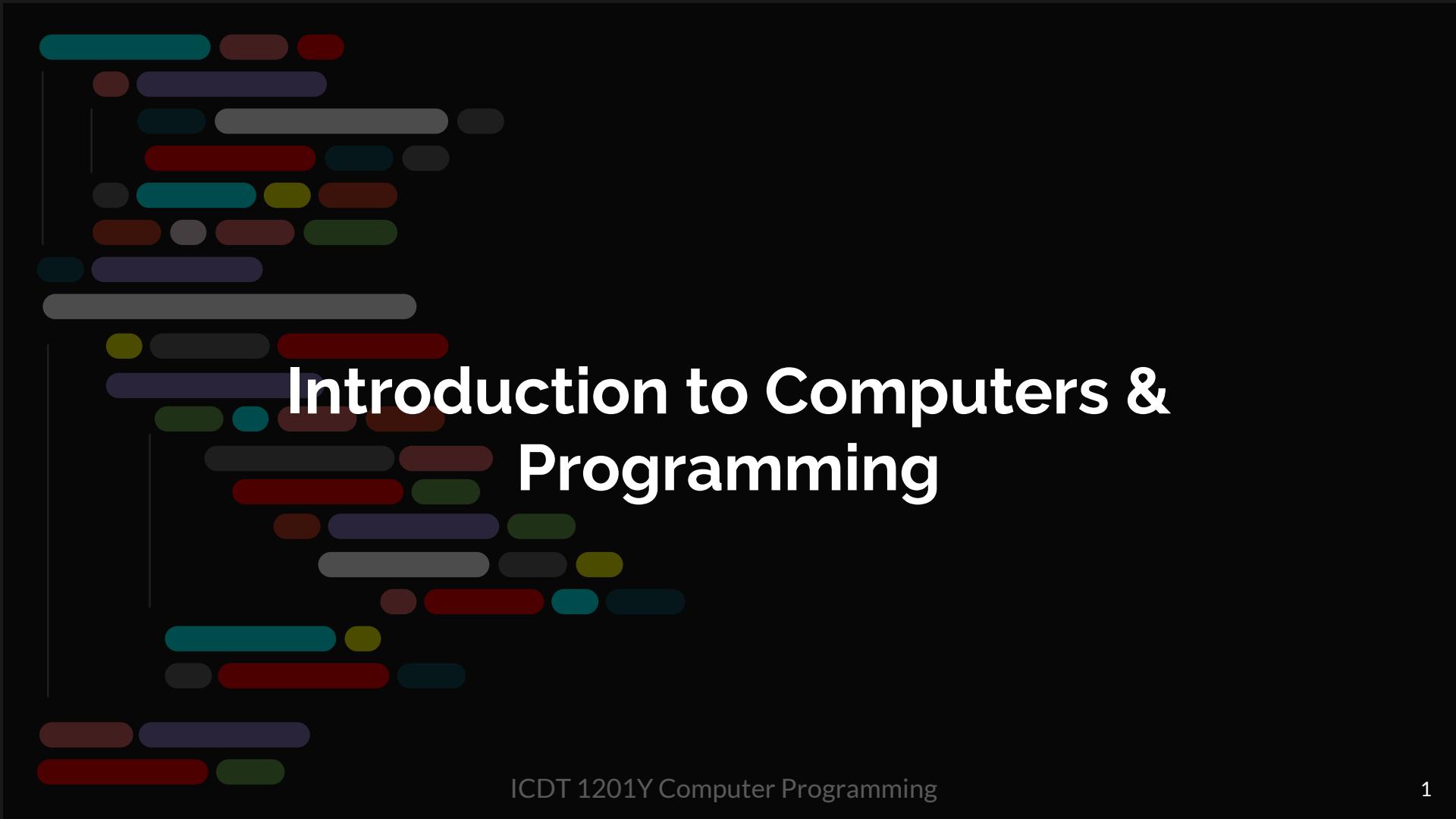




ICDT 1201Y (3)

Computer Programming (week 1)



Introduction to Computers & Programming



Sub-topics

- Computer Model
- Computer Programs
- Translation



Learning Outcomes & Assessment Criteria

Learning Outcomes:

- **Understand the basic components of a computer system:**
 - Students should be able to identify and explain the main components of a computer system (CPU, memory, storage, input/output devices) and their role and function of each component in the execution of programs.
- **Explain what a computer program is:**
 - Students should be able to define what a computer program is and explain how it is used to solve problems.
 - Understand the process of writing, compiling, and running a program.
- **Understand the concept of translation in programming:**
 - Students should be able to distinguish between different types of translators (compilers, interpreters, assemblers). Explain the role of each type of translator in the execution of a program.

Assessment Criteria:

- **Recognition of Computer Components:**
 - Ability to correctly identify the main components of a computer system.
 - Describe the function and importance of each component in simple terms.
- **Definition and Purpose of Programs:**
 - Provide a clear definition of a computer program.
 - Explain, in basic terms, how programs are used to solve problems.
- **Translation Process in Programming:**
 - Distinguish between compilers, interpreters, and assemblers.
 - Explain how each translator works and their role in program execution.



Ice-Breaking Exercise

- What's your idea about software?
- Let's talk about Computer Programming
 - Who are those with prior programming experience?
 - Programming Languages you have heard about?
 - Tips for mastering this module

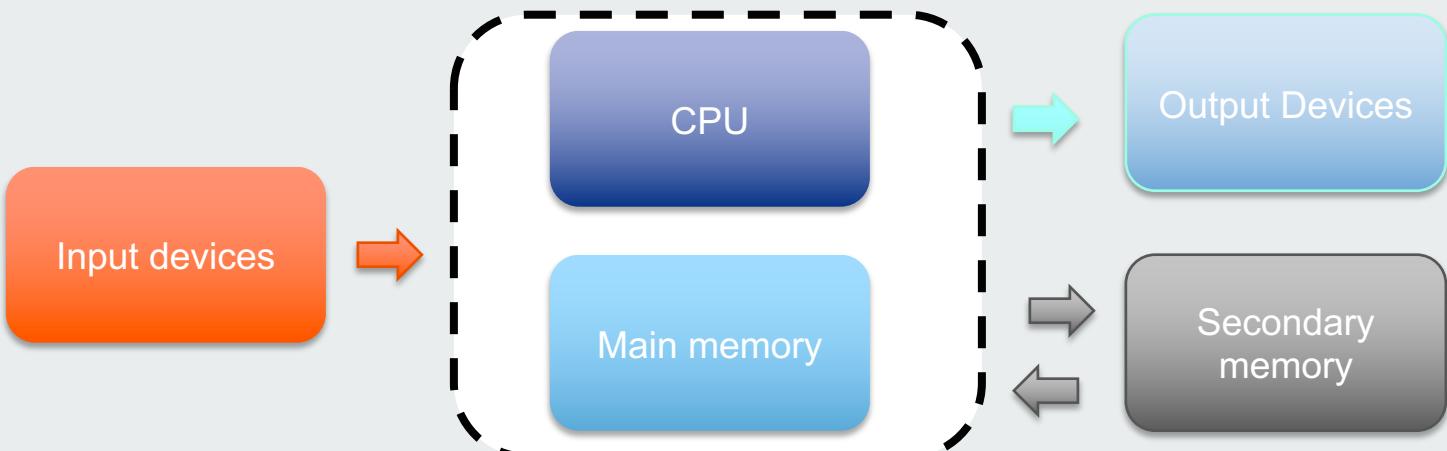
Computer – An Everyday Analogy

- Feed some fruits and veggies to the juicer.
- The juicer crushes and juices the Fruits & Veggies
- Finally the fresh and nice juice comes out
- Input->Process->Output



What is a Computer

- A typical computer consists of Hardware (a CPU, memory, hard disk, input/output devices, and communication devices) and Software.





Programs / Software

Computer programs, known as **software**, are instructions to the computer.

You tell a computer what to do through programs. Without programs, a computer is an empty machine. Computers do not understand human languages, so you need to use computer languages to communicate with them.



Software

Software are programs written to perform specific tasks

Application programs perform a specific task for the user

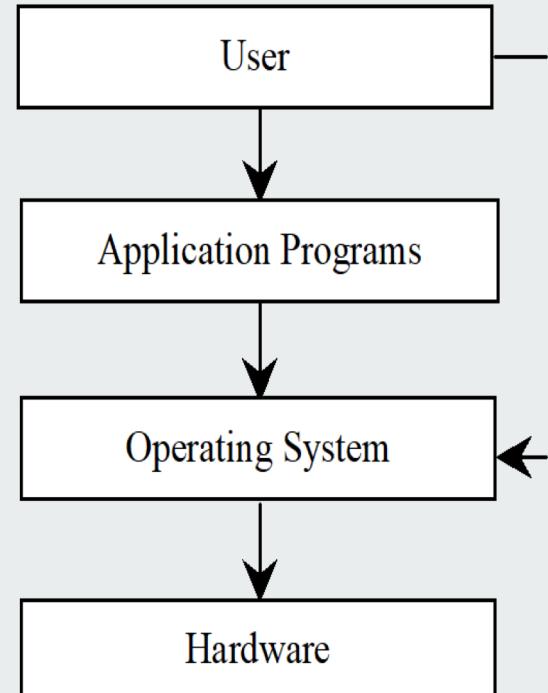
- These programs include word processors, spreadsheets, and games (e.g. Microsoft Office, Google Chrome, Skype, WhatsApp, etc.)

System programs take control of the computer, such as an Operating System (e.g.:Windows , Linux, MacOS, Android, etc.)

- Programs that support the execution and development of other programs
- Operating systems -> execute other programs
- Translation systems (Compilers and Interpreters)-> development of other programs

Operating Systems

- The operating system (OS) is a program that manages and controls a computer's activities.
- Windows is currently the most popular PC operating system. (**Which OS are you using? What about your phones?**)
- Application programs such as an Internet browser and a word processor cannot run without an operating system.





What do Programs do?

- Programs instruct the computer to solve some specific problems
- What Problem Can Be Solved By Computer? (**Discuss**)
 - When the solution can be produced by a set of step-by-step procedures or actions.
 - This step-by-step action is called an *algorithm*.
 - The algorithm will process some inputs and produce some output.
 - Algorithms are implemented using *programming languages*.



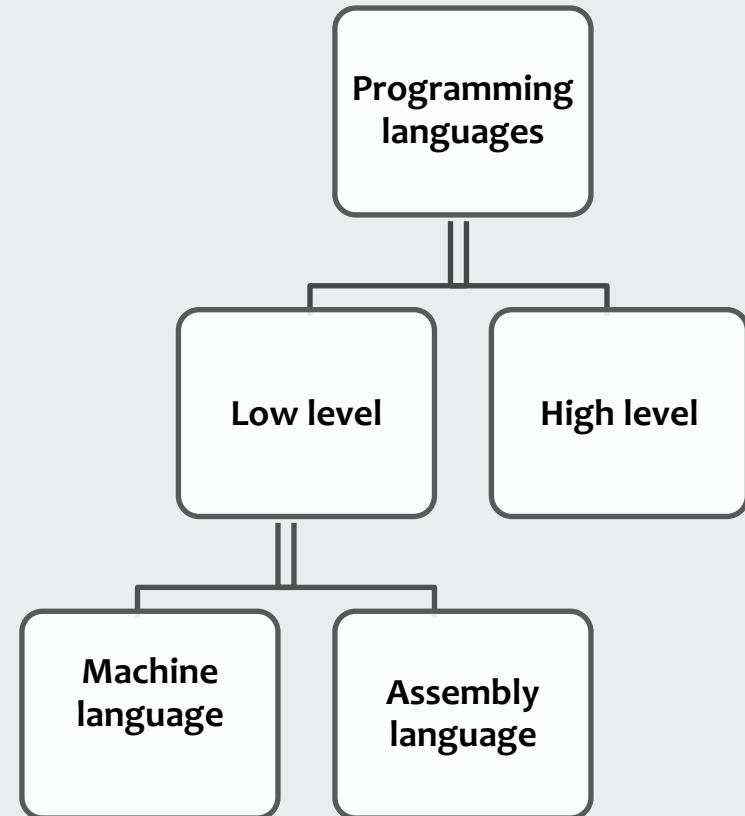
Programming Languages

- Natural language is fraught with ambiguity and imprecision
 - “I saw the man in the park with the telescope!”
- Computer scientists have gotten around this problem by designing notations for expressing computations in an exact, and unambiguous way.
- A set of special notations is called a **programming language**.
 - E.g. C, C++, Pascal, Visual Basic, Java, Python, ...
- Every structure in a programming language has:
 - A precise form, its **syntax**
 - A precise meaning, its **semantics**
- In fact, programmers often refer to their programs as **computer code**.
- An **algorithm** is a finite sequence of effective statements, that when applied to the problem, will solve it.
- The process of writing an algorithm in a programming language is called **coding**.

