SFU SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

# CMPT 120: Introduction to Computing Science and Programming 1

## Final Review

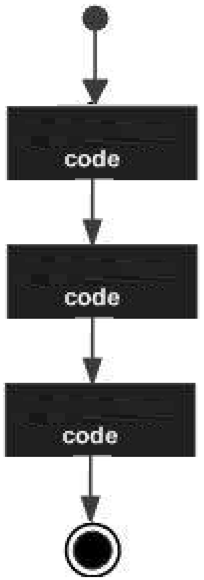
# Control Structures

- **Control Structure**: It is a logical design which refers to the order in which statements in computer programs will be executed.

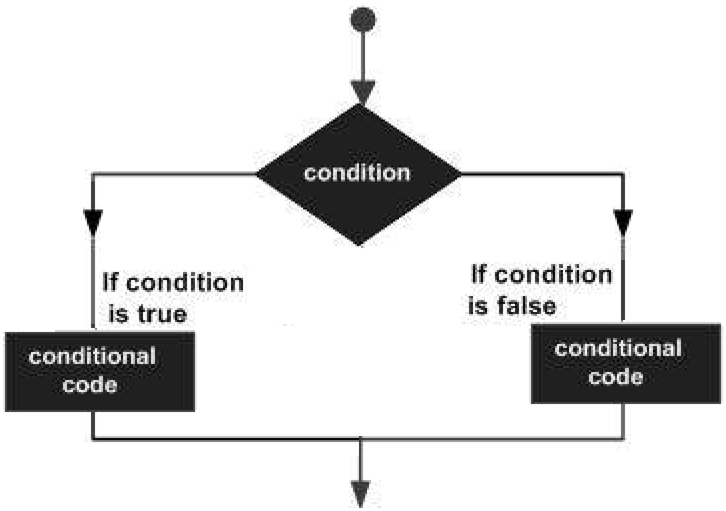
1. **Sequence Structure**: An order where a set of statements is executed sequentially.
2. **Decision Structure**: An order where a set of instructions is executed only if a condition exists.
   a. **Branching**
   b. **Looping**

# Control Structures: Flowcharts

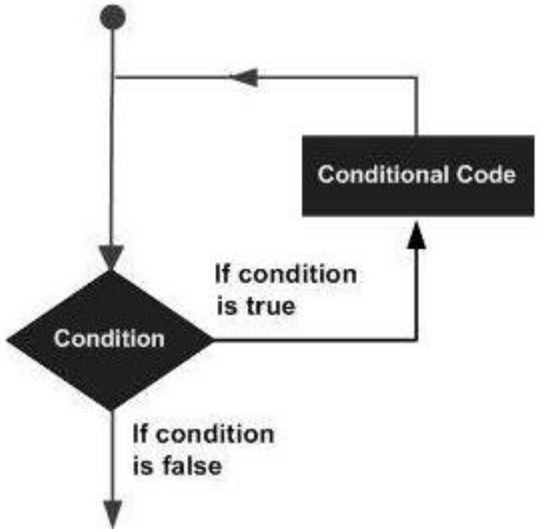
## Sequential Structure




## Decision Structure: Branching




## Decision Structure: Looping



# Decision Structures

- **Branching**: It alters the flow of program execution by making a selection or choice.

