **while** loops - **for** loops

- Note Python do not have **do-while** statement.
- Note Python do not have increment and decrement operators.

```
[37]: i = 1
      while i < 6:
          print(i, end="\t")
          i += 1
```

1            2            3            4            5

**For Loopp** A for loop is used for iterating over a **sequence** (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the for keyword in other programming languages, and works more like an **iterator method** as found in other **object-orientated programming languages**. You can find example in Java here [Example](#)

```
[38]: fruits = ["apple", "banana", "cherry"]
      for x in fruits:
          print(x, end="\t")
```

apple    banana    cherry

### 0.1.5 Python Functions

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.
- In Python a function is defined using the `def` keyword

```
[40]: def my_function():
      print("Hello from a function")

      my_function()

      def sum(nums):
          s = 0
          for num in nums:
              s += num
          return s

      print(sum([1, 2, 3, 4]))
```

Hello from a function

10

### 0.1.6 Python Lambda

A lambda function is a small anonymous function. - A lambda function can take any number of arguments, but can only have one expression.

```
[43]: x = lambda a : a + 10
      print(x(5))
```

15

## 0.2 Linear Programming

```
[1]: !pip install scipy
```

Requirement already satisfied: scipy in e:\anaconda\lib\site-packages (1.5.2)  
Requirement already satisfied: numpy>=1.14.5 in e:\anaconda\lib\site-packages  
(from scipy) (1.19.2)

```
[32]: !pip install pulp
```

Collecting pulp  
 Downloading PuLP-2.6.0-py3-none-any.whl (14.2 MB)  
Installing collected packages: pulp  
Successfully installed pulp-2.6.0

```
[6]: import numpy as np
      from scipy.optimize import linprog
      import pulp as p
```

### Using pulp

```
[54]: # Book Page 84. answer 18520.

      # Create a LP Minimization problem
      Lp_prob = p.LpProblem('Problem', p.LpMinimize)

      # Create problem Variables
      x1 = p.LpVariable("x1", lowBound = 0)    # Create a variable x >= 0
      x2 = p.LpVariable("x2", lowBound = 0)    # Create a variable y >= 0

      # Objective Function
      Lp_prob += -10 * x1 - 8 * x2

      # Constraints:
      Lp_prob += 3 * x1 + 1 * x2 <= 4500
      Lp_prob += 2 * x1 + 2 * x2 <= 4000
      Lp_prob += 1 * x1 + 3 * x2 <= 4500

      solution = Lp_prob.solve()    # Solver
```

```
[55]: # Display the problem
      Lp_prob
```

```
[55]: Problem:
MINIMIZE
-10*x1 + -8*x2 + 0
SUBJECT TO
_C1: 3 x1 + x2 <= 4500

_C2: 2 x1 + 2 x2 <= 4000

_C3: x1 + 3 x2 <= 4500

VARIABLES
x1 Continuous
x2 Continuous
```

```
[57]: print(p.value(x1))
print(p.value(x2))
print(p.value(Lp_prob.objective) * -1)
```

```
1250.0
750.0
18500.0
```

### Using scipy

**Parameters** **c** 1-D array The coefficients of the linear objective function to be minimized.

**A\_ub** 2-D array The inequality constraint matrix.

**b\_ub** 1-D array The inequality constraint vector. Right side values.

More at here: <https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linprog.html>

```
[19]: # Book page 84
obj = np.array([-10, -8])
constraints = np.array([
    [3, 1],
    [2, 2],
    [1, 3],
    [-1, 0],
    [0, -1],
])
rightSide = np.array([4500, 4000, 4500, 0, 0])

res = linprog(obj, A_ub=constraints, b_ub=rightSide)
```

```
[19]: -18499.999772467545
```

```
[22]: ans = res.fun * -1
ans
```