Types of Programming Languages

- Machine Language
- Assembly Language
- High-Level Language

*Are you familiar with any of
the above?*





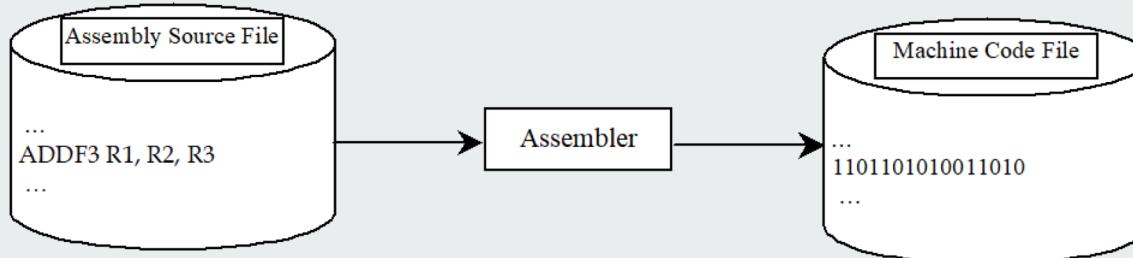
Programming Languages – Machine Language

- Machine language is a set of primitive instructions built into every computer.
- The instructions are in the form of binary code, so you have to enter binary codes for various instructions.
- Programming with native machine language is a tedious process.
- Moreover the programs are highly difficult to read and modify. For example, to add two numbers, you might write an instruction in binary like this:

```
1101101010011010
```

Programming Languages – Assembly Language

- Assembly languages were developed to make programming ‘easy’.
- Since the computer cannot understand assembly language, however, a program called assembler is used to convert assembly language programs into machine code.
- For example, to add two numbers, you might write an instruction in assembly code like this:





Programming Languages – High-Level Language

- The high-level languages are English-like and easy to learn and program.
- For example, the following is a high-level language statement that computes the area of a circle with radius 5: `area = 5 * 5 * 3.1415`
- **And of course, similar to Assembly Language, Computers do not understand High-Level Language.**



Popular High-Level Languages (1)

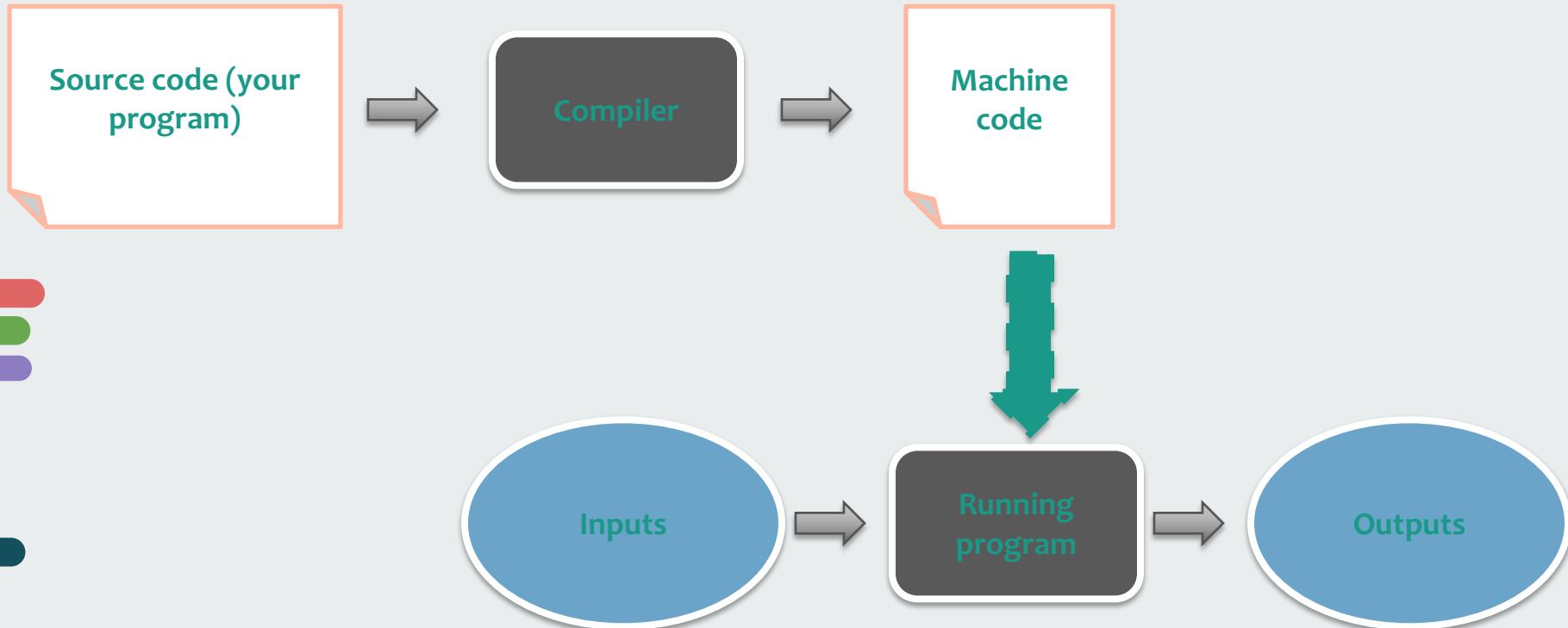
- **Python:** Known for its readability and simplicity, Python is widely used in web development, data science, artificial intelligence, and more.
- **JavaScript:** A core technology of the web, JavaScript is essential for front-end development and is increasingly used for back-end development with Node.js.
- **Java:** A versatile and platform-independent language used in web development, mobile apps (especially Android), and large-scale enterprise systems.
- **PHP:** A server-side scripting language used primarily in web development to create dynamic web pages.

High Level Language to Machine Language

Compiler

- A compiler is a complex computer program that takes another program written in a high-level language and translates it into an equivalent program in the machine language of some computer.
- The written high-level program is called source code.
- The resulting machine code is a program that the computer can directly execute.
- The black arrow in the diagram represents the execution of the machine code.

High Level Language to Machine Language



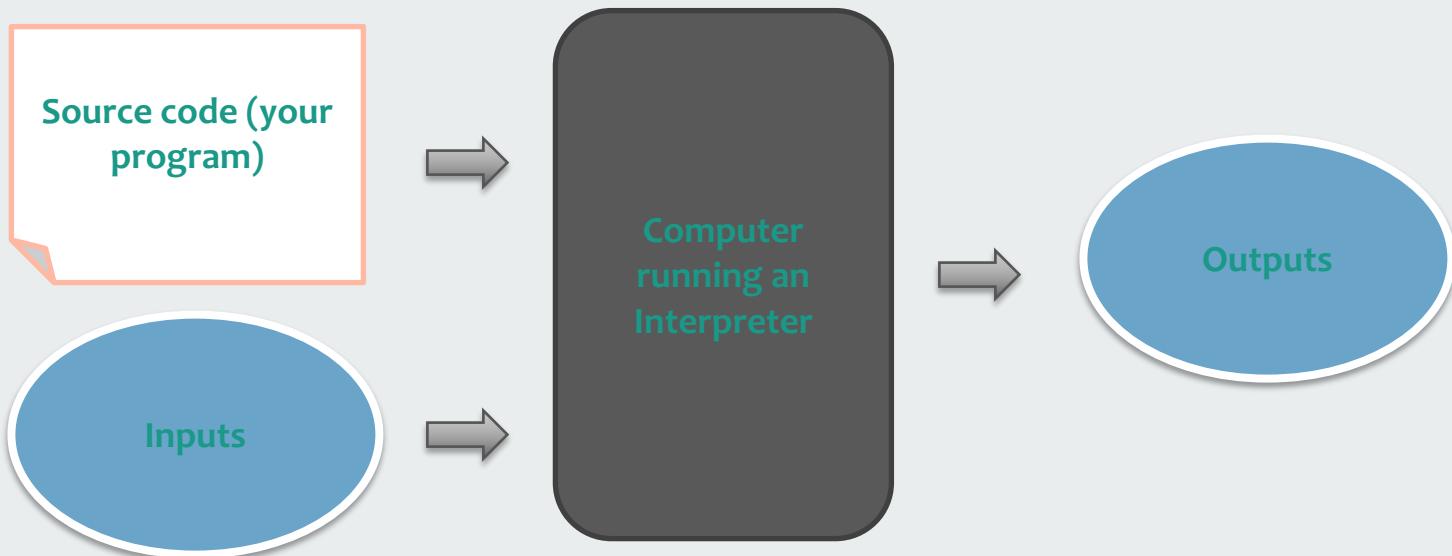


High Level Language to Machine Language

Interpreter

- An interpreter is a program that simulates a computer that understands a high-level language.
- Rather than translating the source program into a machine language equivalent, the interpreter analyzes and executes the source code instruction by instruction as necessary.

High Level Language to Machine Language





Writing Programs

- Learning to write programs requires two skills.
- 1. You need to develop a plan for solving a particular problem. This plan or algorithm is a sequence of steps that, when followed, will lead to a solution of the problem.
- 1. You need to use specific terminology and punctuation that can be understood by the machine; that is, you need to learn a programming language.



Planning your solution

- Identify a problem
- Break the problem into simpler problems
- Describe the steps to be taken to solve the simpler problems
 - Can start with flowcharts
 - Or Textual description (PseudoCodes)

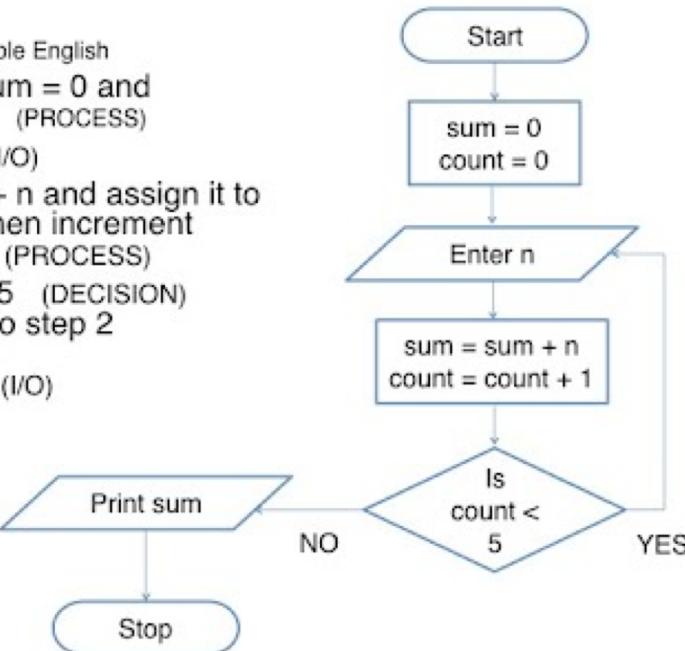
Pseudocode and Flowchart Example

Find the sum of 5 numbers

Algorithm in simple English

1. Initialize sum = 0 and count = 0 (PROCESS)
2. Enter n (I/O)
3. Find sum + n and assign it to sum and then increment count by 1 (PROCESS)
4. Is count < 5 (DECISION)
if YES go to step 2
else
Print sum (I/O)

Flowchart



<https://www.youtube.com/watch?app=desktop&v=vOEN65nm4YU>

ICDT 1201Y Computer Programming



Setting up

Depending on your Operating System, watch the corresponding YouTube Video and setup the environment for Python Development.

Windows <https://www.youtube.com/watch?v=ouANCqLtr98>

Linux <https://www.youtube.com/watch?v=6uswHeYXIBo>

MacOS <https://www.youtube.com/watch?v=NirAuEAbIvo>

Testing your setup

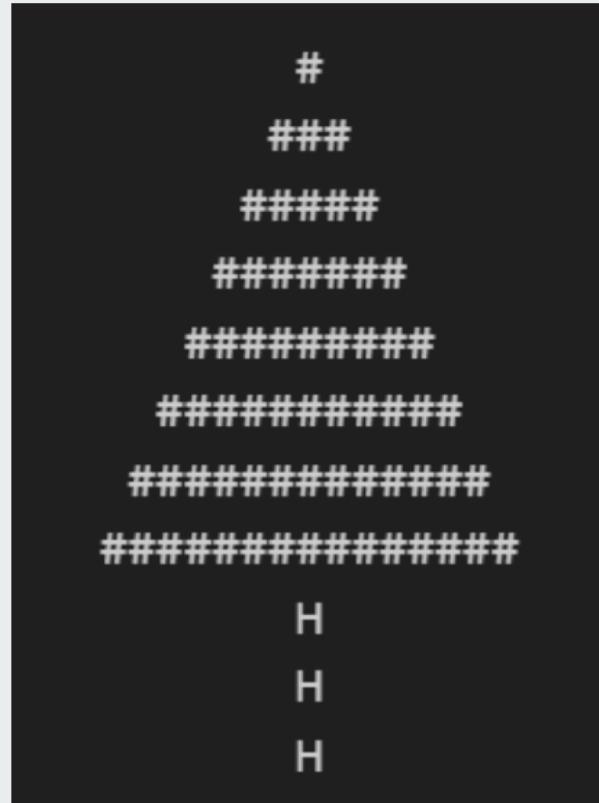
- The traditional way: You write a piece of code to display the text “Hello World” on screen.
- Create a file named *myfirst.py* and type the following codes:

```
print("Hello, my name is John Smith")
```

- Feel free to modify the text as you wish
- Execute your code by clicking on ▶

Lab Exercise

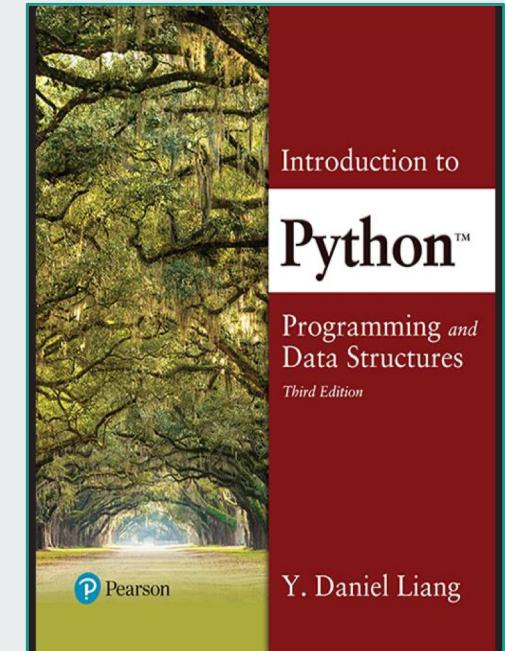
- Using your knowledge of the `print()` on the previous slide, recreate the drawing using python:

