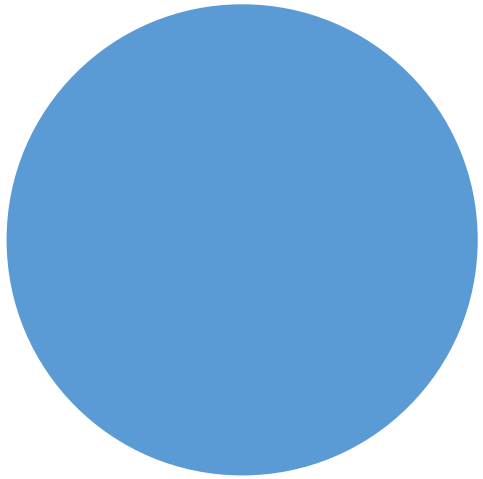
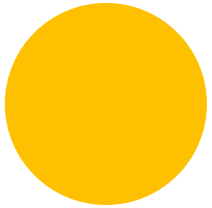



Programming 1

Introduction in programming

Course 2

What we talked about in the last course?

- Course information
  - Class requirements and evaluation
- Basic elements about Python
  - Variables and data types
- Mathematical operations

What we will discuss today?

- Software development process
- Repetitive structures
  - For, while, break, continue
- Predefined data structures
  - List, tuples, dictionary, set

Consider the following scenario …

- You are a computer scientist
- You move to America
- In the morning you listen radio
- Hear the morning news that announce the temperature in Fahrenheit degrees
- PROBLEM! – you cannot convert the Fahrenheit to Celsius degree in order to know how to dress
- SOLUTION! – you think to write a computer program that does the conversion for you

Write a computer program ….

- What information I provide to the computer?
- Which is the formula to convert from Farenheit to Celsius degrees?
- What should the computer respond me?

Write a computer program ….

- What information I provide to the computer?
  - A value representing the temperature in Farenheit degree
- Which is the formula to convert from Farenheit to Celsius degrees?
  - Find the conversion formula C= $(F - 32) * 5/9$ (remark 5/9 evaluates to float in Python 3)
- What should the computer respond me?
  - A value that represents the temperature in Celsius degrees

… the program …

```
tempFarenheit = int ( input ( “Which is the temperature in Farenheit?”))
tempCelsius = (tempFarenheit – 32) * 5/9
print(“Temperature in Celsius degrees is “, tempCelsius)
```

TEST your program

- You should test for some known values if the program gives the expected result

```
Which is the temperature in Farenheit?0
Temperature in Celsius degrees is -17.777777777778

Which is the temperature in Farenheit?100
Temperature in Celsius degrees is 37.777777777778
```

Software Development Process

- Computers must be told what to do right down to the last detail

- Problem solving
  - Broken in stages
  - Each stage
    - Input
    - Output

Software Development Process - Steps

- Analyze the problem
  - Figure out exactly what is the problem that has to be resolved
  - Try to understood as much as possible about the problem
- Determine Specifications (also called Requirements)
- Create a Design
- Implement the Design
- Tests/Debug the program
- Maintain the program

Software Development Process - Steps

- Analyze the problem
- Determine Specifications (also called Requirements)
  - Describe exactly what your program does
  - Do not worry how it will be implemented
  - Clearly identify the available information and what is the expected result
- Create a Design
- Implement the Design
- Tests/Debug the program
- Maintain the program

Software Development Process - Steps

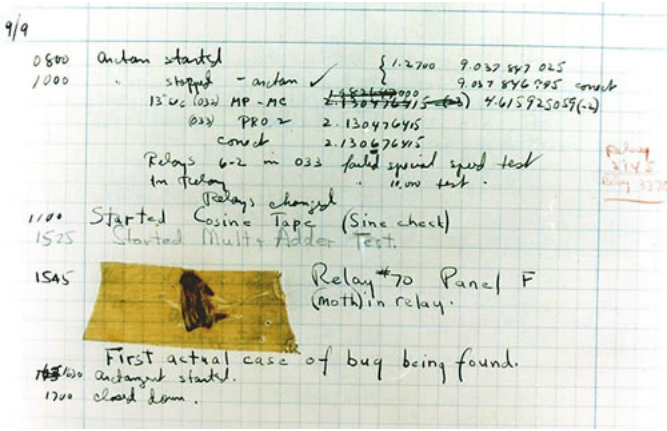
- Analyze the problem
- Determine Specifications (also called Requirements)
- Create a Design
  - Formulate the overall structure of the program
  - Identify and describe the algorithms and data structures
- Implement the Design
- Tests/Debug the program
- Maintain the program

Software Development Process - Steps

- Analyze the problem
- Determine Specifications (also called Requirements)
- Create a Design
- Implement the Design
  - Translate the design into a programming language
- Tests/Debug the program
- Maintain the program

Software Development Process - Steps

- Analyze the problem
- Determine Specifications (also called Requirements)
- Create a Design
- Implement the Design
- Tests/Debug the program
  - Try out to see if it is working
  - It could contain ERRORS (also called bugs) that break the program execution
    - DEBUGGING – the process of identifying and resolving the errors
- Maintain the program
  - Continue developing the programs to respond to users needs.



https://en.wikipedia.org/wiki/Software_bug#Etymology

Software Development Process - Steps

- Analyze the problem
- Determine Specifications (also called Requirements)
- Create a Design
- Implement the Design
- Tests/Debug the program
- Maintain the program
  - Continue developing the program to respond to new users needs.

Temperature converter

- There would be other solution(s) to solve the problem?
  - If you would be an expert in AI (Artificial Intelligence)
    - Program would automatically identify from radio news the temperature value
      - Using speech recognition
    - Display / Announce you about the temperature

… how we solve the following …

- Calculate the following sum

$$S_n = \sum_{i=0}^{n} i = 1 + 2 + \cdots + n$$

- If n=2?
- If n=3?
- If n = 100?