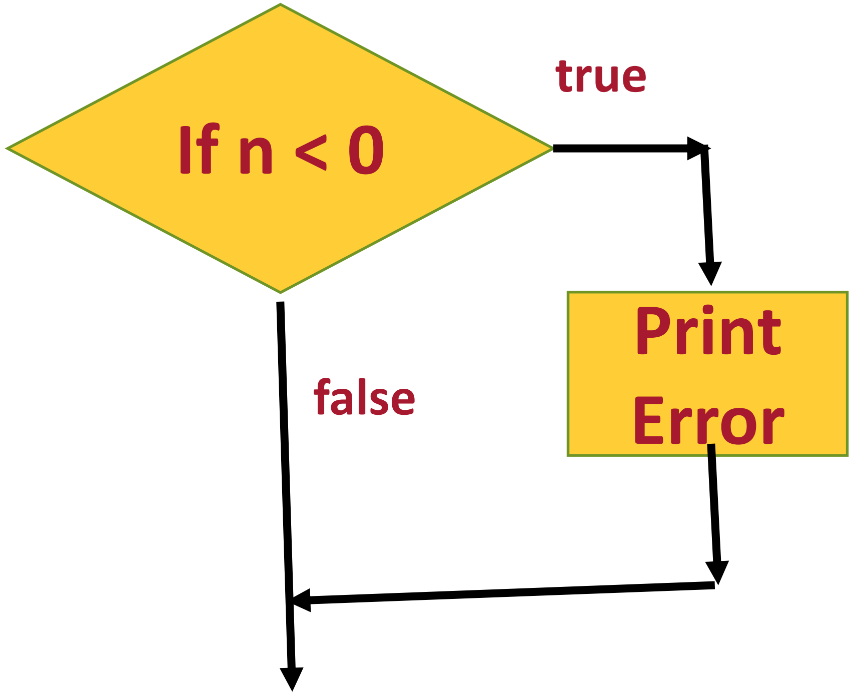
  1. **if**
  2. **if-else**
  3. **If-elif-else** **(A decision structure nested inside another decision structure)**

- **Looping**: It alters the flow of program execution by repetition of a particular block of statement(s).

  1. **for-loop**
  2. **while-loop**

# The if Statement: A Simple Decision Structure

- A simple if statement provides a **single** alternative decision structure.
  - It provides only one alternative path of execution.
  - If condition is not true, exit the structure.



# The if Statement: Syntax

- Python syntax:

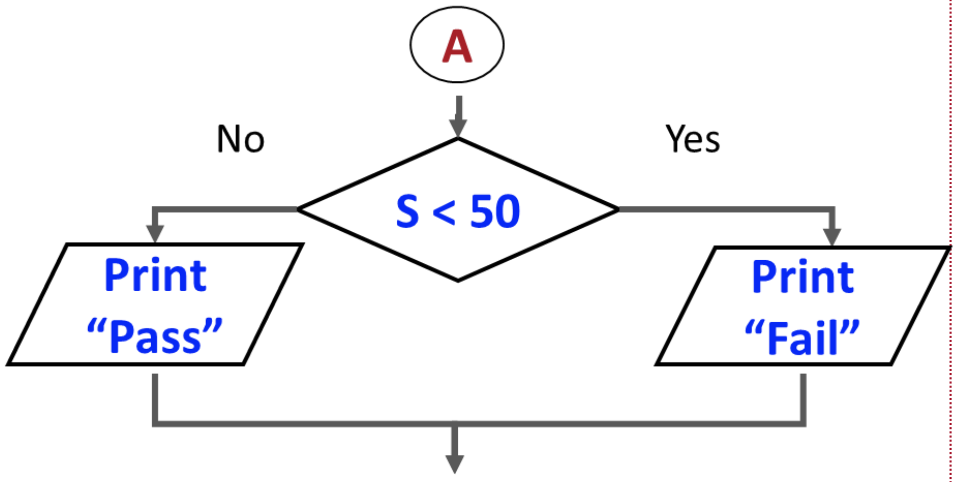
```
if condition:
     Statement
     Statement
```

- **First line** known as the **if clause**.
- It includes the keyword **if** followed by **condition.**
- The condition can be **true** or **false**.
- When the **if statement** executes, the **condition is tested**, and if it is **true** the block statements are executed.
- Otherwise, block statements are skipped.

# The if-else Statement: Dual Alternative Decision Structure

- The if-else decision structure provides:
  - dual alternatives, or
  - two possible paths of execution.

1. One path is taken if the condition is true,
2. And, the other path is taken if the condition is false.



# The if-else Statement: Syntax

- The if-elif-else decision structure allows more than one condition to be tested.
- Python syntax:

```
if condition 1:
    Statement(s)
elif condition 2:
    Statement(s)
elif condition 3:
    Statement(s)
else:
    Statement(s)
```

Insert as many `elif` clauses as necessary.