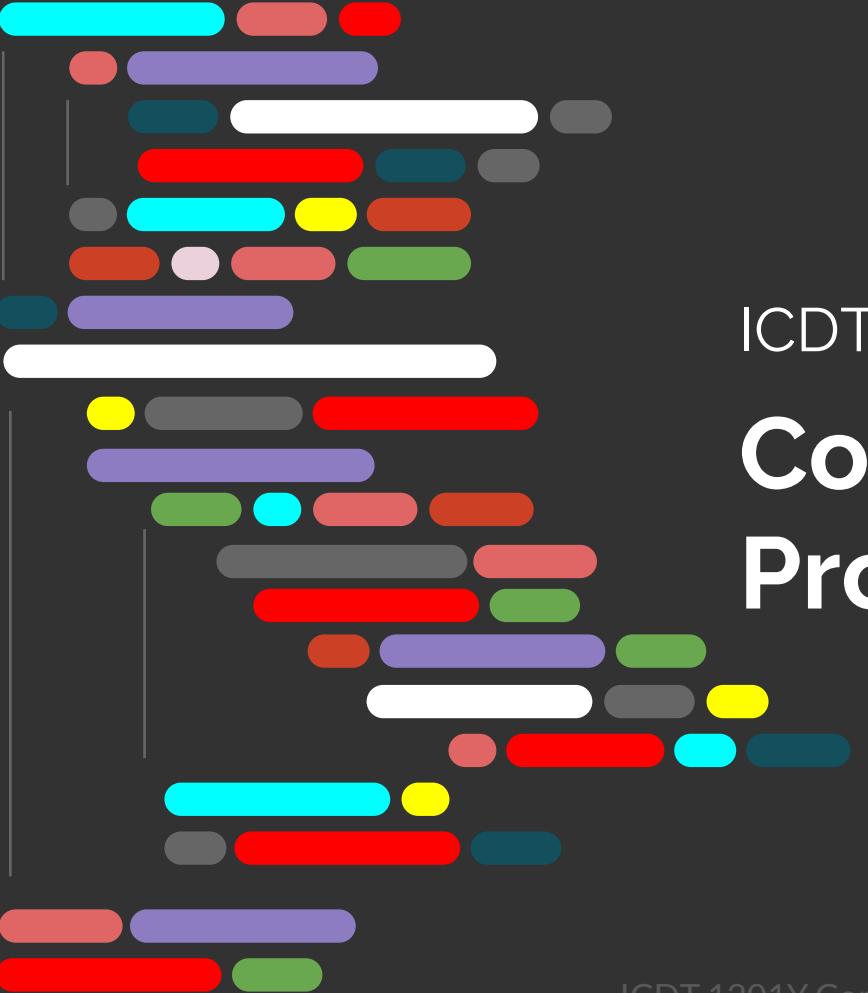


```
#  
###  
#####  
######  
#######  
########  
#######  
######  
H  
H  
H
```

Acknowledgements

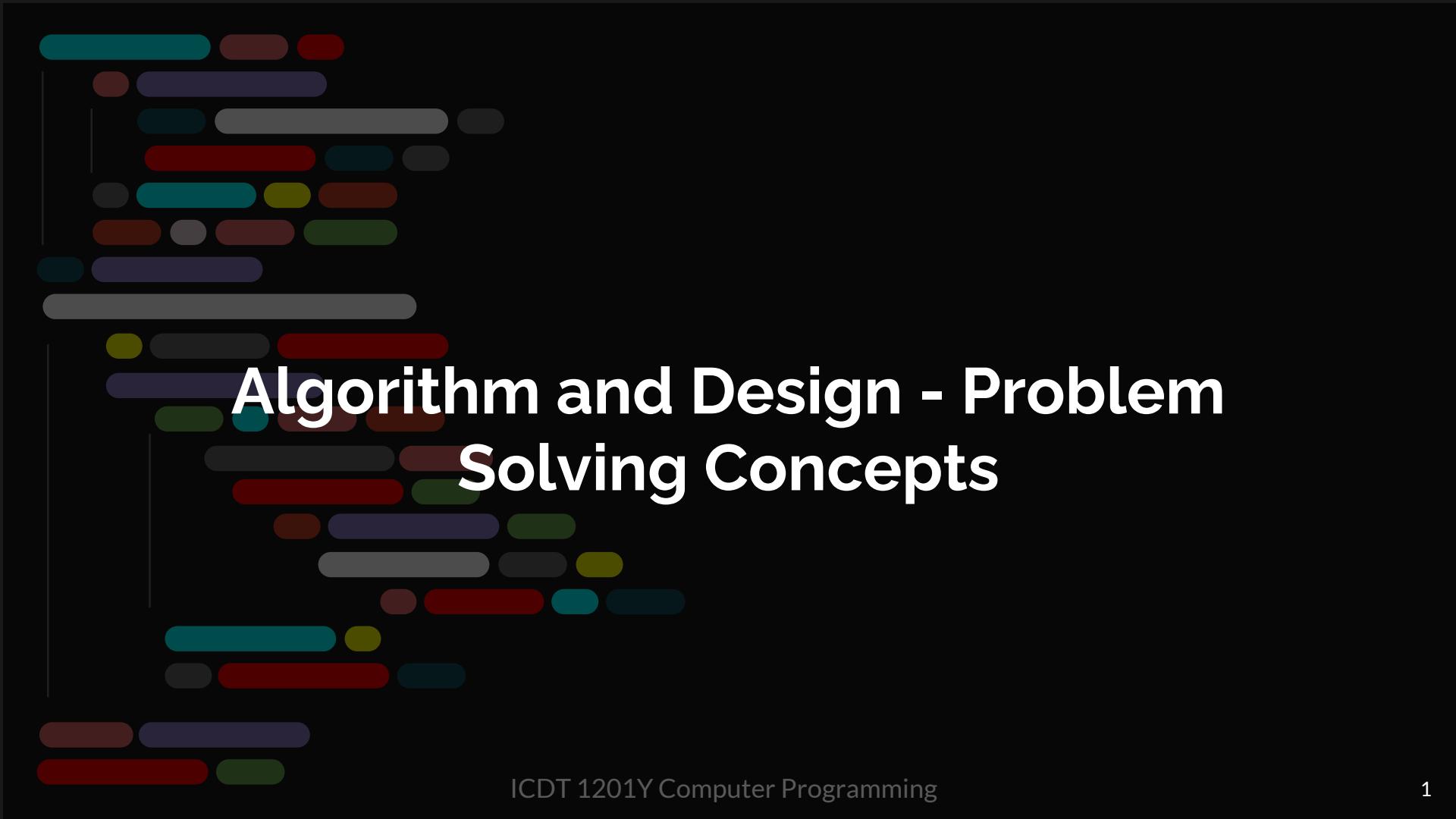
- Slides adapted from:
Dr P.Appavoo's slides &





ICDT 1201Y (3)

Computer Programming (week 2)



Algorithm and Design - Problem Solving Concepts



Sub-topics

- Problem Solving
- Abstraction
- Concept of Algorithms
- Program Design using Flowchart



Learning Outcomes & Assessment Criteria

Learning Outcomes:

- Principles of Problem-Solving:
 - Identify and explain problem-solving techniques and their importance in programming.
- Terminology:
 - Define and use terms like algorithm, abstraction, and pseudocode.
- Problem-Solving Tools:
 - Recognize and use tools such as flowcharts and pseudocode
- Flowchart Design:
 - Design and interpret flowcharts for program development.

Assessment Criteria:

- Problem-Solving Techniques:
 - Identify and explain various techniques.
 - Describe their importance in programming.
- Terminology:
 - Define and use key terms correctly.
- Algorithms:
 - Define algorithms and explain their characteristics.
- Flowcharts:
 - Design flowcharts and explain their symbols and use.



Problem Solving Concepts

- Importance of Problem Solving
 - Helps in breaking down complex problems into manageable parts
 - Fundamental for effective programming
- Techniques:
 - **Understanding the Problem** - Grasp the problem requirements and constraints.
 - **Devising a Plan** - Develop a step-by-step solution (algorithm).
 - **Executing the Plan** - Implement the solution in a programming language.
 - **Evaluating the Solution** - Test and refine the solution to ensure it solves the problem.



Main Phases of Problem Solving

Solving problem by computer undergo two phases:

- Phase 1:
 - Organizing the problem or pre-programming phase.
- Phase 2:
 - Programming phase.



Pre-Programming Phase

- This phase requires a number of steps:
 - Analyzing the problem.
 - Developing the Hierarchy Input Process Output (HIPO) chart or Interactivity Chart (IC) along with the Input-Process-Output (IPO)Chart (where applicable).
 - Writing the algorithms. *
 - Drawing the Program flowcharts.*



Complexities of Problems

- Problems' complexities vary
- It might not be necessary to understand all the details to solve a problem.
 - Example: Your car suffers from a flat tire; Is it necessary for you to know how the tire was manufactured, which raw materials were used, which country it was from etc? Or you just need to know the steps to change the tire? By overlooking the unnecessary details, we end up solving the problem faster...(imagine yourself watching hours of videos how tires are made).
 - This was an example of abstraction!



Abstraction

- Basic Definition
 - Simplifying complex systems by focusing on the main aspects and ignoring the details.
- Levels of Abstraction
 - High-level (e.g., overview of a system)
 - Mid-level (e.g., detailed design)
 - Low-level (e.g., specific implementation details)



Examples of Abstraction

- Example 1: Car as a system
 - High-level: Car as a means of transportation
 - Mid-level: Car's components (engine, wheels, etc.)
 - Low-level: Detailed design of the engine
- Example 2: Computer program
 - High-level: Software application
 - Mid-level: Modules and functions
 - Low-level: Code implementation



Importance of Abstraction

- Helps manage complexity
- Enhances code readability and maintainability
- Facilitates modular design



Analyzing the Problem (1)

- Analyzing The Problem
 - Understand and analyze the problem to determine whether it can be solved by a computer.
- Analyze the requirements of the problem.
- Identify the following:
 - Data requirement.
 - Processing requirement or procedures that will be needed to solve the problem.
 - The output.

Analyzing the Problem (2)

- All These requirements can be presented in a Problem Analysis Chart (PAC)

| Data | Processing | Output |
|---|--|----------------------|
| Given in the problem or provided by the User. | List of processing required or Procedures. | Output requirements. |

- **Further Reading:** In case the problem is big and complex, the processing can be divided into sub-tasks called modules; with each module accomplishing one function. This interaction between modules can be presented using a Hierarchy Input Process Output Chart (HIPO) or an Interactivity Chart (IC)



Concept of Algorithms

- An algorithm
 - A finite sequence of well-defined instructions to solve a specific problem.
- Characteristics
 - Clear and unambiguous
 - Finite number of steps
 - Well-defined inputs and outputs
 - Effective and efficient
- Few Examples
 - Common algorithms such as sorting and searching
 - Finding the maximum number in a list.



Algorithms Vs Programs

- Algorithm: Step-by-step solution
- Program: Implementation of the algorithm in a programming language



Algorithm Examples (1)

- Calculate the total (sum) of two numbers
 1. Input the two numbers
 2. Take first number and add to the second number
 3. Display the value obtained in 2.



Algorithm Examples (2)

- Finding the larger number between two numbers
 1. Input the two numbers
 2. Take first number and check if it is larger than the second number
 3. If first number is larger than second
 - 3.1 Display first number is larger
 4. Otherwise
 - 4.1 Display second number is larger



Algorithm Examples (3)

- Finding the largest number from a group of numbers
 1. Input all the numbers as a list
 2. Assume the first number in the list is the largest
 3. Compare the value of the largest with the next number from the list
 - 3.1 If the next number is larger, update largest value
 - 3.2 Continue comparison till end of the list is reached
 4. Display/Return the value of largest



Class Exercise

1. Write an algorithm to find the minimum number in a list.
2. Write an algorithm to find the average value of three numbers.
3. Write an algorithm to find the average value of a list of numbers.



Pseudocode

- Pseudocode is a way of writing out an algorithm in plain language using a structure that resembles programming code.
- Purpose: Helps in understanding and designing algorithms without worrying about syntax
- Characteristics:
 - Readable and easy to understand
 - Uses plain language
 - Structured similarly to Computer Programming Code
 - bridges the gap between the algorithm and the actual code implementation

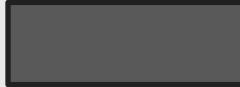
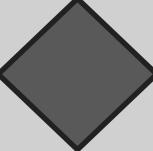


Algorithm Written as Pseudocode

1. Input all the numbers as a list
 2. Assume the first number in the list is the largest
 3. Compare the value of the largest with the next number from the list
 - 3.1 If the next number is larger, update largest value
 - 3.2 Continue comparison till end of the list is reached
 4. Display/Return the value of largest
-
1. Input: `list_of_numbers`
 2. `max = list_of_numbers[0]`
 3. For each number in `list_of_numbers`:
 - a. If `number > max`:
 - i. `max = number`
 4. Output: `max`

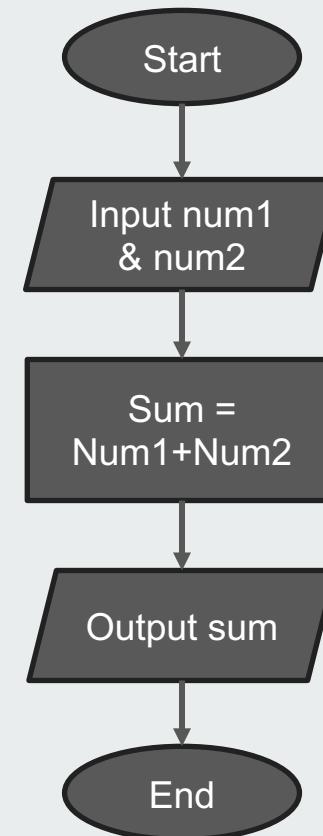
Program Design Using Flowchart

- Visual representations of the steps in an algorithm
- Symbols used in Flowcharts

| Symbol Name | Symbol | Representation |
|--------------------|---|-----------------|
| Oval |  | Start / End |
| Rectangle |  | Process |
| Diamond |  | Decision |
| Parallelogram |  | Input / Output |
| Directional Arrows |  | Flow of Control |

Flowchart Examples – Sum of two numbers

1. Start
2. Input two numbers
3. Calculate the sum
4. Output the sum
5. End





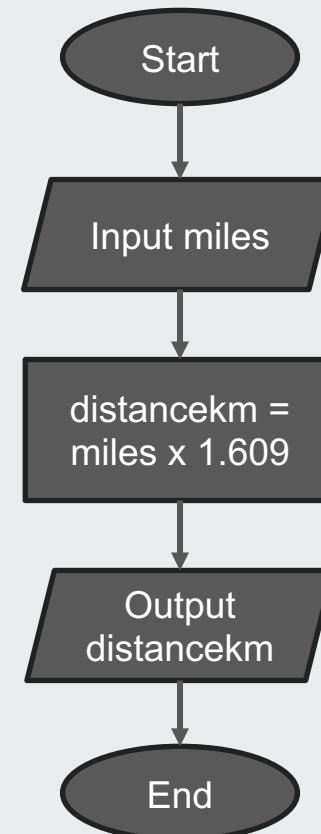
Flowchart Best Practices

- Use standard symbols
- Keep it simple and readable
- Clearly define inputs and outputs
- Ensure logical flow and correctness

Flowchart Examples – Distance Conversion

1. Write the pseudocode and draw the Flow Chart and to convert the distance in miles to kilometers where 1.609 kilometers is equivalent to 1 mile.