… how we solve the following …

- Calculate the following sum

$$S_n = \sum_{i=0}^{n} i = 1 + 2 + \cdots + n$$

- If we rewrite the formula like

$$S_n = S_{n-1} + n$$

- What about S_i?

$$S_i = S_{i-1} + i$$

- Algorithm ?

… how we solve the following …

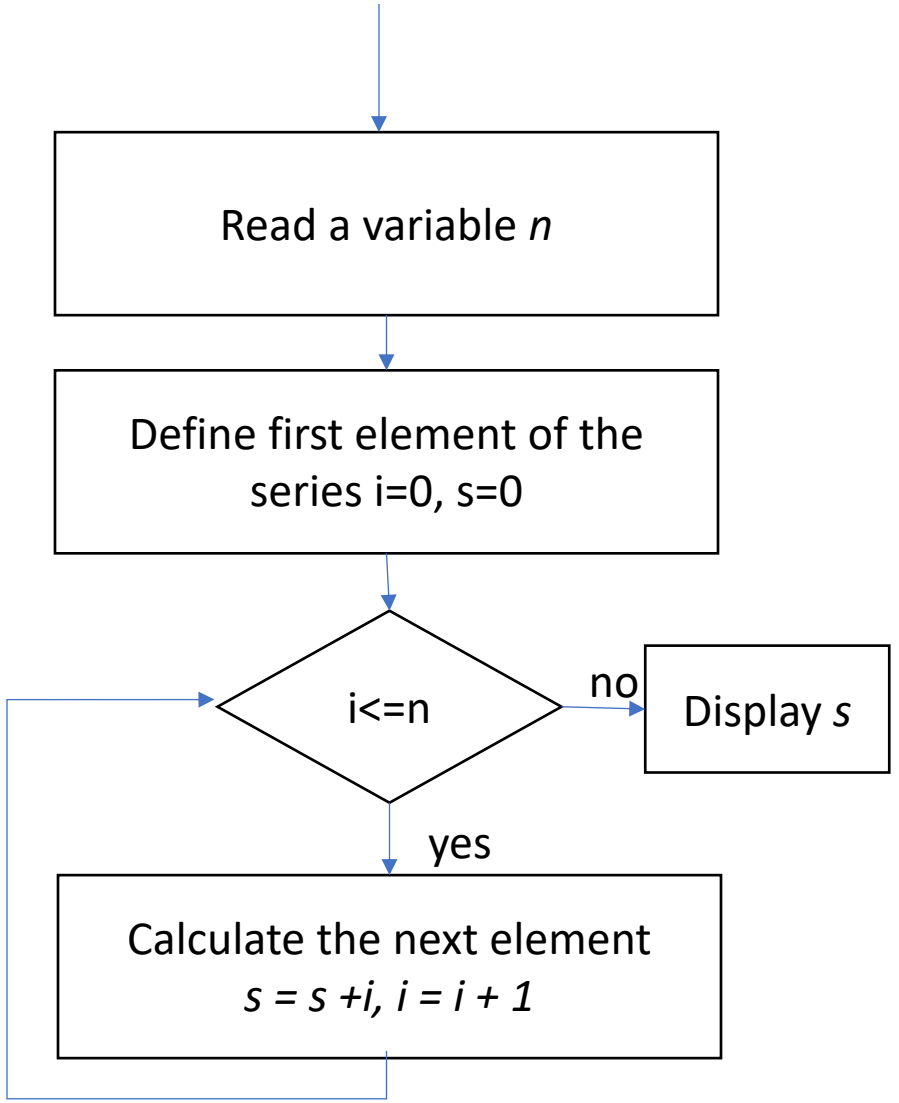
- Calculate the following sum

$$S_n = \sum_{i=0}^{n} i = 0 + 1 + 2 + \cdots + n$$

- What about S_i?

$$S_i = S_{i-1} + i$$

- Algorithm ?
  1) Read a variable *n*
  2) Set *s* with first element of the series (s=0) and set i=0
  3) if i <= m then
     4) Calculate next series element s = s + i
     5) Increment n (i=i+1)
     6) Go to step 3)



Repetitive statements

- In most software, the statements in the program must be repeated several times

- Loop is a control structure that repeats a group of steps in a program
  - Loop body stands for the repeated statements

- The repetitive statements (loops) in Python are **for** and **while**

Repetitive statements - while

- Syntax

while <condition> **:**
    Statement(s)
[**else:**
    statement(s)]

- An expression that evaluates to a Boolean value (True, False)

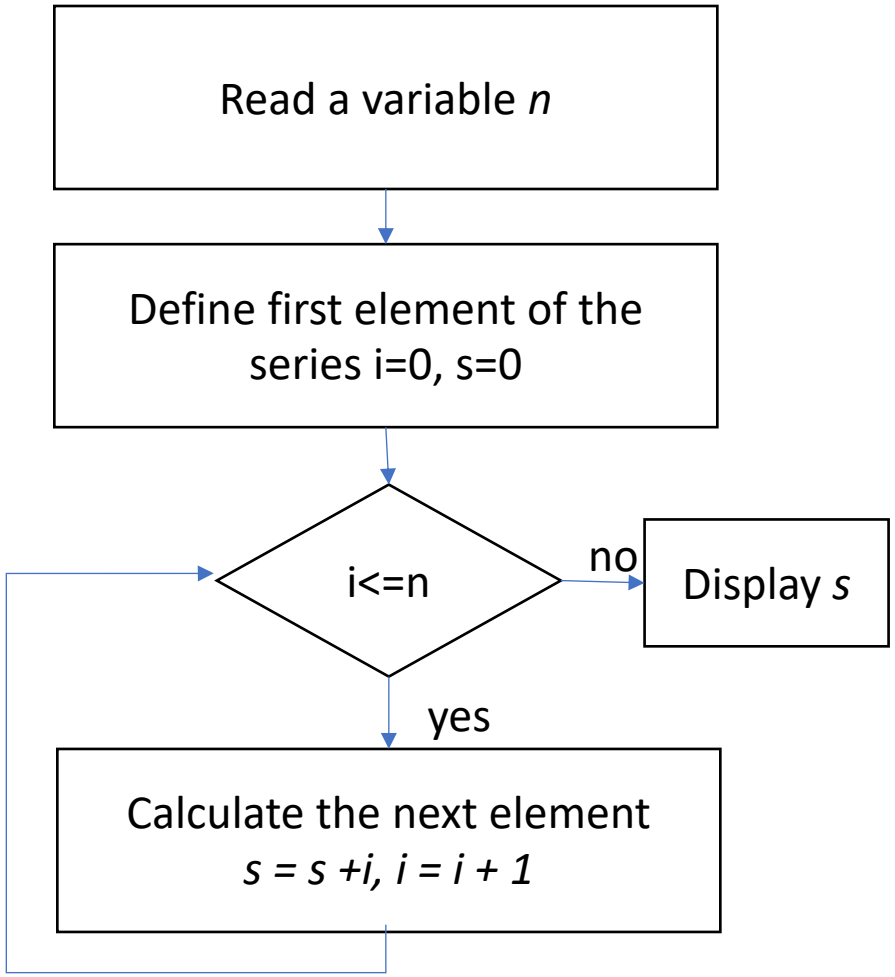
- Loop body, it is executed as long as the *<condition>* is *True*
- Can be formed from one ore more statements
- All statements bellowing to **while** should be at least with one space aligned to right