# Introduction to Loops: Repetition Structures

- **Repetition structure**: A repetition structures makes computer repeat the code (included inside the structure) as many times as required.
  1. **count-controlled** loops (**for** loop **i.e., repeat 5 times, 10 times, 100 times etc.**)
  2. **condition-controlled** loops (**while** loop, **repeat as long as some condition is true.**)

# Count-Controlled Loop (Definite Loop): for Loop

- **Count-Controlled loop**: A definite loop iterates a specific number of times.
- We use a `for` statement to write count-controlled loop.
  - Python **for loop** is designed to work with **sequence of data items**
    - The for loop repeats or iterates once for each item in the sequence.
- General format:

```
for variable in range/list [val1, val2, etc]:
    statements
```

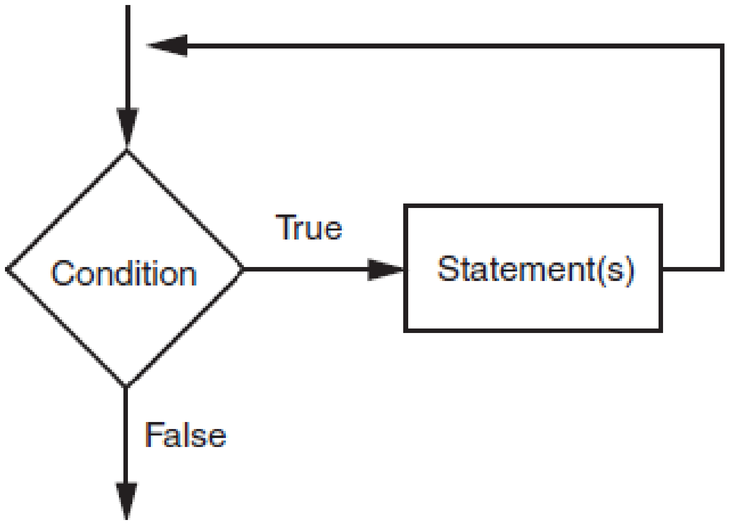
- We refer to the first line as the **for clause**.
- Inside brackets a sequence of values, separated by comma, appear.

# The `while` Loop: Condition-Controlled Loop

- **Condition-Controlled loop**: An **indefinite** loop that iterates an **unspecified** number of times.
  - General format:

```
while condition:
    statements
```

- The loop executes while the **condition** is **true**.
- Based on the result of the **condition**, statements inside the loop may get executed:
  - **zero** time, or
  - **one** time, or
  - **any** number of times.
- We refer to the first line as the **while clause**.



# Nested Loops

- Nested loop: loop that is contained inside another loop

- Key points about nested loops:
  - Inner loop goes through all of its iterations for each iteration of outer loop
  - Inner loops complete their iterations faster than outer loops

# Binary Data Representation

- Data inside computer is **not represented** the same way as we represent numbers and letters in English or native language. **For example**:
  - We represent quantities using symbols (digits) **0, 1, 3,… and 9.**
  - We can write names using English letters **A, B, C,…Z** or **a, b, c,…z**
    - So, we represent a quantity **six** by using the symbol **6**.
    - Using English alphabets, we can represent a street name as: **Dawson Street.**
- **Problem!!!**
- Computer don't use (recognize) the symbols 0,1,2..9 or alphabets a, b, c,…z
- Because, computers use a completely **different** language to represent numbers or letters (or data).
- We call it machine language. (Or, binary language or representation.)

# Binary Data Representation - 2

- The **binary language** consists of two symbols only: **0** and **1**
- That means, every thing in computer **MUST** be represented using the symbols **0** and **1,** only
- So, the quantity **six** must be represented using a commination of **0s** and **1s**. (Binary code)
- The name **Dawson Street** must also be represented using a commination of 0s and 1s.
- Let's create **our own binary codes** to represent letters A, B, C, …Z using a combination of **0s and 1s**.

# Examples

| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 128 | 0 | 32 | 0 | 8 | 0 | 2 | 1 |

= 171

| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 0 | 0 | 32 | 0 | 0 | 0 | 2 | 1 |

= 35

# Storage Units

- **Bit**: storage to represent a **binary** 0 or 1.
- **Byte**: a group of 8-bits.
- More bigger storage units (with approximation, as shown in Study Guide):

| Prefix | Symbol | Factor |
|---|---|---|
| (no prefix) | | $2^0 = 1$ |
| kilo- | k | $2^{10} = 1024 \approx 10^3$ |
| mega- | M | $2^{20} = 1048576 \approx 10^6$ |
| giga- | G | $2^{30} = 1073741824 \approx 10^9$ |
| tera- | T | $2^{40} = 1099511627776 \approx 10^{12}$ |