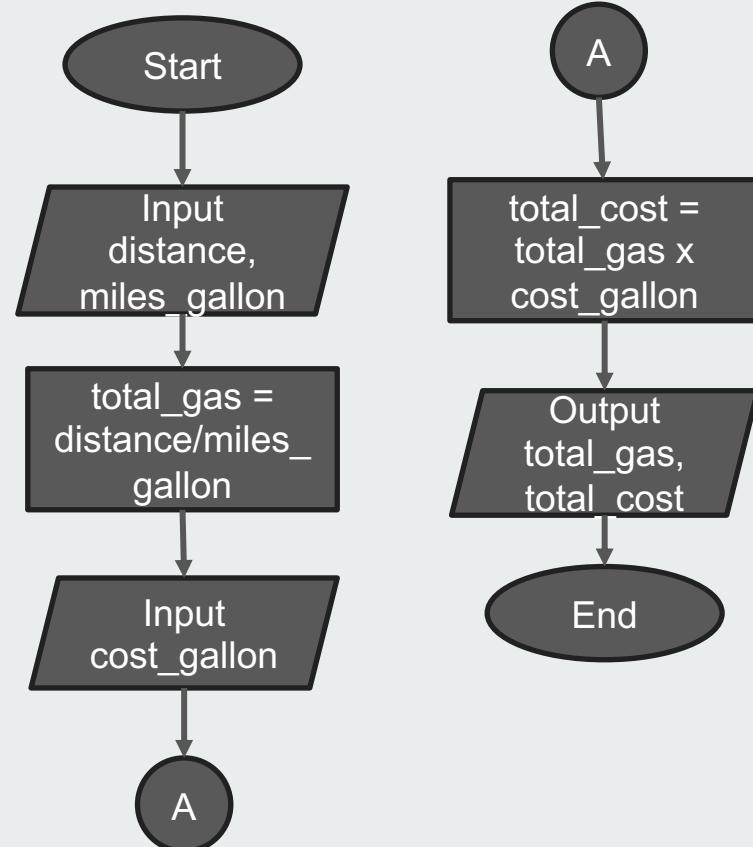
1. Start
2. Input miles
3. $\text{distancekm} = \text{miles} \times 1.609$
4. Output distancekm
5. End



Flowchart Examples – Gas & Cost

Write the pseudocode and draw the Flow Chart that asks a user to enter the distance of a trip in miles, the miles per gallon estimate for the user's car, and the average cost of a gallon of gas. Calculate and display the number of gallons of gas needed and the estimated cost of the trip.

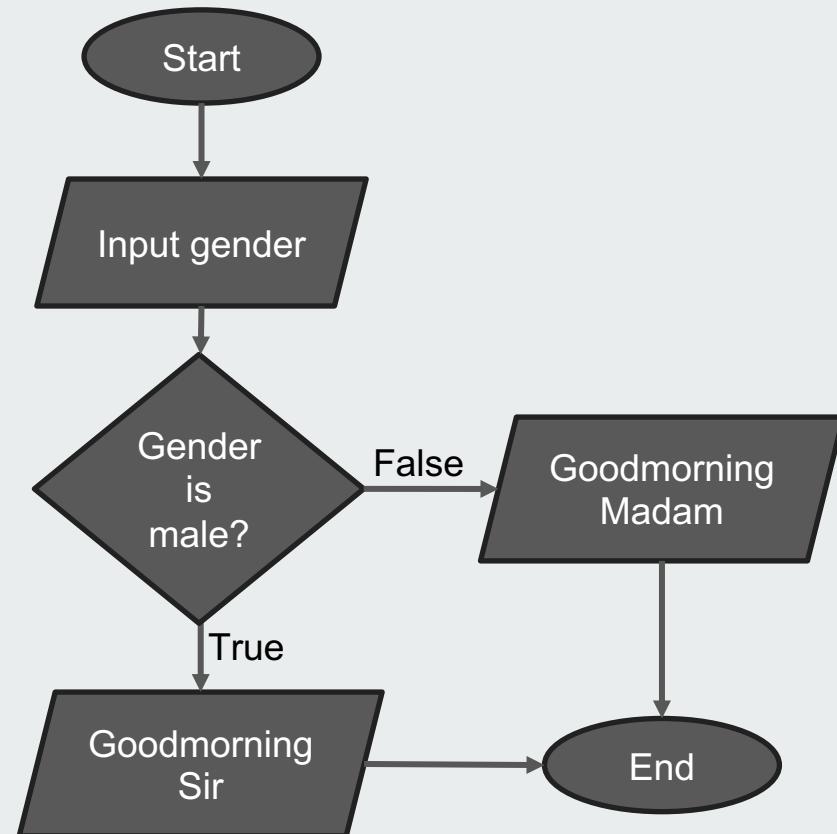
1. Start
2. Input distance and miles_gallon
3. total_gas = distance/miles_gallon
4. Input cost_gallon
5. total_cost = total_gas x cost_gallon
6. Output total_gas, total_cost
7. End



Flowchart Examples – Decisions

Write the pseudocode and draw the Flow Chart that asks a user whether he/she identifies as male or female and greet them accordingly with Goodmorning Sir or Goodmorning Madam.

1. Start
2. Input gender
3. If gender is male
 - 3.1 Output "Goodmorning Sir"
 - 3.2 Output "Goodmorning Madam"
- else
4. End

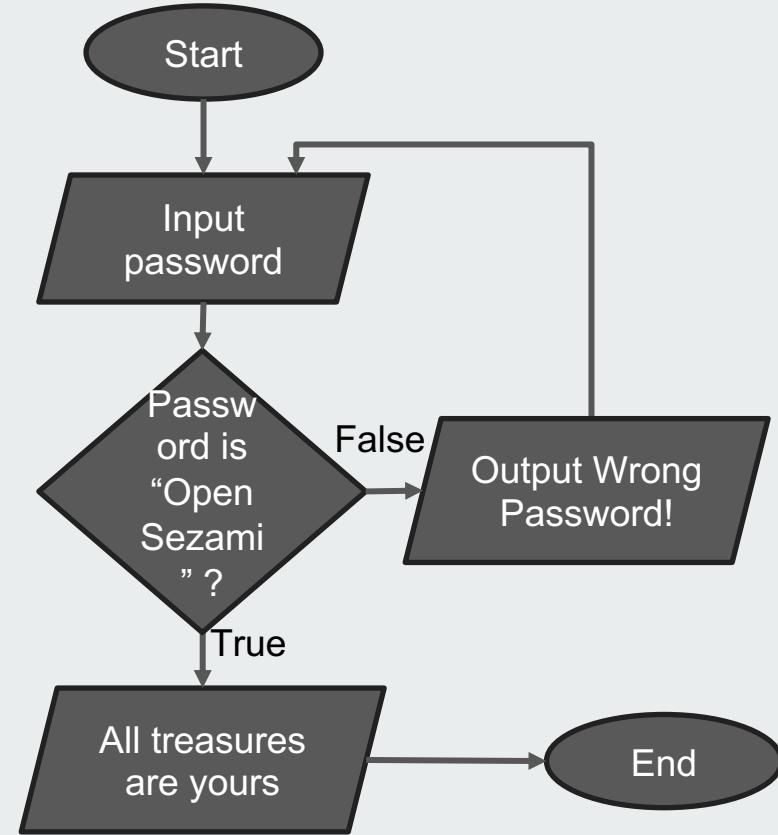


Flowchart Examples – Repetitions / Iterations (1)

Write the pseudocode and draw the Flow Chart that asks a user the password. As long as the wrong password is provided, inform the user that the password is wrong and ask the user for the password again. The password is “OpenSezami”. If successful, inform the user that all the treasures are his/hers!

1. Start
2. Input password
3. If password is “OpenSezami”
 - 3.1 Output “All treasures are yours!”
- else
 - 3.2 Output “Wrong Password”
 - 3.3 GoTo Line 2
4. End

What happens if the right password is never entered?



Flowchart Examples – Repetitions / Iterations (2)

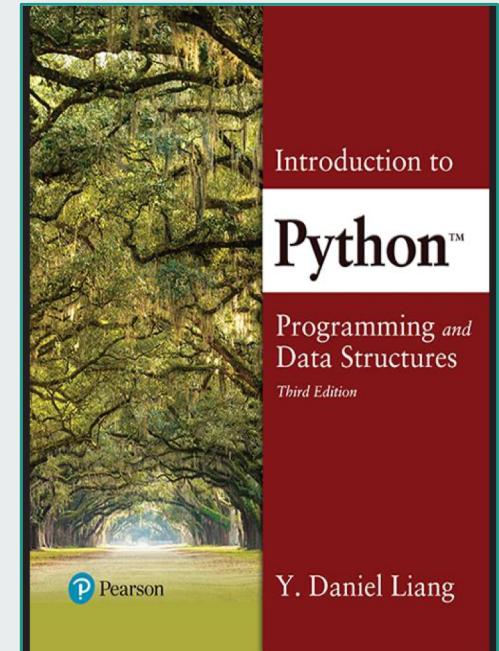
Write the pseudocode and draw the Flow Chart that ask the user for the number of students in the class and then for each student, ask for his/her mark. Finally, calculate and display the average mark of the class.

1. Start
2. Input num_students
3. total_marks=0
4. If num_students not 0
 - 4.1. input mark
 - 4.2. total_marks=total_marks+mark
 - 4.3. num_students=num_students + 1
 - 4.4. GoTo Line 4
5. average= total_marks/num_students
6. Output average
7. End

Draw the Flow Chart!

Acknowledgements

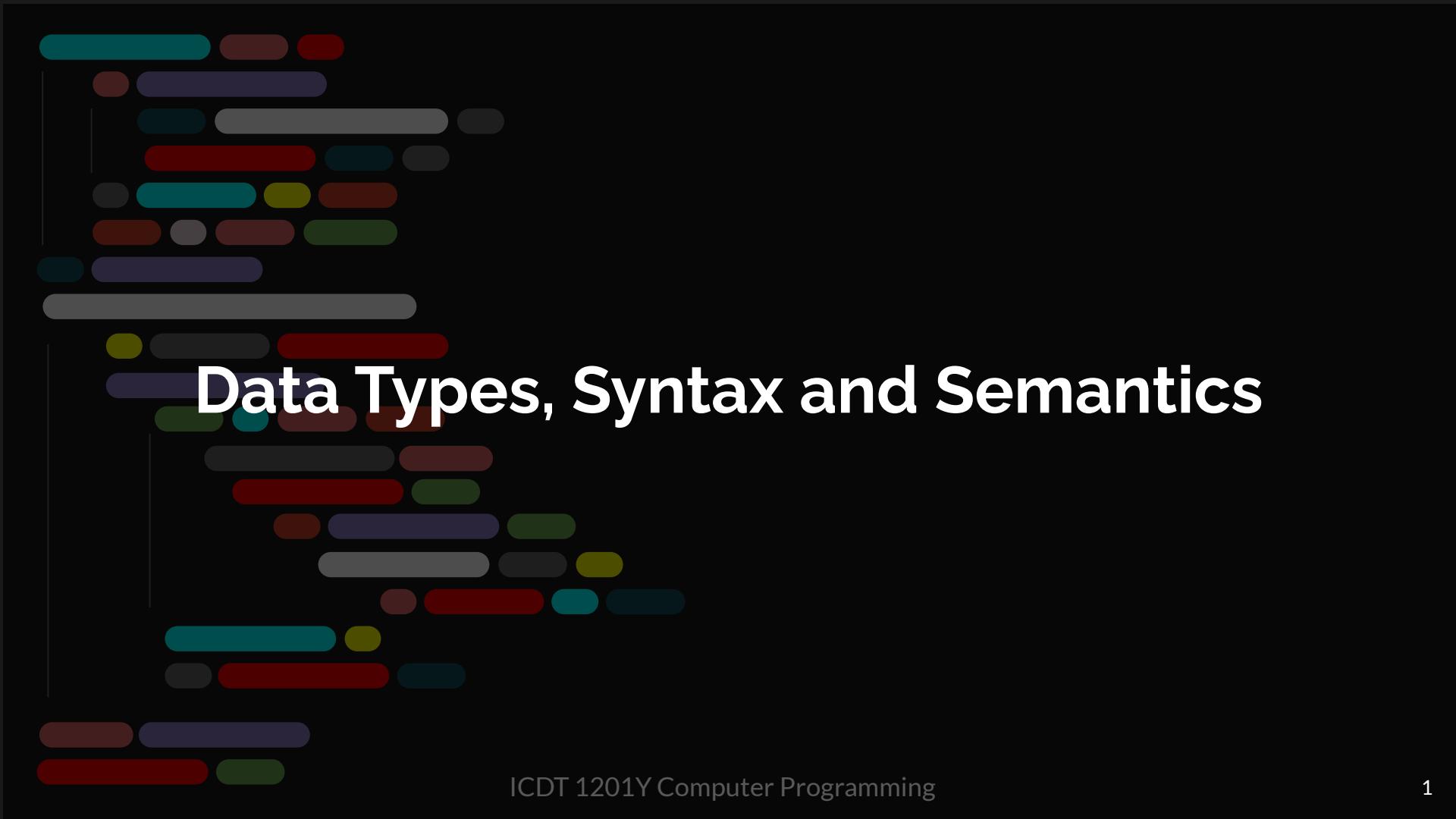
- Slides adapted from:
- & Dr P.Appavoo's slides





ICDT 1201Y (3)

Computer Programming (week 2)



Data Types, Syntax and Semantics



Sub-topics

- Syntax
- Semantics
- Variables
- Arithmetic Operations
- Expressions and Assignments



Learning Outcomes & Assessment Criteria

Learning Outcomes:

- Demonstrate an Understanding of Data Types, Variables, Syntax, and Semantics:
 - Explain the importance of data types in programming.
 - Identify common data types (integer, float, string, boolean) and their uses.
 - Understand the concept of variables and their role in storing data.
 - Differentiate between syntax (rules for writing code) and semantics (meaning of the code).