- Optional clause (can be omitted) specific to Python language that executes when **while** loop finishes

A loop is called infinite loop if its *<condition>* is always *True*.

Repetitive statements - while

Calculate $S_i = S_{i-1} + i$



Translated to Python programming language

```
n = int(input("n="))
s = 0
i = 0
while i <= n:
    s = s + i
    i = i + 1
else:
    print("S=", s)
```

Repetitive statements - while

- More examples
  - A la russe multiplication
    - Multiply two numbers *x* and *y* using the following algorithm:
      - Write x and *y* on the same line
      - Divide x with 2 and write the quotient under *x*
      - Multiply *y* with 2 and write the result under *y*
      - Continue while *x* is different from 1
      - The *n*m* multiplication result is the sum of values from *y* column that correspond to odd numbers on *x* column

Example

X = 13	Y = 25
13	**25**
6	50
3	**100**
1	**200**

Result: x*y = 25 + 100 + 200 = 325

Repetitive statements - while

- More examples
  - A la russe multiplication
    - Multiply two numbers *x* and *y* using the following algorithm:
      - Write x and *y* on the same line
      - Divide x with 2 and write the quotient under *x*
      - Multiply *y* with 2 and write the result under *y*
      - Continue while *x* is different from 1
      - The *n*m* multiplication result is the sum of values from *y* column that correspond to odd numbers on *x* column

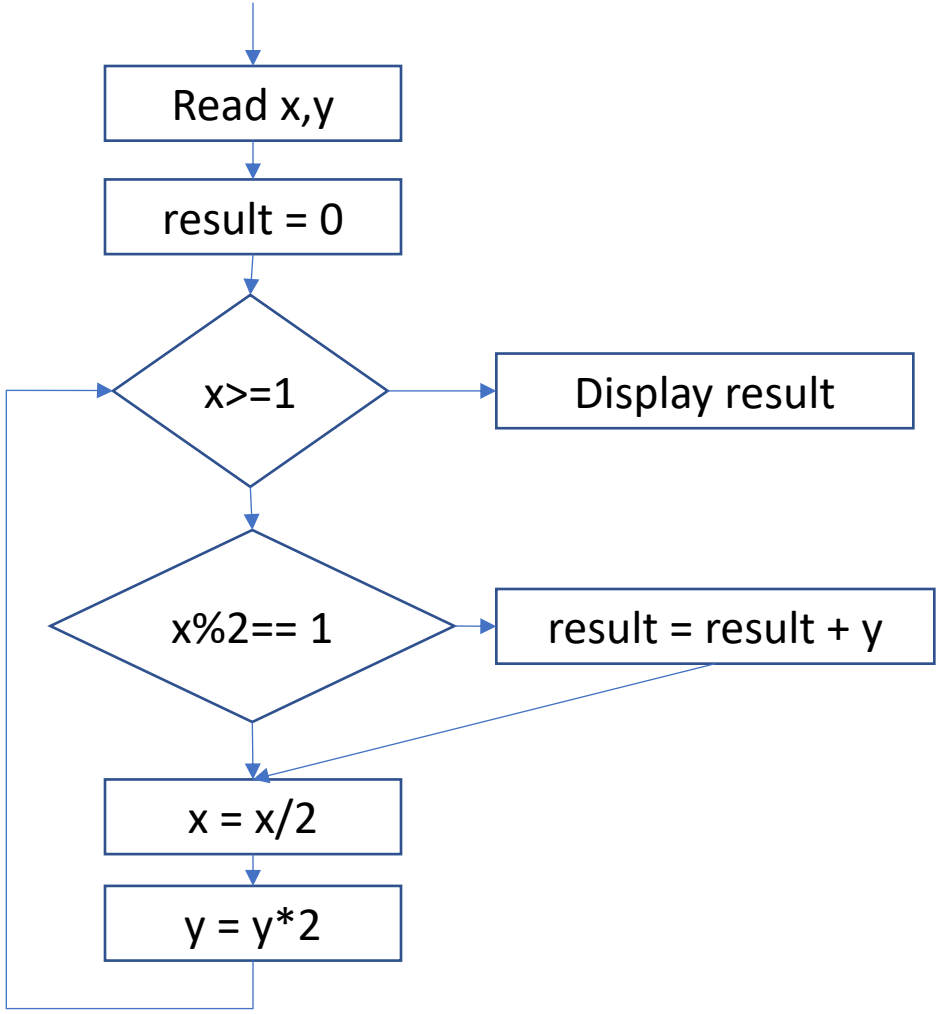
X = 13	Y = 25
13	**25**
6	50
3	**100**
1	**200**