[22]: 18499.999772467545

From scratch

```
[8]: from enum import Enum

class Constraints(Enum):
    LESS_THAN = 1
    GRAEATER_THAN = 2
    EQUAL = 3

class Type(Enum):
    MAX = 1
    MIN = 2

class SimplexMethod:
    def __init__(self, noOfVars, noOfConstraint):
        self.noOfVars = noOfVars
        self.noOfConstraint = noOfConstraint
        self.totalConstraints = 1
        self.matrix = np.zeros(
            (noOfConstraint + 1, noOfConstraint + noOfVars + 1))

    def convert(self, eq, constrain_type):
        if constrain_type == Constraints.GRAEATER_THAN:
            eq = [eq * -1 for eq in eq]
            return eq
        elif constrain_type == Constraints.EQUAL:
            return eq
        elif constrain_type == Constraints.LESS_THAN:
            eq = [eq * 1 for eq in eq]
            return eq

    def _add(self, eq, position, eqType="obj"):
        # get position row of matrix
        table = self.matrix[position, :]
        if eqType == "obj":
            eq = [round(float(i), 2) for i in eq.split(',')]
        if len(eq) < self.noOfVars:
            raise Exception("Invalid number of variables")

        for i in range(0, self.noOfVars):
            if eqType == "obj":
                table[i] = eq[i] * -1
            else:
                table[i] = eq[i]
```

```

        # right side value.
        table[-1] = eq[-1]
        self.matrix[position, :] = table

def add_obj(self, eq):
    self._add(eq, 0)

def add_constraints(self, eq, constraint_type):
    eq = [round(float(i), 2) for i in eq.split(',')]
    eq = self.convert(eq, constraint_type)
    self._add(eq, self.totalConstraints, 'constraint')
    self.totalConstraints += 1

def check_negative_topmost_row(self):
    # get the first row of matrix
    table = self.matrix[0, :]
    m = min(table)
    if m < 0:
        return True
    return False

def find_negative_index(self):
    neg_index = np.where(self.matrix == min(self.matrix[0, :]))
    return {
        "col": neg_index[1][0],
        "row": neg_index[0][0]
    } if len(neg_index[1]) > 0 and len(neg_index[0]) > 0 else None

def locate_pivot(self):
    total = np.array([])
    r = self.find_negative_index()
    c = r['col']
    r = r['row']
    row = self.matrix[r, :]
    col = self.matrix[:, c]
    lastCol = self.matrix[:, -1]
    m = min(row)
    for left, right in zip(col, lastCol):
        if left > 0 and right / left > 0:
            total = np.append(total, right / left)
        else:
            total = np.append(total, 0.0)
    # find min value by excluding zero.
    index = np.where(total == min(total[total > 0]))[0][0]
    return [index, c]

def pivot(self, row, col):

```

```

        lc = len(self.matrix[0, :])
        lr = len(self.matrix[:, 0])
        table = np.zeros((lr, lc))
        pivot = self.matrix[row, col]
        pivot_row = self.matrix[row, :]
        if pivot > 0:
            r = pivot_row / pivot
            for i in range(len(self.matrix[:, col])):
                k = self.matrix[i, :]
                c = self.matrix[i, col]
                if list(k) != list(pivot_row):
                    # subtract pivot row from each row and multiply by pivot
                    table[i, :] = list(k - r * c)
            table[row, :] = r
            self.matrix = table

    ↪value

def generate_vars(self):
    v = []
    for i in range(self.noOfVars):
        v.append('x' + str(i+1))
    return v

def solve(self):
    while self.check_negative_topmost_row() == True:
        self.pivot(self.locate_pivot()[0], self.locate_pivot()[1])
        # print(self.matrix)

    i = 0
    var = {}
    for i in range(self.noOfVars):
        col = self.matrix[:, i]
        s = sum(col)
        m = max(col)

        if float(s) == float(m):
            location = np.where(col == m)[0][0]
            var[self.generate_vars()[i]] = self.matrix[location, -1]
        else:
            var[self.generate_vars()[i]] = 0
    var['max'] = self.matrix[0, -1]
    return var

def maximize(self):
    return self.solve()

def minimize(self):

```

```
# multiply first row by -1
self.matrix[0, :] = self.matrix[0, :] * -1
return self.solve()
```

```
[5]: s = SimplexMethod(2, 3)
s.add_obj('10, 8,0')
s.add_constraints('3, 1, 4500', Constraints.LESS_THAN)
s.add_constraints('2, 2, 4000', Constraints.LESS_THAN)
s.add_constraints('1, 3, 4500', Constraints.LESS_THAN)
print(s.maximize())
```

```
{'x1': 1250.0, 'x2': 749.9999999999999, 'max': 18500.0}
```