- Define the Key Components of a Statement in a Program:
 - Recognize the basic structure of a program

Assessment Criteria:

- Understand the concepts of literals, identifiers, and mathematical operations.
- Understand the difference between a valid and an invalid identifier.
- Distinguish between different types of data.
- Use variables and write simple expressions.
- Understand the construct of simple statements such as assignments and *function calls*.



Introduction to Python

- Python is a general-purpose programming language that can be used effectively to build almost any kind of program that does not need direct access to the computer's hardware.
- Easy to learn, powerful programming language
- It has efficient data structures
- **It is “interpreted”!**
- Other characteristics: interactive, portable, extensible, scalable, has a large standard library of built-in functions



Python Programs

- A Python program, sometimes called a script, is a sequence of definitions and commands.
- A **program** is a sequence of definitions and commands
 - definitions evaluated
 - commands executed by Python interpreter in a shell
- **commands (statements)** instruct interpreter to do something
 - can be typed directly in a shell or stored in a file that is read into the shell and evaluated



Data Types in Programming

- What are data types?
 - Kinds of data that a *variable* can hold.
 - Common Data types in Python (primitive):
 - Integer (int)- Whole numbers[8,-4,0]
 - Float (float): Decimal numbers [3.142, -34.5]
 - String (str): Sequence of characters/ Text ["Hello", "24", "Bonjour Moris"]
 - Boolean (bool): True or False values
 - There are more COMPLEX Computer Programming data/composite) types... which
- Data Type Analogy in Real-life
 - Liquids....stored in bottles
 - Large Veggies - Potatoes & Onions....stored in 'net-bag'? [forgive us if you do not believe these are veggies]
- **Liquids and Large Veggies** would be equivalent to two different *data types*
- **The Bottles and 'Net-Bags'** would be the *variables* (*containers!*)

Experimenting with Data Types

- In Python, a special code (function) enable us to find out the data type of some values.
- You already know the `print()` function
- Now we will use the `type()` function [inside on the `print()`]
- Python automatically assigns the data type based on the value provided.

| Python Code | Output |
|--|------------------------------------|
| <code>mark = 10 print(type(mark))</code> | <code><class 'int'></code> |
| <code>pi = 3.142 print(type(pi))</code> | <code><class 'float'></code> |
| <code>greeting = "Hello" print(type(greeting))</code> | <code><class 'str'></code> |
| <code>is_married = True print(type(is_married))</code> | <code><class 'bool'></code> |

Take note of mark, pi, greeting, is_married...these are names/identifiers for variables



Type Conversions

- Sometimes values of one data type need to be converted to another
 - The `float()` function converts an `int` to a `float`
 - The `int()` function converts numbers into `ints`
 - The `round()` function rounds a `float` off to the nearest whole number and returns a `float`
 - The `str()` ?

Try:

- `int(4.5)`
- `int(3.9)`
- `float(int(3.3))`
- `round(-3.5)`



Understanding Syntax

- Syntax refers to the rules that define the structure of a programming language.
- Correct syntax is crucial for the program to compile and run.
- For example:
 - Code blocks/groups of code are defined with indentation
 - A statement usually end with a newline (not a full-stop or semi-column)
 - **Comments start with a '#'** - What is a comment and its role in programming?

1. I ate pizza. Tomorrow I will eat cake.
2. eat pizza i cake will eat tomorrow I

The examples above demonstrate the importance of syntax/rules... Even in normal English we need to respect it if we want our communication to be understandable.

Comments in Python

- A good way to enhance the readability of code is to add comments.
 - Text following the symbol # is not interpreted by Python. For example, one might write:

```
1 pi=3.142
2 side = 1 #length of sides of a unit square
3 radius = 1 #radius of a unit circle
4 #subtract area of unit circle from area of unit square
5 areaC = pi*radius**2
6 areaS = side*side
7 difference = areaS - areaC
```

Investigate multiple-line commenting in Python.



Understanding Semantics

- Semantics refers to the meaning of the code or what the code does when executed.
- **Syntax** is about **structure**, while **semantics** is about **meaning**.

Understanding Literals and Identifiers

- **Literals** are fixed values assigned to variables (e.g., numbers, strings).

```
1 gavin_age = 64 # 64 is a literal
2 full_name = "Gavin Hacker" # "Gavin Hacker" is a literal
```

- **Identifiers** are names given to variables, functions, etc.
 - Must start with a letter or underscore (_).
 - Can contain letters, digits, and underscores.
 - Cannot be a reserved keyword.



Valid Identifier in Python

- Rules for Valid Identifiers:
 - Must start with a letter or underscore.
 - Can contain letters, digits, and underscores.
 - Cannot be a reserved keyword (e.g., def, class, if).

Examples: Valid Identifiers

name
my_name
name2
_age
myName
NAME

Invalid Identifiers

1stname
middle-name
def
My name
allName\$

Variables in Programming

- Variables are used to store data that can be manipulated and retrieved.
- To declare a variable, We give it a unique name (identifier) and use

```
name = "Gavin"  
age = 49  
height = 5.9  
is_lecturer = True
```

- Remember our liquids, potatoes and onions?
- We need appropriate containers to keep them
- The contents of the containers can be removed/replaced/changed
- **We give names to our containers: Pot Disel / Sac Winners Nouvo... why?**



Storing Data Values in Variables

- To store data values, programs make use of variables
 - Variables have a type but are never declared in Python.
 - They are instantiated when they are assigned for the first time.
 - For Example:

```
x =5  
y=10  
z=x+5
```

Using Variables

```
1 # Declare variables  
2 name = "Gavin"  
3 age = 49  
4  
5 # Use variables  
6 print(name) # Output: Gavin  
7 print(age) # Output: 49  
8  
9 # Update variables  
10 age = 50 # we forgot Gavin's birthday...he is now 50  
11 print(age) # Output: 50  
12
```

Variables can be updated by assigning a new value.

The value of a variable can be printed using the `print()` function.

Exercise 1. on Variables

- What would be the expected output if the codes below are executed?

```
1     a=5  
2     b=10  
3     c=a+5  
4     print(a)  
5     print(B)  
6     print(c)
```

- Remember....Pythons are sensitive snakes...b & B are different...case-sensitive they call it.



Displaying Variables Names and Values

Consider

`x = 5`

What is the difference between the following two statements:

`print("x")`

`print(x)`

How do we display `x = 5` ?



Assigning String Values to Variables

- In the previous slides, numerical values had been assigned to variables.
- To assign non-numerical (string) values, the values should be between single or double quotes, eg.:

```
lecturer = "Gavin Hacker"
```

```
course= "BSc(hons) Hacking Methodologies"
```

- If we run the statement
`print(lecturer, course)`

- We will see the output

```
Gavin Hacker BSc(hons) Hacking Methodologies
```



Exercise 2. on String and Variables

- Write a program that assigns the value “Pizza” to the variable **menu** and “Cheese and Capsicum” to the variable **topping** and makes use of the variables **menu** and **topping** to display the following output:

The menu Pizza has topping Cheese and Capsicum

Arithmetic Operations

- Arithmetic operations involve basic mathematical operations.
- Common Operators:
 - Addition (+): Adds two numbers.
 - Subtraction (-): Subtracts one number from another.
 - Multiplication (*): Multiplies two numbers.
 - Division (/): Divides one number by another. (Result is always a float)
 - Modulus (%): Returns the remainder of a division.

```
1 # Addition  
2 week = 5 + 2  
3 print(week) # Output: 7  
4  
5 # Subtraction  
6 weekdays = 7 - 2  
7 print(weekdays) # Output: 5  
8  
9 # Multiplication  
10 month = 4 * 7  
11 print(month) # Output: 28  
12  
13 # Division  
14 weeks = 31 / 7  
15 print(weeks) # Output: 4.428571428571429  
  
quiz = 9 % 2  
print(quiz)
```

Expressions and Assignments

- An Expression is a combination of variables, literals, and operators that produces a value.

- An Assignment is the operation during which the result of an expression is assigned to a variable using the = operator.

```
1 # Declare variables
2 start_age = 24
3 expected_retirement = 65
4
5 # Expression and assignment
6 years_of_service = expected_retirement - start_age
7 print(years_of_service) # Output: 41
8
9 # Update variable with expression
10 start_age = start_age + 3
11 print(start_age) # Output: 27
```

Variables can store the result of expressions.
Variables can be updated with new expressions.

Complex Expressions

- More complex and interesting expressions can be constructed, by combining simple expressions, through the use of operators.

```
a=2  
b=3  
print(a*a + b*b)
```



Precedence in Arithmetic Operations

- The arithmetic operators have the usual mathematical precedence.
- For example, `*` binds more tightly than `+` so the expression $x+y^2$ is evaluated by first multiplying y by 2 and then adding the result to x . The order of evaluation can be changed by using parentheses to group subexpressions, e.g., $(x+y)^2$ first adds x and y , and then multiplies the result by 2 .

Additional Practice Exercise 3

Dry-run and write down the expected outcome for each of the command below. Then in the Python shell, the commands, pressing ‘Enter’ after each, to verify your answer.

```
print ("Hello, World")
print (2 + 3)
print("2 + 3 =", 2 + 3)
print ("Hi! How are you?")
print ("I am now in the Computer Lab at UoM.")
print( "Hi! How are\n you?")
print ("I am now \t in the Computer Lab at UoM.")
x = 2
print( x)

x = 2
y = 3
print( x + y)
```

Additional Practice Exercise 4

Write down the expected output (afterwards you may verify on your machines)

```
1 num1 = 15
2 num2 = 25
3 num3 = 5
4 sum = num1 + num2 + num3
5 print("The sum is:" + sum)
```

String concatenation is when two or more values of type String are ‘joined’ or ‘concatenated’. The ‘+’ operator is used for this. However, Python does not allow Strings to be joined with integers directly. Investigate possible techniques for addressing this issue.

Additional Practice Exercise 5

Write a Python program that declares a string variable, assigns a value to it, and prints the length of the string.

One ‘easy’ technique would be for you to manually count the number of characters; Will this technique still work if you were provided with a huge paragraph of text?

Just like the `print()` or the `type()` functions that you are now already familiar with, there is another in-built function called `len()`.

```
1  text = "Hello everybody, my name is Gavin, also known as Gavin Hacker"  
2  length = len(text)  
3  print("The length of the string is:", length)
```

Additional Practice Exercise 6

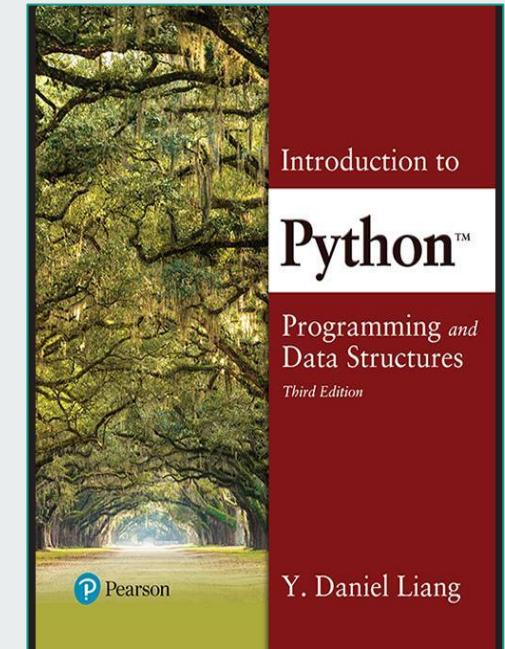
Simultaneous Assignments: Investigate how to use Simultaneous Assignments in Python through the general format:

<var>, <var>, ..., <var> = <expr>, <expr>, ..., <expr>

And look into how a swap can be performed through Simultaneous Assignments

Acknowledgements

- Slides adapted from:
- & Dr P.Appavoo's and Dr Baichoo's slides



Disclaimer: Gavin is a fictitious character used as an example....



ICDT 1201Y_Computer Programming

Control Structures (Week 4)



Learning Objectives

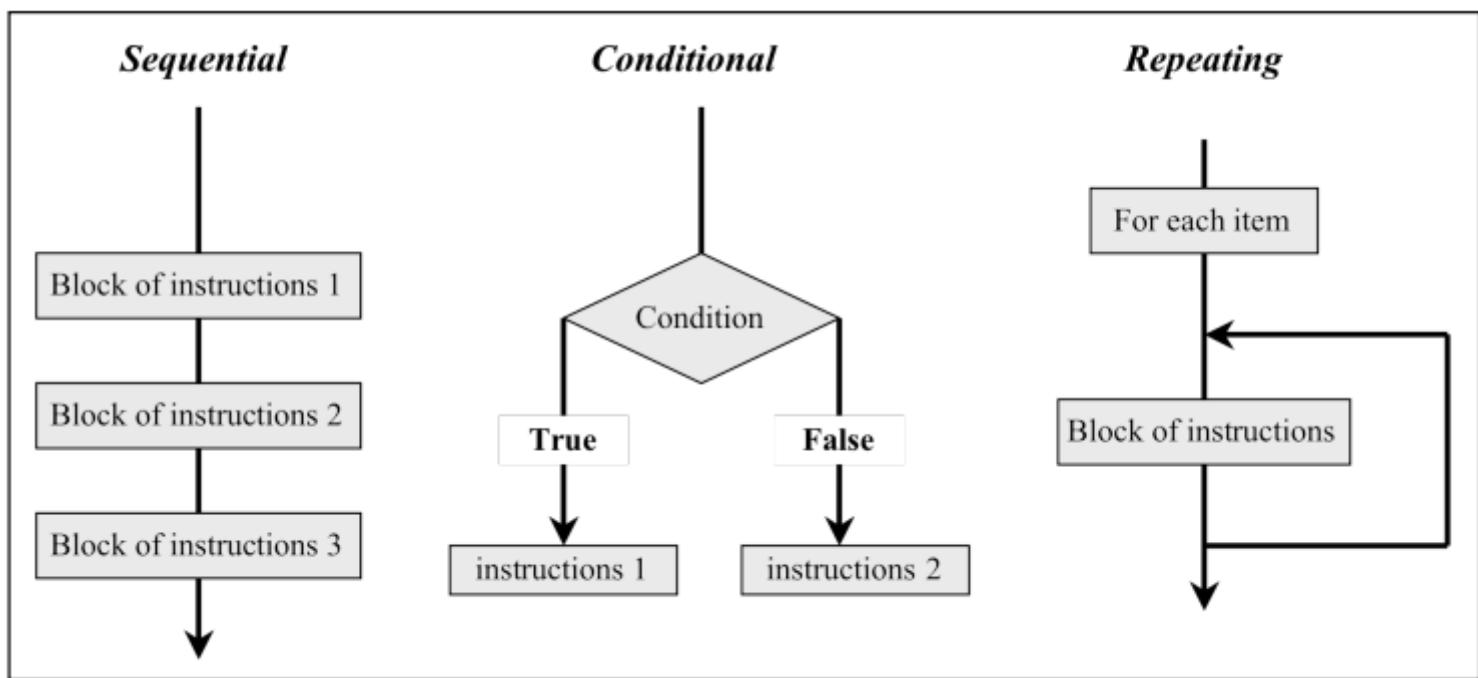
- Understand the objectives of decision in programs
- Illustrate a given problem using flowchart and pseudocodes
- Write programs using if... elif ...else statement in Python



Control Structures

- The three main types of flow in a computer program:
 - sequential, in which instructions are executed successively,
 - conditional, in which the blocks “instructions 1” and “instructions 2” are executed if the Condition is True or False, respectively, and
 - repeating, in which instructions are repeated over a whole list.