Result: x*y = 25 + 100 + 200 = 325

Lets try to reformulate

Step1: *result* = 0
Step2: if *x* is odd then *result* = *result* + *y*
Step3: *x* becomes *x/2*
Step4: *y* becomes *y*2*
Step5: if *x* not equal with *1* go to *Step2*;
otherwise *result* = *result* + *y*
Step6: display the *result*

Repetitive statements - while



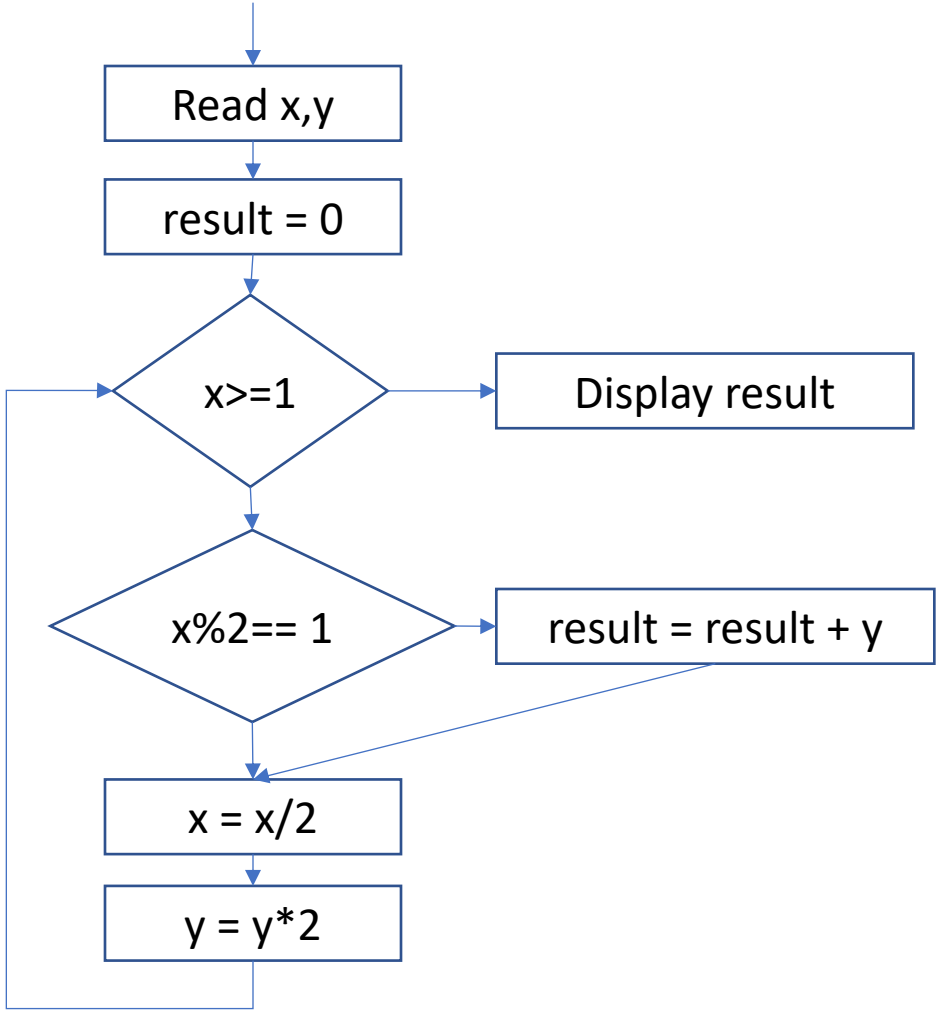
X = 13	Y = 25
13	**25**
6	50
3	**100**
1	**200**

Result: x*y = 25 + 100 + 200 = 325

Lets try to reformulate

Step1: *result* = 0
Step2: if *x* is odd then *result* = *result* + *y*
Step3: *x* becomes *x/2*
Step4: *y* becomes *y*2*
Step5: if *x* not equal with *1* go to *Step2*;
otherwise *result* = *result* + *y*
Step6: display the *result*

Repetitive statements - while




```
x = int(input("x="))
y = int(input("y="))
result = 0
while x>=1:
    if x%2 == 1:
        result = result + y
    x = x // 2
    y = y * 2
print('x*y=', result)
```

X = 13	Y = 25
13	**25**
6	50
3	**100**
1	**200**

Result: x*y = 25 + 100 + 200 = 325

Lets try to reformulate

Step1: *result* = 0
Step2: if *x* is odd then *result* = *result* + *y*
Step3: *x* becomes *x/2*
Step4: *y* becomes *y*2*
Step5: if *x* not equal with *1* go to *Step2*;
otherwise *result* = *result* + *y*
Step6: display the *result*

Repetitive statements - while

- Use to get input from users

```
r=int(input("Response correct at the following? (3+4-2)"))
while r != 5:
    r=int(input("Response correct at the following? (3+4-2)"))
```

- Used to count something

```
i=0  # initialize e value
while i < 5:
    print(i)
    i += 1 #modify the value
```

Repetitive statements - for

- For statements behave differently in Python from other programming languages as C, C++, Java, Pascal
  - It iterates on lists
  - Does not use expressions to iterate

=>

First discuss briefly about lists in Python

Data Structures

- Lists
- Sets
- Tuples
- Dictionaries

Lists

- What is a list?
  - a collection of objects
  - it represents an ordered sequence of data

- Example
  - [1, 2, -3, 5, 7]
  - [‘abc’, ‘efg’, ‘hij’]
  - []
  - lst = [3, 5, 8]

Generating lists of numbers

- Range function
  - Syntax
    - range([start,] stop [, step])
  - Generates a list of numeric values in interval [*start, stop)* with *step* frequency

- Example
  - range(5) → [0, 1, 2, 3, 4]
  - range(2,5) → [2, 3, 4]
  - range(0,5,2) → [0, 2, 4]
  - range(10, 0, -2) → [10, 8, 6, 4, 2]

Back to repetitive statements - for

- For iterates over a sequence (list) of values
- Syntax

  for <variable> **in** <sequence>**:**

  statement(s)

- Example
  - Display the content of a list using for statement

```
lst = [1, 3, 5, 7]
for el in lst:
    print (el)
```

Back to repetitive statements - for

Rewrite using for

```
i=0  # initialize the value
while i < 5:
    print(i)
    i += 1 #modify the value
```

USING FOR

```
for i in range(5):
    print (i)
```

In Python:

Not all you write with **while** can be written with **for.**

Break repetitive statements

- Sometime repetitive statements have to be break
- Break statements
  - Break
    - Interrupt a cycle
  - Continue
    - Skip some of cycle body statements

Break Statement

- A loop control statement which is used to terminate the loop.
- As soon as the break statement is encountered
  - The loop iterations stops
  - The control returns from the loop immediately to the first statement after the loop.