- Example, "**12 megabytes**" is: **12 × 2^20** bytes
- = **12,582,912** bytes =>

**12582912 × 8** bits = **100,663,296** bits .

- **More specifically:**

**Multiples of bytes** V·T·E

| Decimal | | | Binary | | | | |
|---|---|---|---|---|---|---|---|
| Value | Metric | | Value | IEC | | JEDEC | |
| 1000 | kB | kilobyte | 1024 | KiB | **kibibyte** | KB | kilobyte |
| $1000^2$ | MB | megabyte | $1024^2$ | MiB | mebibyte | MB | megabyte |
| $1000^3$ | GB | gigabyte | $1024^3$ | GiB | gibibyte | GB | gigabyte |
| $1000^4$ | TB | terabyte | $1024^4$ | TiB | tebibyte | – | |
| $1000^5$ | PB | petabyte | $1024^5$ | PiB | pebibyte | – | |
| $1000^6$ | EB | exabyte | $1024^6$ | EiB | exbibyte | – | |
| $1000^7$ | ZB | zettabyte | $1024^7$ | ZiB | zebibyte | – | |
| $1000^8$ | YB | yottabyte | $1024^8$ | YiB | yobibyte | – | |

Orders of magnitude of data

# Signed Integer Data Representation: Binary

- A **signed integer**: For a positive integer represented by N binary digits the possible values are $-2^{N-1}-1 <= \text{value} <= 2^{N-1}-1$.

| Sign bit | N -1 Binary Digits |
|---|---|

+/- 127

| | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| +/- | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
| | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| +12 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| -12 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

# Signed Integer Data Representation: One's Complement

- Integer is represented by a string of **binary** digits.
  - But, is represented in 1's compliment form.

| Sign bit | N -1 Binary Digits: 1's Compliment |
|---|---|

- **How a number is converted to its 1's Compliment form:**

1. If a number is positive, simply convert the number to its binary equivalent.
   - For example, if the number is: **6** **0 0 0 0 0 1 1 0**
2. If a number is negative, **convert** the number to its binary equivalent and **flip** the bits.
   - For example , if the number is: -6 **0 0 0 0 0 1 1 0**
   - Flip the bits: **1 1 1 1 1 0 0 1**

# Signed Integer Data Representation: One's Complement

- Suppose an 8-bit 1's pattern is shown as: **1 0 1 1 0 0 0 1**

- **What number this pattern represents?**
  - If first bit 0, then it is an unsigned/positive number, as shown (simply convert it to its decimal equivalent).
  - If first bit is 1, then:

1. Flip all the bits. So, **1011 0001** becomes **0100 1110**
2. Convert to decimal: 01001110 $= 2^6 + 2^3 + 2^2 + 2^1 = 64 + 8 + 4 + 2 = 78$
3. Add a minus sign. So **10110001** represents **-78** in one's Complement form.

# Two’s Complement Signed Integer Representation

- **Integer is represented by a string of binary digits.**
  - Representation is in 2’s compliment form.
  - Right most bit is used for sing.
  - Remaining bits represent the value.

| Sign bit | N-1 Binary Digits: 2's Compliment |
|---|---|

- **Decimal to 2’s Compliment form:**
- **For a Positive Number:**
  1. First bit is 0.
  2. Convert the number to its binary equivalent.
- **+ 7** is represented as: 0000 0111
- **+ 13** is represented as: 0000 1101

- **For a Negative Number:**
  1. Convert the number to its binary equivalent.
  2. Flip the bits
  3. Add 1.
- **- 7** would be represented as:
  1. Convert to binary: 0000 0111
  2. Flip the bits: 1111 1000
  3. Add 1. 1 = 1111 1001
- **- 13** would be represented as:
  1. Convert to binary: 0000 1101
  2. Flip the bits: 1111 0010
  3. Add 1. 1 = 1111 0011

# Turtle Intro

Turtle is a Python feature that allows you to draw and animate graphic shapes.

```
# Import turtle package
import turtle

# Create our turtle
myTurtle = turtle.Turtle()

# Move forward 50 pixels
myTurtle.forward(50)

# Turn right 90 degrees
myTurtle.right(90)

# Move forward 50 pixels
myTurtle.forward(50)
```

Create a Turtle **"object"**

# Using turtle in Python

- To make use of the turtle methods and functionalities, we need to import turtle.
- ”turtle” comes packed with the standard Python package and need not be installed externally.
- Four steps for executing a turtle program :
  1. **Import** the turtle module
  2. **Create** a turtle to control (using **Turtle()**)
  3. **Draw** around using the turtle methods.
  4. Run **turtle.done()**.