Control Structures



1. Decision Structure



Decision making statements

- Python programming language provides the following types of decision making statements.
 - if statements
 - if....else statements
 - if..elif..else statements
 - nested if statements



Simple Decisions

- Sequencing is not sufficient to solve every programming problem.
- To alter the sequence of flow in a program, a decision structure/control structure can be used to suit the needs of a particular situation.
- Conditional constructs are used to incorporate decision making into programs.
- The result of this decision making determines the sequence in which a program will execute instructions.

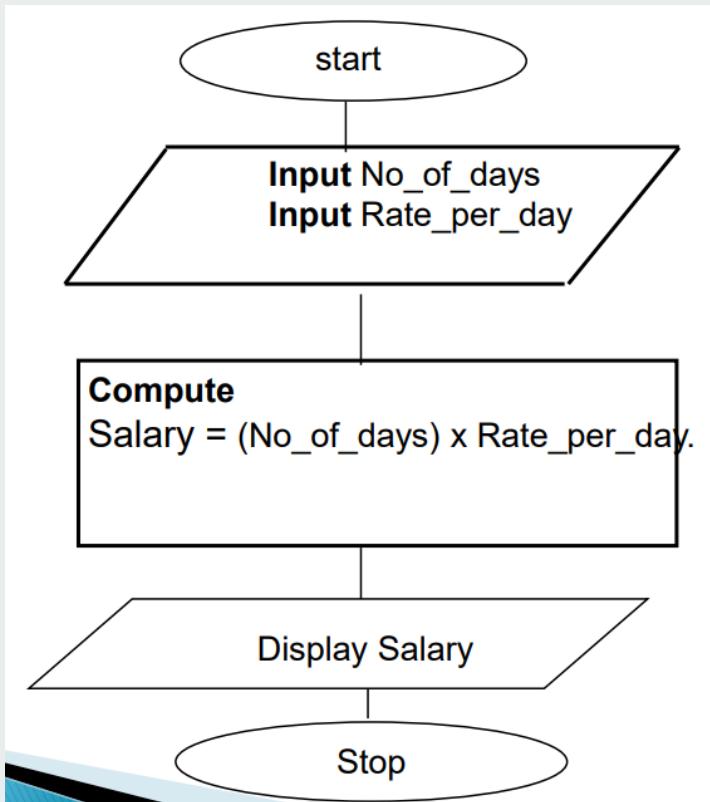


Decision in Programs

Consider the following Problem:

- Draw the flowchart and write the program to calculate the salary of a person at the end of a month. The program should take as inputs the no_of_days and the rate_per_day.
- Does this problem involve any decision?

Flowchart for the program



The program in Python

```
# Input num of days and rate per day
no_of_days = int(input("Enter the number of days worked: "))
rate_per_day = float(input("Enter the rate per day: "))

# Calculate the salary
total_salary = no_of_days* rate_per_day

# Display the result
print(f"The salary at the end of the month is:
{total_salary:.2f}")
```

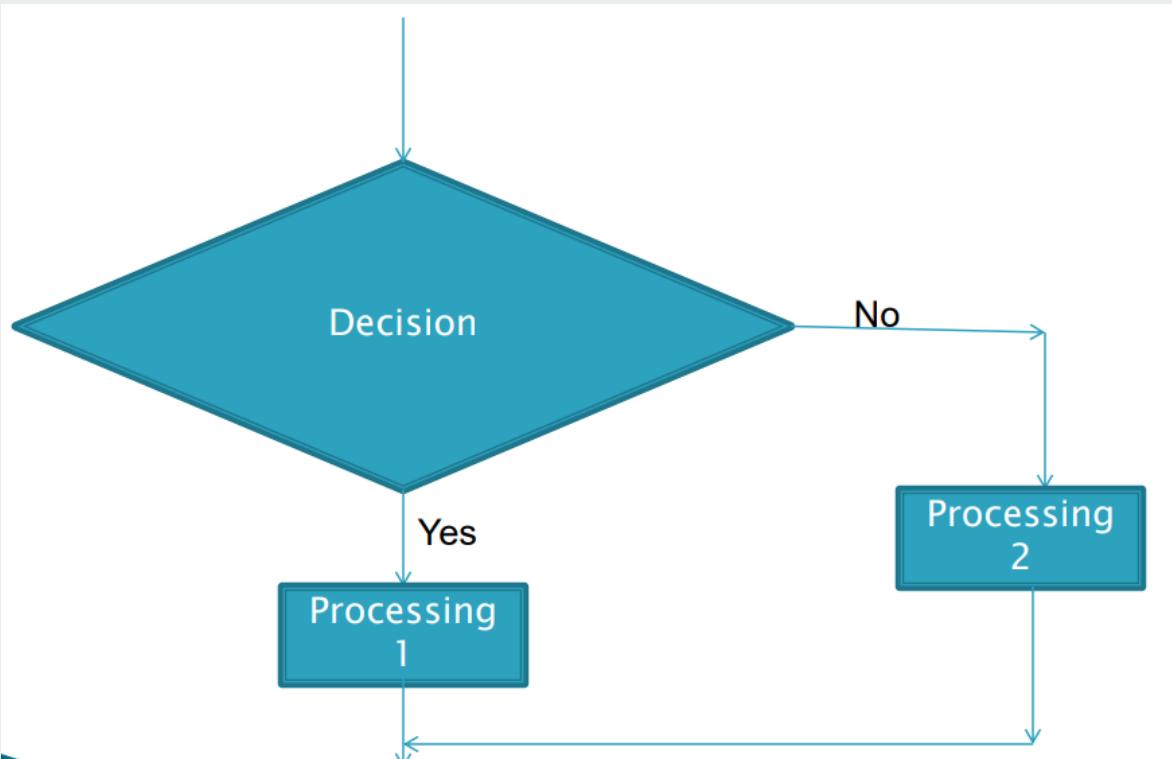


Problems involving decisions

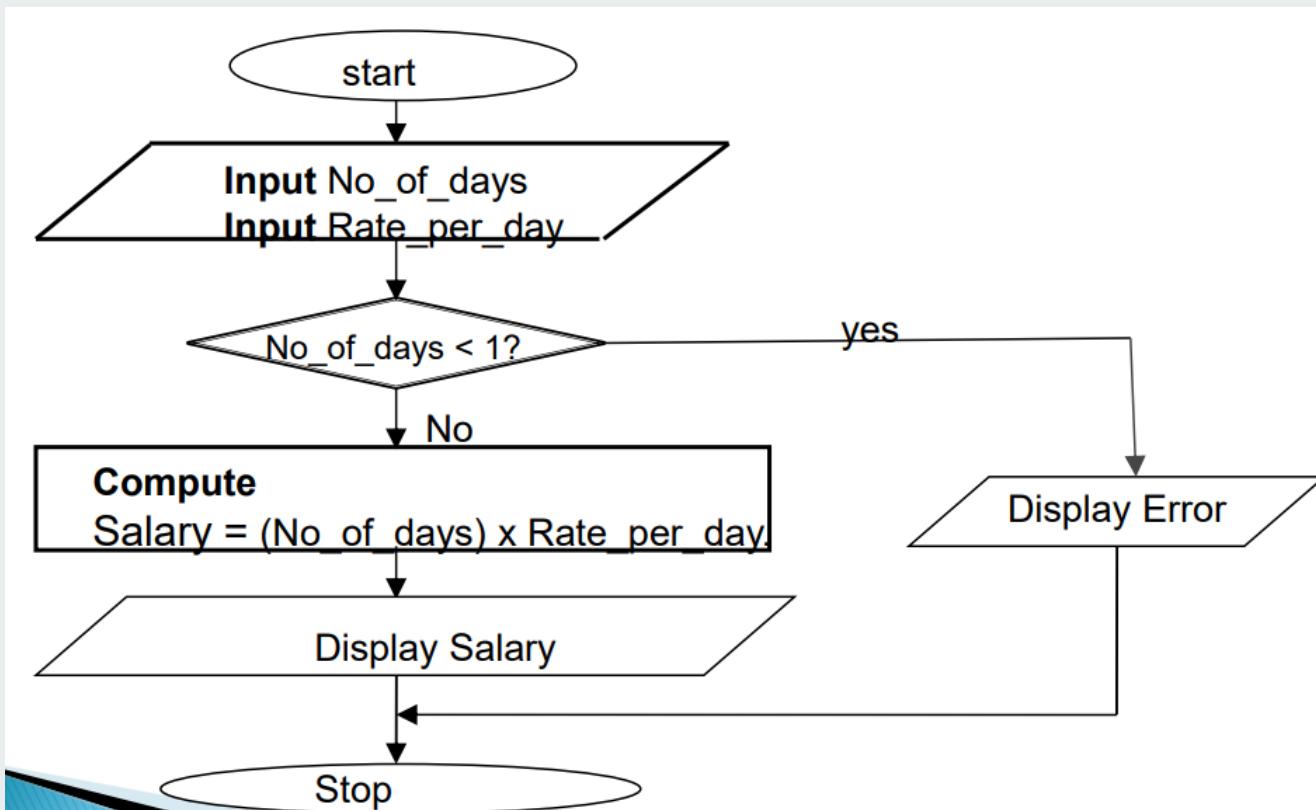
Consider the problem below:

- Draw the flowchart and write the program to calculate the salary of a person. You know the `no_of_days` and the `rate_per_day`. If the `no_of_days` is < 1 , then your program should display an error.
- Does this problem involve any decision now?

The Decision symbol

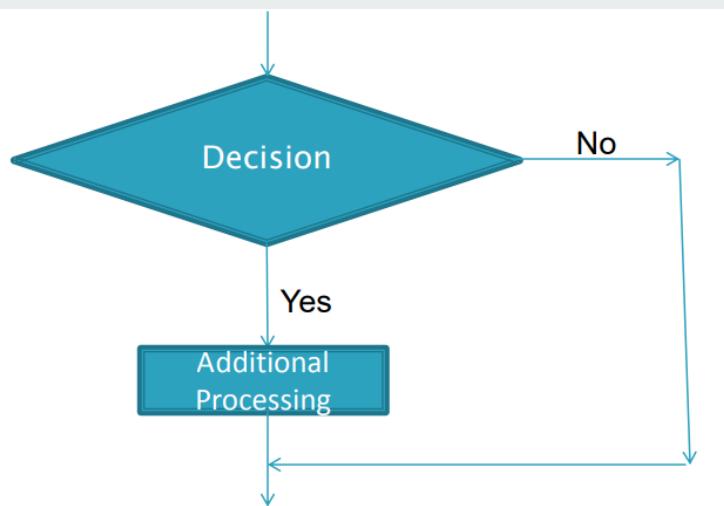


Flow chart including the decision part



Simple decision

- Simple Decisions are cases where there is a path to be followed (processing to be done) if a condition is true. If the condition evaluates to false, the path is simply not followed.