- Example
  - Simulate a two dices throwing, stop when 7 is thrown

```
from random import random
while True:
    dice1 = 1 + int(random()*6)
    dice2 = 1 + int(random()*6)
    print ("dice1=", dice1, "dice2=", dice2)
    if dice1+dice2 == 7:
        break
```

Continue statement

- A loop control statement that is used to skip the remaining statements within the body
  - The loop condition is checked to see if the loop should continue or be exited

- Example
  - Calculate the sum of even numbers of a list

```
l = [23, 45, 66, 77, 98]
s = 0
for el in l:
    if el % 2 == 1:
        continue
    s += el
print("S=", s)
```

Nested loops

- As conditional statements can be nested loops can also be

- How to draw the following figure?

*****

*****

*****

*****

*****

- Solution

```
n = int(input("n="))
for i in range(n):
    for j in range(n):
        print('*', end='')
    print()
```

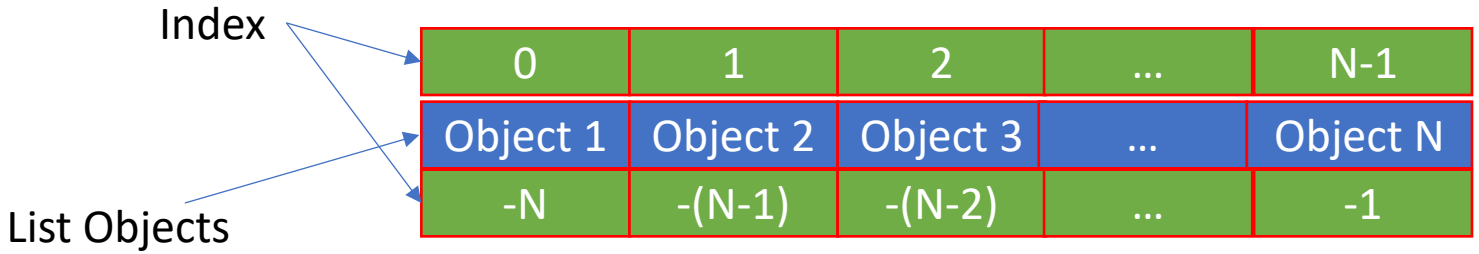
Data Structures Again ….

- The Python language supports native the following data structures
  - Lists
  - Sets
  - Tuples
  - Dictionaries

Lists

- What is a list?
  - a collection of objects
  - it represents an ordered sequence of data
  - Are mutable objects

- Example
  - [1, 2, -3, 5, 7]
  - L1 = [‘abc’, ‘efg’, ‘hij’]
  - []
  - lst = [3, 5, 8]



Python lists are internally represented as arrays.

More about lists

- List are specified using []
- List elements
  - usually homogeneous (ie, all integers)
  - can contain mixed types (not common)
- List elements can be referred by index
  - First index is 0
  - Last index is the length of the list -1

List operations

lst = [“aa”, 3, “bb”, [1, 2]]

- Finding the number of elements of a list
  - len(lst) → 4
- Accessing an element from a list
  - lst[3] → [1, 2]
- Modifying an element of a list
  - lst[3] = “asd” → [“aa”, 3, “bb”, “asd”]
- Adding elements to list
  - lst.append(“zzz”) → [“aa”, 3, “bb”, [1, 2], “zzz”]
  - lst.insert(2, “cc”) → [“aa”, 3, “cc”, “bb”, [1, 2]
- Removing elements from a list
  - lst.pop() → [“aa”, 3, “bb”]
  - lst.remove(3) → [“aa”, “bb”, [1, 2]
  - del(lst[2]) → [“aa”, 3, [1, 2]

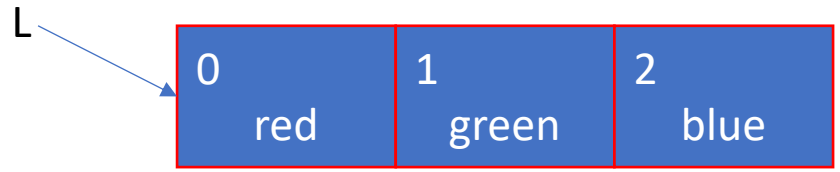
List operations

- Slicing
  - Extracting sublists from list

- Example
  - L = [8, 9, 10, 11, 12, 13, 14, 15]
  - L[3:5] → [11, 12]
  - L[:3] → [8, 9, 10]
  - L[5:] → [13, 14, 15]
  - L[0:6:2] → [8, 10, 12]

List operations

- Sorting
  - sort()
  - sorted()
- Example
  - L = [“red”, “green”, “blue”]



  - L.sort() -> [“blue”, “green”, “red”]
    print(L)

  - print(sorted(L))
    print(L)

A new list is returned by sorted() function that contains the sorted list

Tuples

Immutable - cannot change an element value

- What are tuples?
  - Are sequence of ordered and immutable objects
- Represented with parentheses
- Example
  - T = () #empty tuple
  - T = (“Programming I”, “S1”, 6)
  - T[1] -> accessing value “S1”
  - len(T) -> evaluate to 3
  - (“Programming I”, “S1”, 6) + (3, 4) -> (“Programming I”, “S1”, 6, 3, 4)
  - T[1:3] -> evaluates to ('S1', 6)
  - T[1:2] -> evaluates to ('S1',)