# Common Turtle Methods (See Documentation)

| METHOD | PARAMETER | DESCRIPTION |
|---|---|---|
| Turtle() | None | Creates and returns a new tutrle object |
| forward() | amount | Moves the turtle forward by the specified amount |
| backward() | amount | Moves the turtle backward by the specified amount |
| right() | angle | Turns the turtle clockwise |
| left() | angle | Turns the turtle counter clockwise |
| penup() | None | Picks up the turtle's Pen |
| up() | None | Picks up the turtle's Pen |
| down() | None | Puts down the turtle's Pen |
| color() | Color name | Changes the color of the turtle's pen |
| fillcolor() | Color name | Changes the color of the turtle will use to fill a polygon |

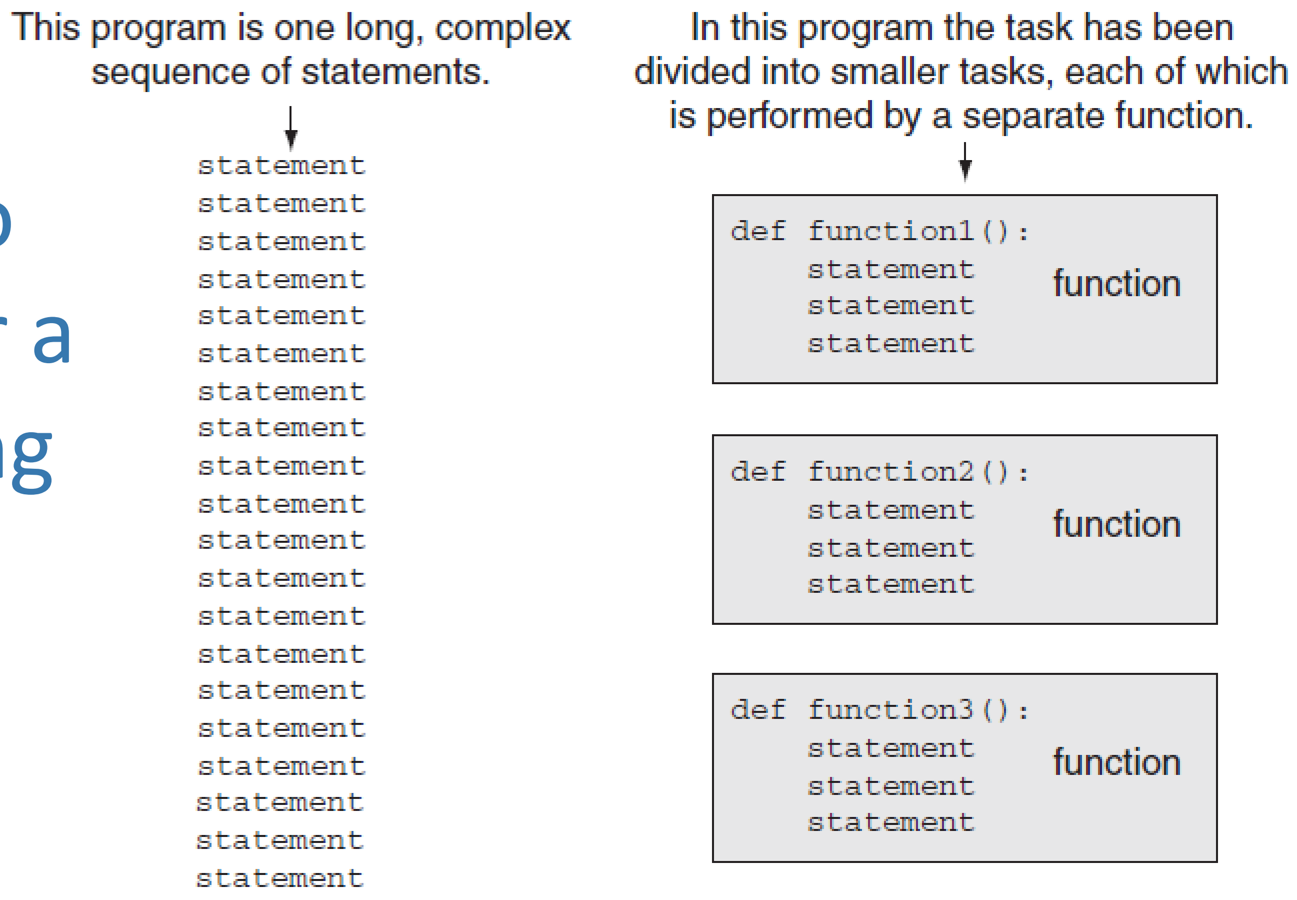
Adapted from: Janice Regan, 2013.