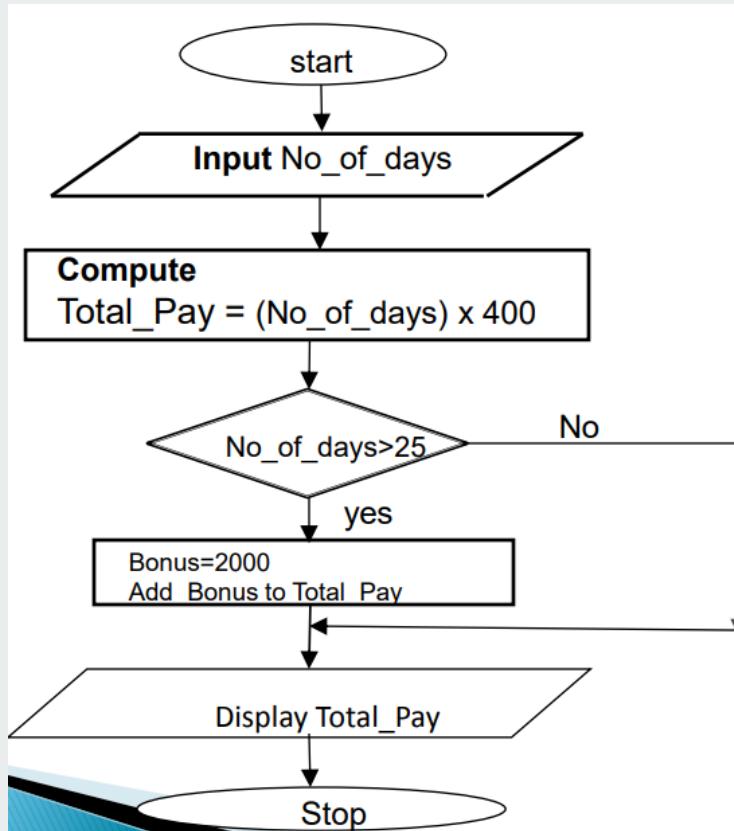


Simple Decisions

Example 5.1 :

- Consider a person being paid on a daily rate of Rs. 400. At the end of the month, he obtains his salary. Additionally, he obtains a bonus of Rs 2000 if he has worked for more than 25 days. We need a program to calculate his total pay at the end of a month.
- Inputs: No. of days.
- Outputs: Total Pay.
- Processing:
 - General : $\text{Total Pay} = \text{No.of days} * 400$
 - Conditional: $\text{Total Pay} = \text{Total Pay} + \text{Bonus}$
- The bonus part is conditional and is based on a simple decision.
- Let's draw the flowchart

Flowchart for example 5.1





The if statement

- In Programming, to be able to solve problems which involve decision making, we shall make use of the If statement.
- In the simplest form, the if statement (in pseudocodes) is as follows:

```
If <condition> Then  
    <statement(s)>  
End If
```

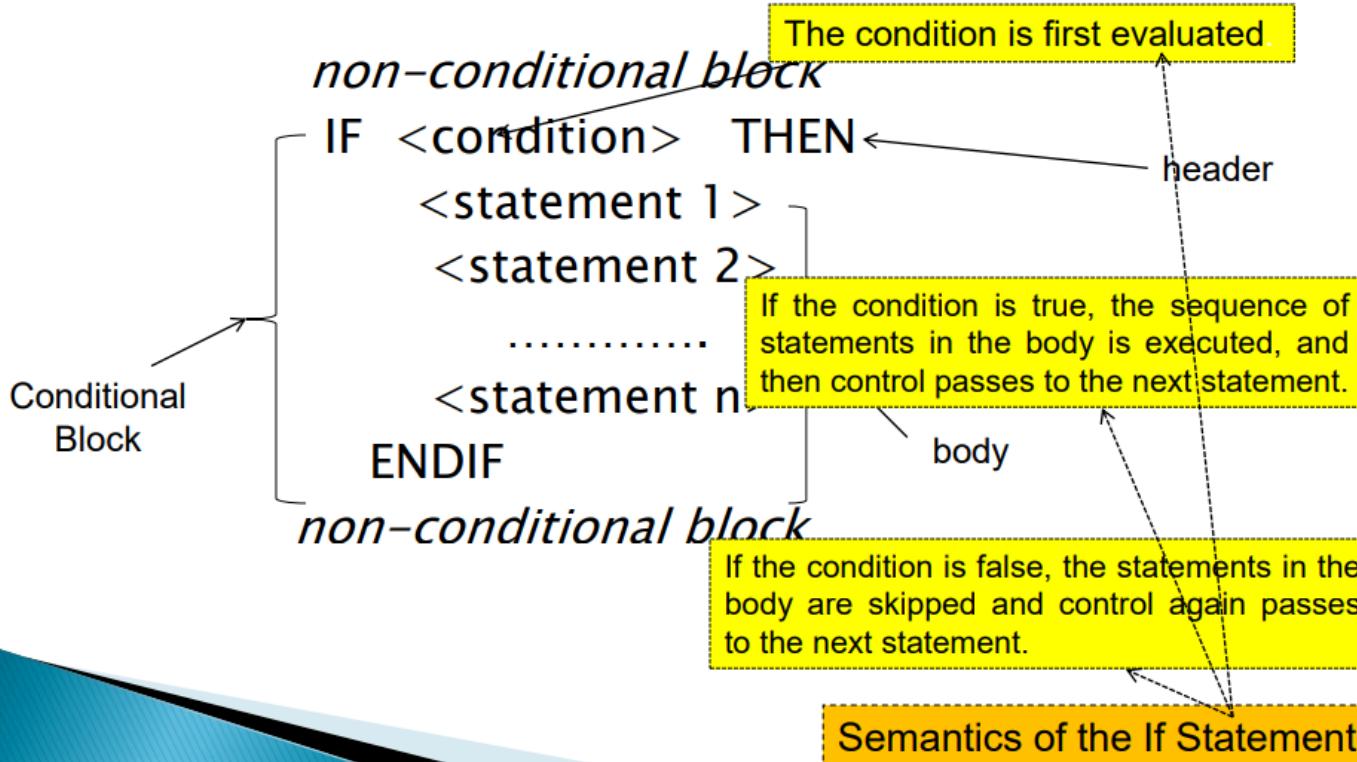
- The condition is evaluated first. If the *condition* is *true then* the statement(s) block is executed. *Otherwise*, the next statement following the statement(s) block is executed.

General Form of the basic if statement

The diagram illustrates the general form of a basic if statement. It features a large curly brace on the left labeled "Conditional Block". Inside this brace, the code structure is shown. At the top is the header "IF <condition> THEN". Below the header is the body, which consists of multiple statements: "<statement 1>", "<statement 2>", a dotted line indicating continuation, and "<statement n>". The entire body is enclosed in another curly brace labeled "body". At the bottom of the code structure is the keyword "ENDIF". Above the entire conditional block, the text "non-conditional block" is repeated twice, once above the header and once below the "ENDIF" keyword.

```
non-conditional block
IF <condition> THEN ← header
    <statement 1>
    <statement 2>
    .....
    <statement n> } ← body
ENDIF
non-conditional block
```

Semantics of the if statement





Semantics of the *if* statement

- The condition is first evaluated.
- If it is true, the sequence of statements in the body is executed, and then control passes to the next statement.
- If the condition is false, the statements in the body are skipped and control again passes to the next statement.

Pseudocode for example 5.1

- Problem: Consider a person being paid on a daily rate of Rs. 400. At the end of the month, he obtains his salary. Additionally, he obtains a bonus of Rs 2000 if he has worked for more than 25 days. We need a program to calculate his total pay at the end of a month

```
Input No_of_days  
Total_Pay= No_of_days *400 } Non-  
IF (No_of_days>25) THEN  
    Bonus=2000  
    Total_Pay=Total_Pay + Bonus  
ENDIF  
Output Total_Pay } Non-  
Conditional Block
```

The pseudocode illustrates a conditional logic structure. It starts with an input statement for 'No_of_days'. This leads to a conditional block, indicated by a bracket on the left labeled 'Conditional Block'. Inside this block, there is a calculation of 'Total_Pay' and a decision point based on whether 'No_of_days' is greater than 25. If true, it adds a 'Bonus' of 2000 to 'Total_Pay'. The entire conditional block is enclosed in 'IF' and 'ENDIF' statements. Finally, the total pay is outputted. A bracket on the right labeled 'Non-Conditional Block' covers the output statement and the part of the conditional block that follows the 'ENDIF'.



Syntax of if in Python

```
if condition:  
    #block of codes
```

Program in Python for example 5.1

```
# Given daily rate and bonus condition
daily_rate = 400

# Taking input from the user
no_of_days = int(input("Enter the number of days worked in the month: "))

# Calculate the total pay
salary = no_of_days * daily_rate

# Calculate the bonus based on the condition
if no_of_days > 25:
    bonus = 2000
else:
    bonus = 0

total_pay = salary + bonus

# Display the result
print(f"The total pay at the end of the month is: Rs. {total_pay}")
```



Example 5.2

- Write a Program that takes, as input, a temperature in fahrenheit, converts it to celsius and displays:
 - The value of the temperature in Celsius
 - The message “It’s very hot” if the celsius value exceeds 35 .
 - The message “Program Over” at the end of the program
- Write the pseudocodes design of the program, draw the flowchart and write the Python program.

Note: Conversion: $celsius = (fahrenheit - 32) \times 5/9$

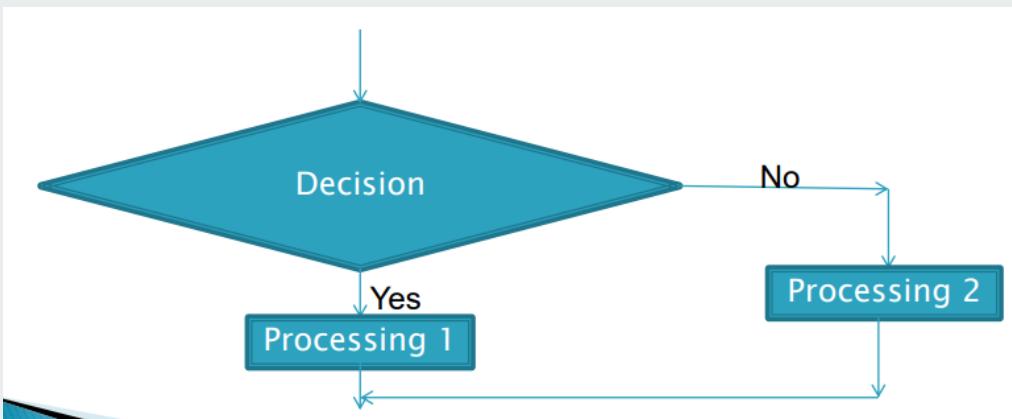


Exercise 5.3

- Modify the program in Exercise 5.2 so that it still displays the previous outputs, but additionally, if the temperature is below 12C, it displays “It’s quite cold”.
- Again write the pseudocode design, draw the flowchart and write the Python program

Using two way decisions

- The simple decision is used when we have no processing in case the condition evaluates to false.
- However, the more general case is when there is different processing depending on whether the condition is true or not.

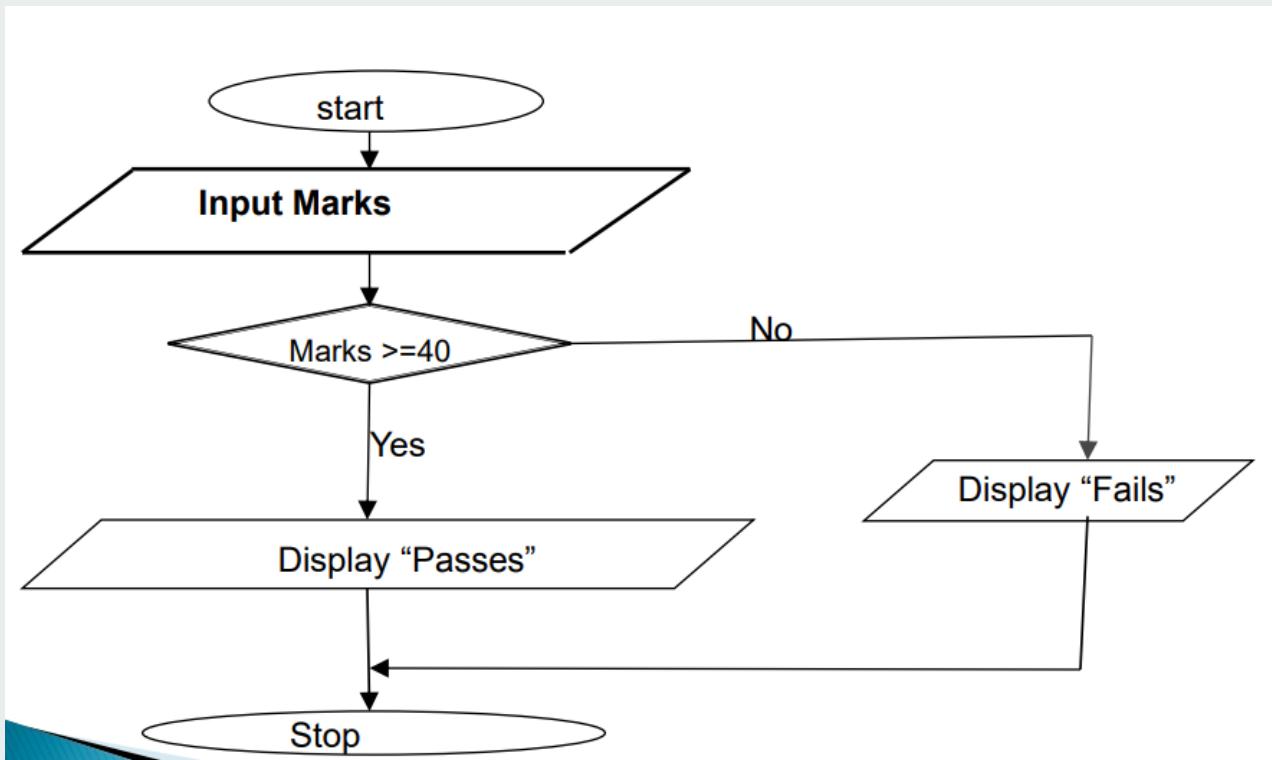




Using two way decisions

- Consider the following example:
- Example 5.2
 - A program allows the input of a student's marks in a module. If the marks are 40 or above, the program displays “Passes”. If the marks are below 40, the program displays “Fails”.
 - Draw the flow chart.

Flowchart- example 5.2





If ... else statement:

- An if statement can be combined with an else statement.
- An else statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a false value.
- The else statement is an optional statement and there could be at most one else statement following an if .



The form of the if...else statement

```
if <condition>Then  
    <Statement(s)>  
else  
    <Statement(s)>  
end if
```

- The condition is evaluated first. If the condition is true then the statement(s) in the *if* block are executed. Otherwise, the statement(s) in the *else* block is executed

Pseudocode if....else (example 5.2)

Input marks

If marks >= 40 Then

 output “Passes”

else

 output “Fails”

End if



Syntax of if..else in Python

```
if condition:  
    #block of codes  
else :  
    #block of codes
```

Program in Python example 5.2

```
# Taking input from the user
marks = float(input("Enter the student's marks: "))

# Checking the condition and displaying result
if marks >= 40:
    print("Passes")
else:
    print("Fails")
```



Exercise:5.4

- A man stitches garments for a particular factory. If he stitches 100 pieces or more in a day, he is paid at the rate of Rs 35 per garment. Otherwise, he is paid at the rate of Rs 20 per garment.
- Write the pseudocode design, draw the flowchart, and write the Python program to calculate the total amount of money earned by the man for a given day.
- The input to the program should be the number of garments stitched



Multi way decision

- In some cases, our decision can take more than two possible paths. In such cases we have to make use of the elif statement



The form of if...elif and else statement

If <condition1> Then

 <statement(s)>

elif <condition2> Then

 <statement(s)>

else

 <Statement(s)>

End if

Example for illustration

```
if x == 3:  
    print "x equals 3."  
elif x == 2:  
    print "x equals 2."  
else:  
    print "x equals something else."  
print "This is outside the 'if'."
```

Note:

- Use of indentation for blocks
- Colon (:) after boolean expression



Another if form

- An alternative if form returns a value
- This can simplify your code
- Example:
 - `return x+1 if x < 0 else x -1`
 - `return 'hold' if delta==0 else sell if delta < 0
else 'buy'`



The elif statement in Python

- The if/elif statement is a chain of if statements. They perform their tests, one after the other, until one of them is found to be true.
- The *elif* statement allows you to check multiple expressions for truth value and execute a block of code as soon as one of the conditions evaluates to true.
- Like the else, the *elif* statement is optional. However, unlike else, for which there can be at most one statement, there can be **an arbitrary number of elif statements following an if**.