The comma is added to make the object a tuple

Tuple useful for …

- Swapping variables

```
X=Y
Y=X
```
NOT OK

```
aux=X
X=Y
Y=aux
```
OK

```
(X, Y) = (Y, X)
```
OK

- Returning multiple values from a function
  - A function return a single value
  - Tuples allow to return multiple values

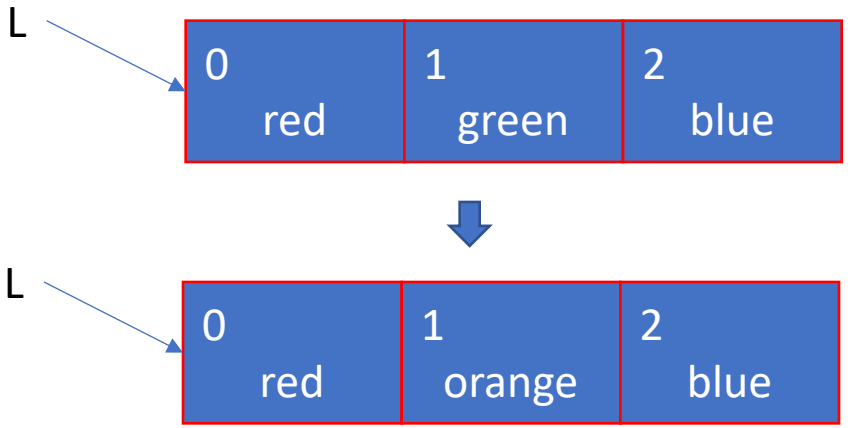
Tuple - Immutable

- Immutable
  - cannot change an element value

- Example
  - T = (“Programming I”, “S1”, 6)
  - T[1] = “S2” -> ERROR

List - Mutable

- Lists are mutable
  - Values of the stored elements can be changed
- Example
  - L = [“red”, “green”, “blue”]
  - L[1] = “orange”



List - Mutable

- Lists are mutable
- Behave differently than immutable types
- Is an object in memory
- Variable name points to object
- Any variable pointing to that object is affected
- Key phrase to keep in mind when working with lists is side effects

MUTATION, ALIASING, CLONING

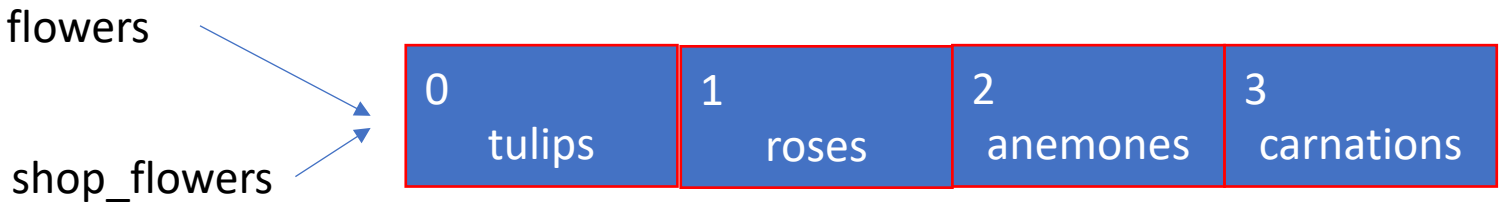
Aliases

```
a=1
b=a
b=2
print(a)
print(b)
```

```
1
2
```

```
flowers = ["tulips", "roses", "anemones"]
shop_flowers = flowers
shop_flowers.append("carnations")
print(flowers)
print(shop_flowers)
```

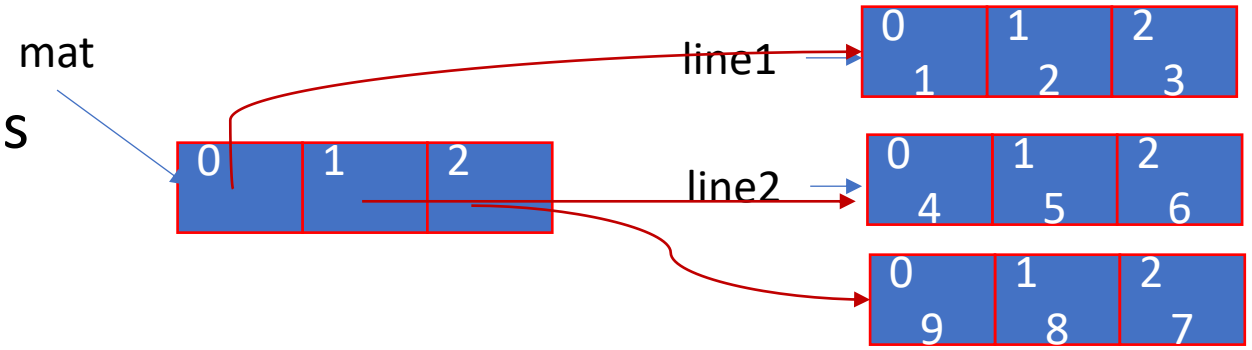
```
['tulips', 'roses', 'anemones', 'carnations']
['tulips', 'roses', 'anemones', 'carnations']
```



Alias are names that refers same values.
Changes done in the value reflect into all aliases variable.