# Introduction to Functions

- **Function**: group of statements within  a program that perform as specific task.
  - Usually one task of a large program.
    - Functions can be executed in order to perform overall program task.
  - Known as *divide and conquer* approach
- Modularized program: program wherein each task within the program is in its own function.

# Functions: A Divide and Conquer Approach

- We use functions to Divide and Conquer a large task by dividing into subtasks.
- We also call it a modular approach.



# Void Functions and Value-Returning Functions

- A void function:
  - Simply executes the statements it contains and then terminates.
- A value-returning function:
  - Executes the statements it contains, and then it returns a value back to the statement that called it.
    - The `input`, `int`, and `float` functions are examples of value-returning functions.

# Defining and Calling a Function

- **Functions are given names (like we give names to variables).**
  - Function naming rules:
    - Cannot use key words as a function name.
    - Cannot contain spaces.
    - First character must be a letter or underscore.
    - All other characters must be a letter, number or underscore.
    - Uppercase and lowercase characters are distinct.

# External storage

- When we shut down an application (e.g.: Python IDLE, Word or Excel) and/or turn off our computer, often we do not want our information (code, data) to disappear.
  - We want our information to **persist** until the next time we use it.
  - We achieve persistence by saving our information to **files** on external storage like **hard disk**, **flash memory**, etc…
  - We can use text files to store the input/output data.

# Files

- **Text Files:**
  - The sequence of 0's and 1's represents human-readable characters, i.e., UNICODE/ASCII characters
  - To view the content of a text file, one needs to use the appropriate application such as a text editor (notepad).
  - Example:
  - In CMPT 120, we shall open or read text files to get data in to the program, or to write from a program.

# Introduction to Recursion

# Hardcode data inside program.

```
quiz1 = 45
quiz2 = 56

total = quiz1 + quiz2
print(total_mark)
```

# Get data using input() function.

```
quiz1 = int(input())
quiz2 = int(input())

total = quiz1 + quiz2
print(total_mark)
```

# Get data from a text file.

```
# Opening a file for reading
fileR = open('mark_data.txt', 'r')
# Read its first line -> a string
quiz1 = fileR.readline()
# Read its second line -> a string
quiz2 = fileR.readline()
quiz1 = int(quiz1)
quiz2 = int(quiz2)
total = quiz1 + quiz2
print(total)
# Close the file
fileR.close( )
```

# Open a file in a Python program

- To use a file in our Python program, we must first open it in the appropriate mode:

**Syntax:**

```
<fileObject> = open(filename, <mode>)
```

Optional **string** describing the way in which the file will be used.

# A word about <mode>

- A mode describes the way in which a file is to be used
- Python considers the following modes:

1. Read
2. Write
3. Append
4. Read and write

# Open a File for Reading

- To read from a file, we need to first open it in **read** mode with `'r'`:

**Syntax:** fileRead = open(<filename>, 'r’)

OR fileRead = open(<filename>)

- `fileRead` is (called) a file object.

- If the file does not exists in the current directory, then:
  - Python interpreter produces and prints an error.

```
FileNotFoundError: [Errno 2] No such file or
                 directory: 'fileDoesNotExist.txt'
```

# Dictionaries

- We have used variables and lists to store data previously.