Lists of lists of lists …

- It is possible to define nested lists
- Mutation can be side effect

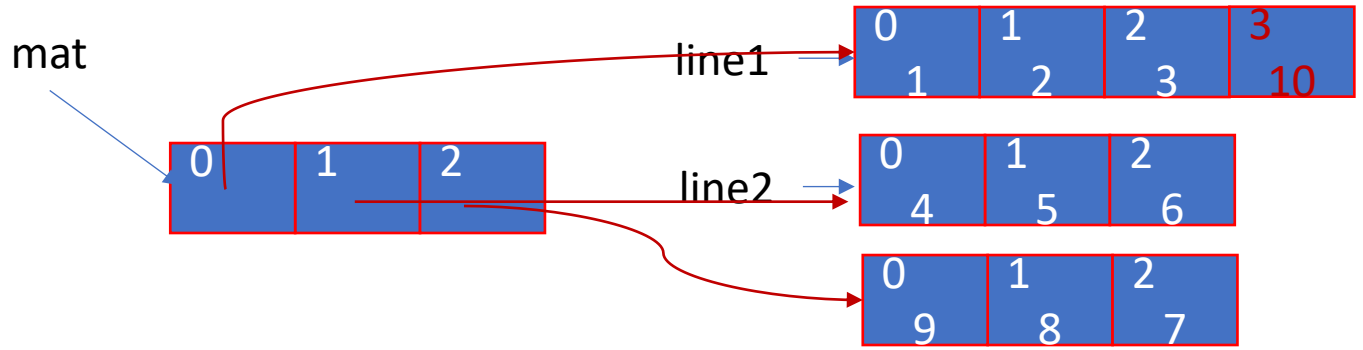



```
line1 = [1, 2, 3]
line2 = [4, 5, 6]
mat = [line1, line2, [9, 8, 7]]
print(mat)
```

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

```
line1.append(10)
print(mat)
```

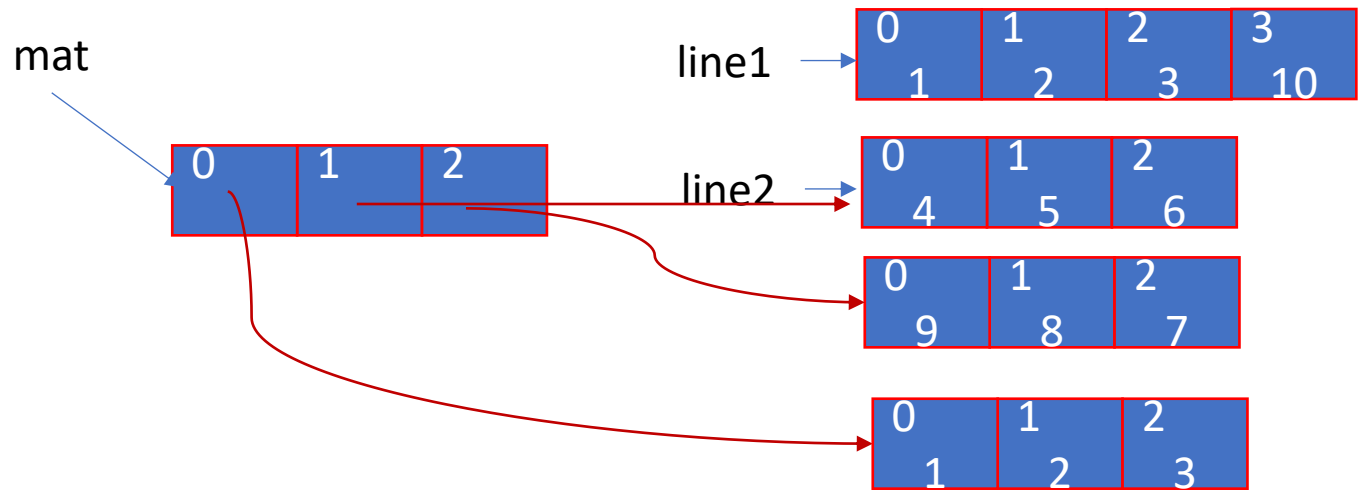
[[1, 2, 3, 10], [4, 5, 6], [9, 8, 7]]



Cloning

```
line1 = [1, 2, 3]
line2 = [4, 5, 6]
mat = [line1[:], line2, [9, 8, 7]]
print(mat)
line1.append(10)
print(mat)
```

```
[[1, 2, 3], [4, 5, 6], [9, 8, 7]]
[[1, 2, 3], [4, 5, 6], [9, 8, 7]]
```



Cloning

- Create a new list and copy every element using [:]

- Example
  - new_list = L1[:]

Set

- A set is an unordered collection of items.
- Every element is unique (no duplicates) and must be immutable.
- Itself is mutable - can add or remove items from it.
- Can be used to perform mathematical set operations like union, intersection, symmetric difference

Set

- Creating

```
S = set()  #empty set
S = {1, 2, 3}
```

S={} #NOT OK is a initialization for other object type dictionary

```
print(type(S))
```

- Adding elements

```
S.add(2)
S.add(2)
```

Set Operations

- Removing elements
  - S.remove(2) #removes the element with value 2
- Union $A \cup B$
  - A.union(B)
- Intersection $A \cap B$
  - A.intersection(B)
- Difference $A - B$
  - A.difference(B)
- Membership $element \in B$
  - element **in** A

Dictionaries

- How to store information about students?

Names =[‘Ionescu Ion’, ‘Popescu Pavel’, ‘Marinecu Maria’]
Current_year_mean = [9.4, 8, 6.78]
Year = [1, 2, 1]

- a **separate list** for each item
- each list must have the **same length**
- info stored across lists at **same index**, each index refers to info for a different person