  For example, **quiz_1 = 14** or

  **marks_list = [12, 15, 40, 30]**
- **Dictionary**: is another object in Python that stores a **collection of data**.
- We use **{ }** to define data in a dictionary.
- Each element in a dictionary consists of a **key** and a **value.**

  **Format:** **<dictionary_name> = {key1:val1, key2:val2, …}**
- Often referred to as mapping of key to value
- To retrieve a specific value, use the **key** associated with it.

# Retrieving a Value from a Dictionary

- To retrieve a specific value, use the **key** associated with it.
- General **format** to retrieve a from a dictionary: **dictionary_name[key]**
- If **key** is in the dictionary, associated value is returned, otherwise, **KeyError** exception is raised.
- To test whether a **key** is in a dictionary use the **in** and **not in** operators.
    - These operators can helps prevent **KeyError** exceptions.
- Elements in dictionary are unsorted

# Adding Elements to an Existing Dictionary

- Dictionaries are **mutable** objects
- To add a new **key-value** pair: `dictionary_name[key] = value`
  - If **key** exists in the dictionary, the value associated with it will be changed. Else, added.

```
country_population = {'Canada' : 36624199, 'USA' : 324459463}

country_population['Mexico'] = 129163276

print(country_population)

{'Canada' : 36624199, 'USA' : 324459463, 'Mexico' : 129163276}
```

# Some Dictionary Methods

- **clear()** method: Deletes all the elements in a dictionary, leaving it empty.
- Format: **dictionary_name.clear()**
- **get()** method: Gets you a **value** associated with specified the specified **key**.
- Format: **dictionary_name.get(key, default)**
  - **default** is returned if the key is not found.

    **print(country_population.get('China',** 'No Value Found'**))**
  - Alternative to **[ ]** operator.
  - Cannot raise KeyError exception.

# Sequences

- **Sequence**: an object that contains **multiple items** of data. For instance:
  - my_list = [ 6, 78, 9] is an example of a sequence.
    - The distinctive name of the this sequence is **list**.
    - So list is a type of sequence.
  - The items are stored in sequence one after another.
- Python provides different types of sequences, including lists and tuples.
  - The difference between these is that:
    - a list is **mutable**
    - a tuple is **immutable**

# Lists

- List: an object that contains multiple data items separated by a comma.
  - An data item in a list is called an **Element**.
  - Format: `list = [item1, item2, etc.]`
  - A list can hold items of different types.
  - my_list = [7,  “Ted”, [56, 78]]
    - Contains three elements of type int, str and list.
- `print` function can be used to display an entire list.
- `list()` function can convert certain types of objects to lists.
  - For instance, to convert a tuple into a lit.

# The Repetition Operator and Iterating over a List

- Repetition operator: makes multiple copies of a list and joins them together
  - The * symbol is a repetition operator when applied to a **sequence** and an **integer**.
    - Sequence is left operand, number is right
  - General format: `list * n`
  - [7, “Ted”, [56, 78]] * 2 = [7, “Ted”, [56, 78], 7, “Ted”, [56, 78]]
- You can iterate over a list using a `for` loop
  - Format: `for x in list:`

# Indexing

- Index: a number specifying the position of an element in a list
  - Enables access to individual element in list
  - Index of first element in the list is 0, second element is 1, and n'th element is n-1
  - Negative indexes identify positions relative to the end of the list
    - The index -1 identifies the last element, -2 identifies the next to last element, etc.

# The `len` function

- An `IndexError` exception is raised if an invalid index is used.
- `len` function: returns the length of a sequence such as a list
  - Example: `size = len(my_list)`
  - Returns the number of elements in the list, so the index of last element is `len(list)-1`
  - Can be used to prevent an `IndexError` exception when iterating over a list with a loop.
    - for i in range(len(my_list)):

# Lists Are Mutable

- Mutable sequence: the items in the sequence can be changed
  - Lists are mutable, and so their elements can be changed
- An expression such as
- `list[1] = new_value` can be used to assign a new value to a list element.
  - Must use a valid index to prevent raising of an `IndexError` exception

# Introduction to Searching

- Have your ever used **Ctrl-F** keys?
  - We use it to **search** a **value**.
  - How to search a value – how to search it fast?
- **Searching**: Locating an item in a list of data.
- Two of search algorithms are:
  1. **Linear** or Sequential Search.
  2. **Binary** Search.
     - Half-interval search.
     - Logarithmic search.