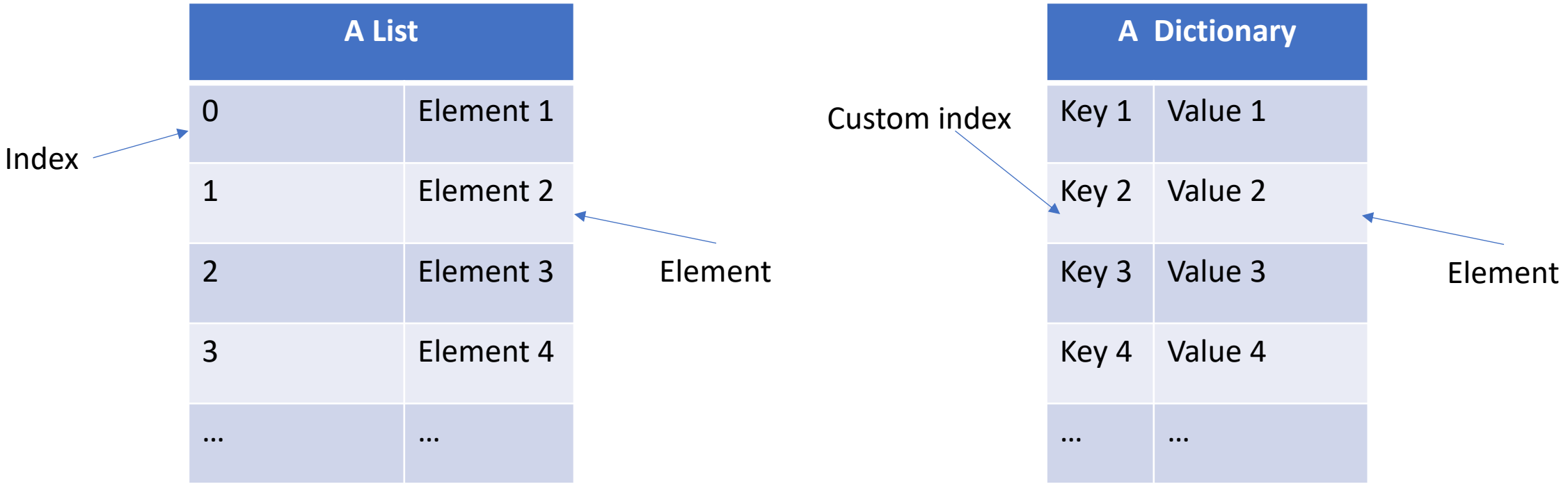
How to update students information?

```
name = input(“Student name”)
i = names.index(name)
Current_year_mean[i] = 8.7
Year[i] = 2
```

- **messy** if have a lot of different info to keep track of
- must maintain **many lists** and pass them as arguments
- must **always index** using integers
- must remember to change multiple lists

Better and clearer - dictionary

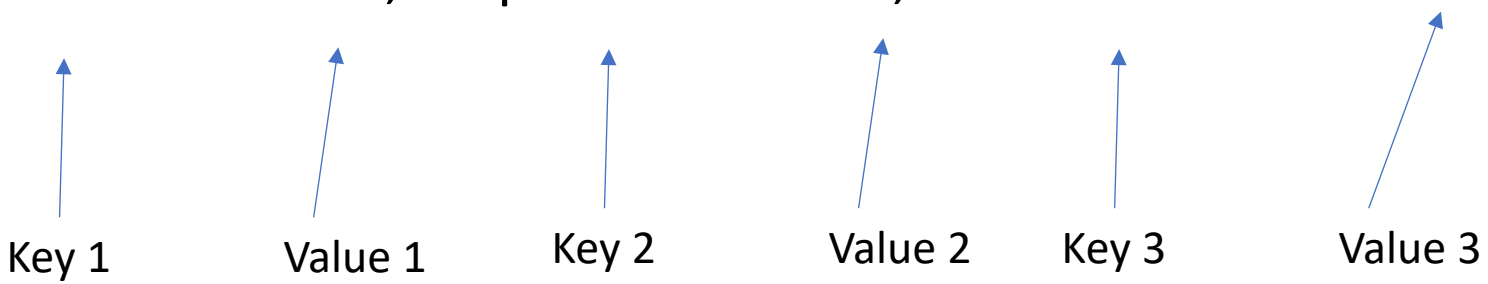
- Use one data structure
- Index based on key not on position in data structure



Dictionaries

- Store pairs of data
  - (key, value)

- Creating
  - dict1={} #empty dictionary
  - dict_grades= {‘Ionescu Ion’ : 9.4, ‘Popescu Pavel’ : 8, ‘Marinecu Maria’ : 6.78}



Dictionary

- Accessing elements
  - Similar with list
  - Using key

- Example
  - dict_grades= {‘Ionescu Ion’ : 9.4, ‘Popescu Pavel’ : 8, ‘Marinecu Maria’ : 6.78}
  - dict_grades[‘Ionescu Ion’] evaluates to 9.4
  - dict_grades[‘Ionescu Vasile’] evaluates to *error key does not exist*

Dictionary Operations

- dict_grades= {‘Ionescu Ion’ : 9.4, ‘Popescu Pavel’ : 8, ‘Marinecu Maria’ : 6.78}

- Add an entry
  - dict_grades[‘Enescu Ene’] = 8.7

- Test if an entry is in dictionary
  - ‘Ionescu Ion’ **in** dict_grades

- Delete an entry
  - del(dict_grades[‘Ionescu Ion’])

Dictionary - itertate

- dict_grades= {‘Ionescu Ion’ : 9.4, ‘Popescu Pavel’ : 8, ‘Marinecu Maria’ : 6.78}

- Get keys
  - dict_ grades.keys()

```
for key in dict_grades.keys():
    print(key)
```

- Get values
  - dict_ grades.values()

```
for value in dict_grades.value():
    print(value)
```

- Get (key, value) pairs
  - dict_ grades.items()

```
for key, value in dict_grades.items():
    print(key, “:”, value)
```

Dictionary keys and values

- Values
  - Any type (**immutable and mutable**)
  - Can be **duplicated**
  - Dictionary values can be lists, even other dictionaries!

- Keys
  - must be **unique**
  - **Immutable** type (int, float, string, tuple, bool)
  - actually need an object that is **hashable,** but think of as immutable as all

- Immutable types are hashable
  - Careful with float type as a key
  - **no order** to keys or values!
    d = {4:{1:0}, (1,3):"twelve", 'const':[3.14,2.7,8.44]}

Lists vs. Dictionaries

Lists

- **ordered** sequence of elements
- look up (reference) elements by an integer index
- indices have an **order**
- index is an **integer**

Dictionaries

- **matches** “keys” to “values”
- look up one item by another item
- **no order** is guaranteed
- key can be any **immutable** type

Bibliography

- https://youtu.be/0jljZRnHwOI?t=1020
- https://www.youtube.com/watch?v=RvRKT-jXvko
- John Zelle, **Python Programming: An Introduction to Computer Science (chapter 2)**