# Linear Search

- Starting at the first element, this algorithm steps through an array **sequentially**, examining each element until it locates the desired value.
  - Suppose, an array **list** contains following values:

| | | | | | | |
|---|---|---|---|---|---|---|
| 17 | 23 | 5 | 11 | 2 | 29 | 3 |

list[0] list[6]

  - To search a **value 11**, Linear Search compares 17, 23, 5, and 11.
  - Say, we define two variable:
  - **VALUE = 11**
  - **found = False**
  - How you will perform this Linear Search?

# Big O

- Estimate the order of the number of calculations needed
  - Order is the largest power of n in the estimated upper limit of the number of operations.
- For most n (amount of data) it is generally true that an order $n^k$ algorithm is significantly faster than an order $n^{k+1}$ algorithm
- An algorithm with order n operations is said to run in linear time
- An algorithm with order $n^2$ operations is said to run in quadratic time.

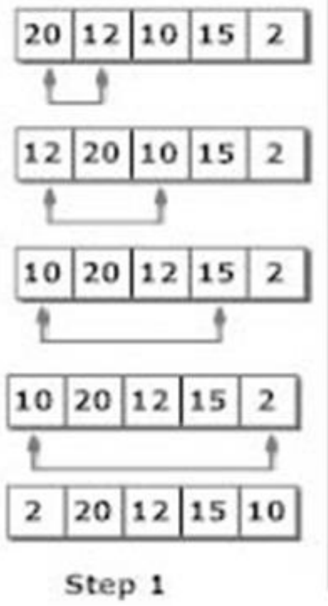
# Estimate of how fast

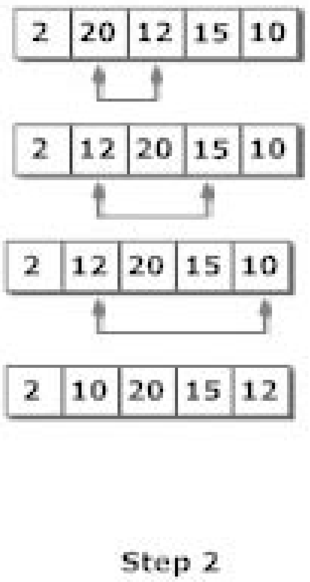
- Looking for a 'good' upper limit
- Just consider the Order.
  - The order is the largest power of n
- First example: 9 operations
  - $\mathcal{O}(9) = 0$ Order 0 (not a function of n)
- Second example: 6*n +1 operations
  - $\mathcal{O}(6*n +1) = n$ Order 1 (largest power of n is 1)
- Third example: $1+3n+11n^2$
  - $\mathcal{O}(1+3n+11n^2) = n^2$ Order 2 (largest power of n is 2)

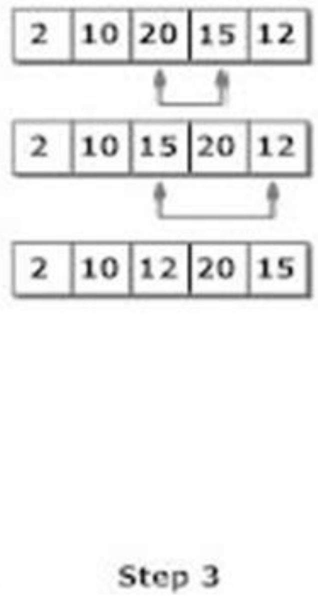
# Introduction to Sorting

- Sorting: Arranging values into an order:
  - Alphabetical
  - Ascending numeric
  - Descending numeric
- One of the simplest algorithms is **Selection sort**.

# Selection Sort Example (Ascending Order)





**Iteration 1:**

1. Find the smallest element between lis[0] and lis[4].
2. Swap if smaller.

**Iteration 2:**

1. Find the smallest element between lis[1] and lis[4].
2. Swap if smaller.

**Iteration 3:**

1. Find the smallest element between lis[2] and lis[4].
2. Swap if smaller.

**Iteration 4:**

1. Find the smallest element between lis[3] and lis[4].
2. Swap if smaller.

# ?

# Questions?