If elif else :Example 5.3

Consider the following problem.

- A student is asked to input his marks. If his score is ≥ 70 , he passes with Grade A. Else If his score is below 70, but ≥ 40 he passes with Grade B. Else if his score < 40 he fails with Grade F. Write the pseudocode for the above problem to output the Grade of the student depending on his marks.

Pseudocode for example 5.3

Input marks

If marks >= 70 Then

 output “Grade A”

Else If marks >=40 Then

 Output “Grade B”

Else

 Output “Grade F”

End if

Program in Python

```
# Input the student's marks
marks = float(input("Enter the student's marks: "))

# Determine the grade based on the marks
if marks >= 70:
    grade = 'A'
elif marks >= 40:
    grade = 'B'
else:
    grade = 'F'

# Output the grade
print(f"The student's grade is: {grade}")
```



Key points

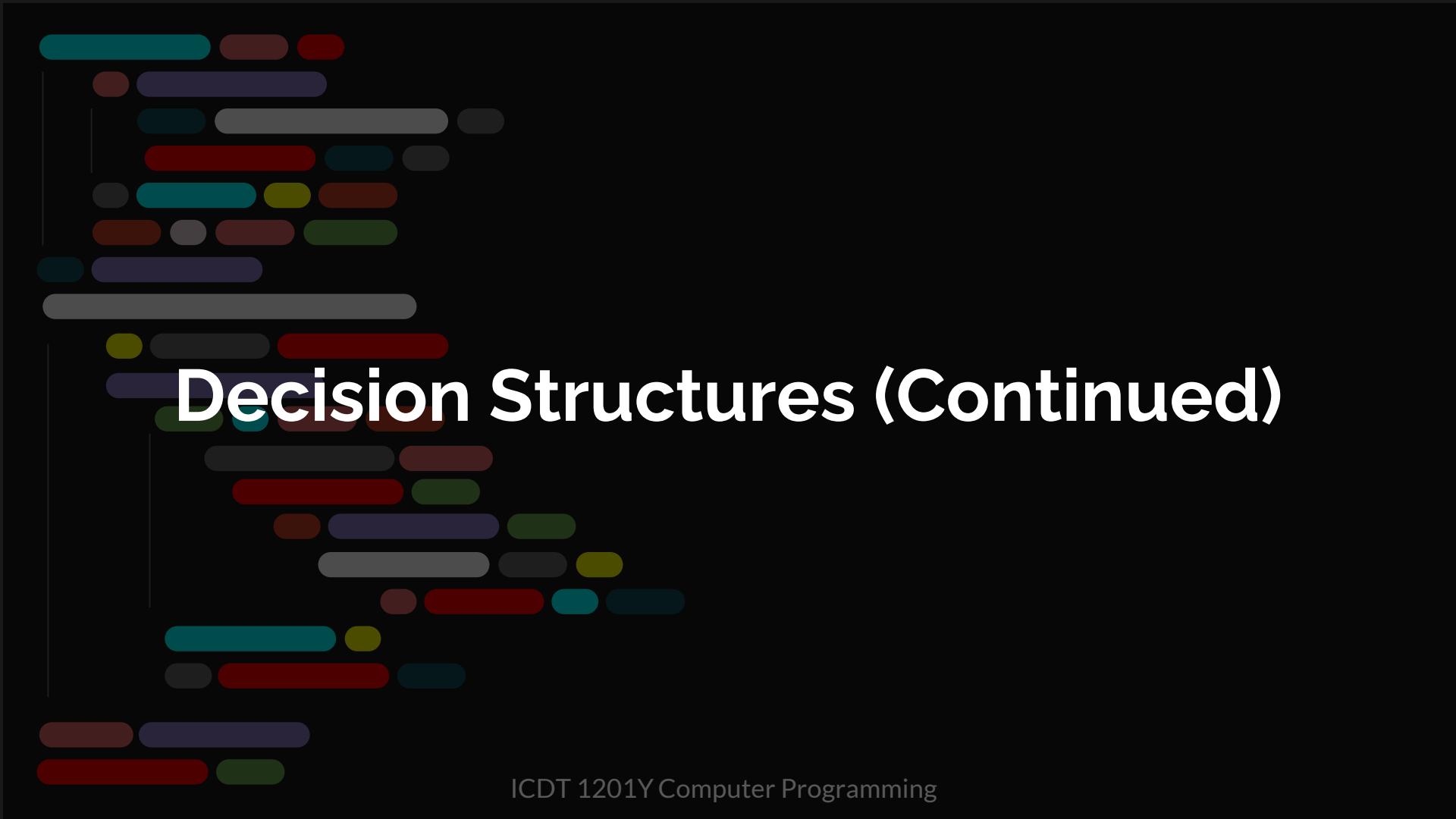
- Conditional statements, such as if, elif, and else, enable us to execute specific blocks of code based on certain conditions. They help us create branching paths in our program.



ICDT 1201Y Computer progarmming

Week4_part2

ICDT 1201Y Computer Programming



Decision Structures (Continued)



Learning Objectives

- Understand the nested if statement in Python.
- Use of the Logical Operators and, or, and not to chain together simple conditions.



The nested if statement

- There may be a situation when you want to check for another condition after a condition resolves to true.
- In such a situation, you can use the nested if construct.
- In a nested if construct, you can have an if... elif ...else construct inside another if ...elif ...else construct.

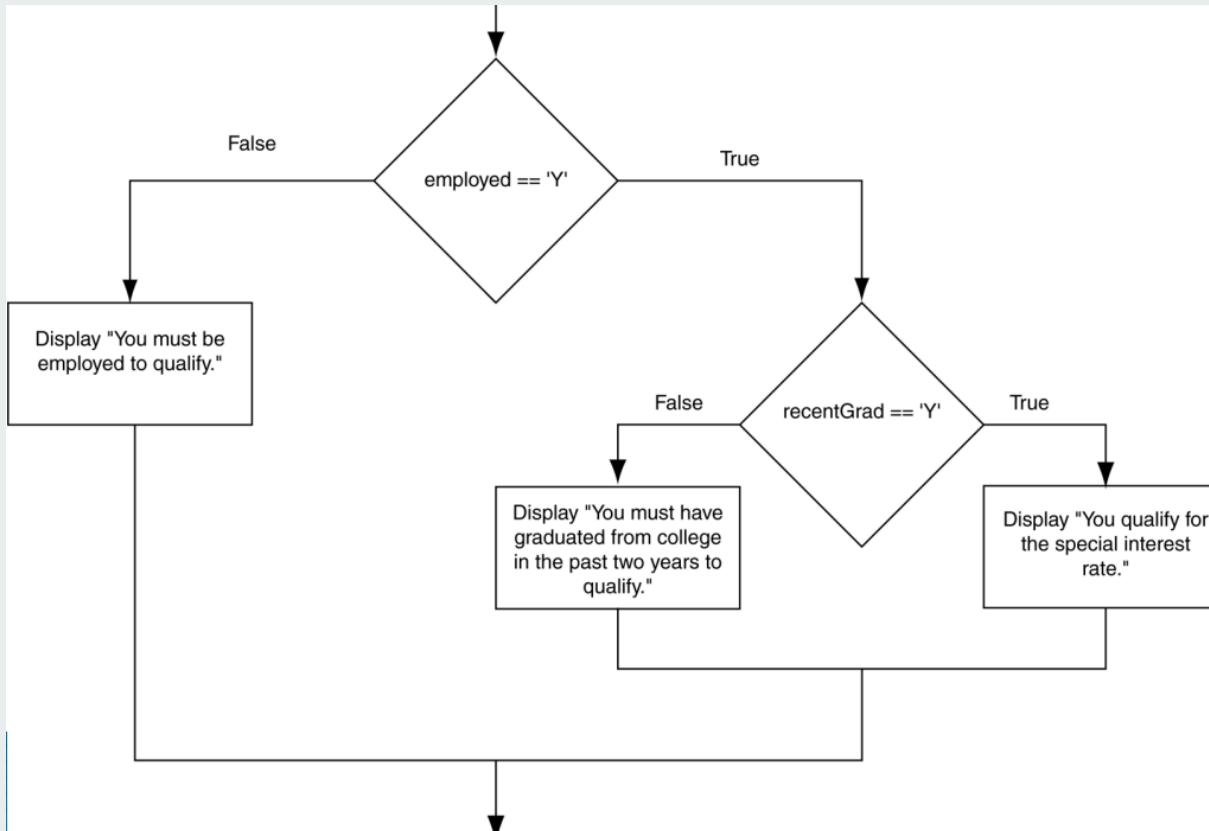


Example 4.1

Consider the following case:

- A person qualifies for a loan with a special interest rate, if he/she is employed provided he/she has recently graduated.

Flowchart for a nested if statement



Pseudocodes

```
if (employed=='Y') then
    if (recently_graduated=='Y') then
        Output "Qualified for loan with special interest rate"
    else
        Output "Not qualified for loan (not recently graduated)"
else
    Output "Not qualified for loan (not employed)"
End
```

Nested if statements

Check eligibility for special interest rate loan

```
if employment_status == "Y":  
    if graduation_status == "Y":  
        print("You qualify for the special interest rate")  
    else: #Not a recent graduate but employed  
        print("Only those who have graduated within the last two years to  
             qualify")  
else: #Not Employed  
    print("You must be employed to qualify")
```



Proper Indentation

- Proper indentation in Python is crucial for maintaining code clarity and ensuring that the structure of nested if statements (and other control structures) is visually clear and logically correct.
- Python uses indentation to define the scope of code blocks, unlike languages that use curly braces {} or keywords like begin and end



Nested if example 4.2

Consider the following problem.

- A student is asked to input his marks. If his score is ≥ 70 , he passes with Grade A. Else If his score is below 70, but ≥ 40 he passes with Grade B. Else if his score < 40 he fails with Grade F.
- Additionally there is another check if the score is ≥ 70 . If the score is above 80, the message “Excellent” gets displayed. If the score is above 90, the message “Outstanding” gets displayed.
- Exercise: Write the pseudocode for the above problem to output the Grade of the student depending on his marks and the message “Excellent” or “Outstanding” if applicable

Pseudocode nested if example 4.2

```
if student_marks >= 70 then
    Output "Grade A"
    if student_marks > 90 then
        Output "Outstanding"
    elif student_marks > 80 then
        Output "Excellent"
    End
elif student_marks >= 40 then
    Output "Grade B"
else
    Output "Grade F"
End
```



Exercise 4.1

- Convert the previous Pseudocode into a Python program

Logical Operators

- Used to create relational expressions from other relational expressions
- Operators, meaning, and explanation:

| | | |
|--|-----|---|
| | AND | New relational expression is true if both expressions are true |
| | OR | New relational expression is true if either expression is true |
| | NOT | Reverses the value of an expression – true expression becomes false, and false becomes true |

- The relational operators can be used to form compound condition



Compound Conditional Statements

Logical AND

Let's reconsider example 4.1.

- The problem can be reformulated as :
- A person qualifies for a loan with special interest, if he/she is employed and has recently graduated.
- The conditional statement can be written, in pseudocodes, as below:

```
If ((employed=='Y') And (recentGrad=='Y'))  
    Output "You qualify for the special interest rate"  
Else  
    Output "You must be employed and have recently graduated to  
        qualify for the special interest rate"  
End
```



The Logical AND Operator in Python

- In Python, the logical operator used for “AND” is the keyword “and”
- Syntax:

expression1 and expression2

Thus, in Python the code becomes:

```
if employment_status == "Y" and graduation_status == "Y":  
    print("You qualify for the special interest rate")
```

NOTE.:It's important to use parentheses to clarify the order of operations when combining multiple conditions with and, especially in complex expressions.



Logical OR operator example

- Consider that a person qualifies for a loan if his income exceeds a threshold (minimum) income or has worked for more than a minimum number of years.
- The conditional statement can be written as below (pseudocode):

```
If ((income >= MIN_INCOME) OR (years >= MIN_YEARS))
    output "You qualify for loan"
Else
    output message specifying terms to qualify
End
```



The Logical OR operator in Python

- In Python, the logical operator used for OR is “or”
- Syntax:

expression1 or expression2

```
if income >=MIN_INCOME or years >= MIN_YEARS :  
    print("You qualify for loan")  
else:  
    print("Sorry, you are not qualified for the loan ")
```



Logical not operator

- The `not` operator computes the opposite of a Boolean expression.
- `not` is a *unary* operator, meaning it operates on a single expression.

Example of not operator

```
#Example using the not operator with a variable  
a = False  
print("Original value of 'a':", a)  
print("Using 'not' operator on 'a':", not a)
```

Output:

Original value of 'a': False

Using 'not' operator on 'a': True



Logical Operator -Order of preference

- We can put these operators together to make arbitrarily complex Boolean expressions.
- In Python, when multiple logical operators (and, or, not) are used in a single expression, they are evaluated based on their precedence and associativity rules. Here's the order of precedence from highest to lowest:
- Parentheses (): Parentheses can be used to explicitly specify the order of operations.
- Unary not: Unary not operator has the highest precedence.
- Binary and: Binary and operator (and) is evaluated next.
- Binary or: Binary or operator (or) has the lowest precedence.



Logical Operator -Order of preference

Example:

- Consider the expression A and B or C:
 - *and* has higher precedence than *or*, so the expression is evaluated as (A and B) or C.
 - If A is True, B is True, and C is False, then (A and B) evaluates to True, so the entire expression evaluates to True.
 - If A is False, B is False, and C is True, then (A and B) evaluates to False, so the entire expression evaluates to True.



Logical Operator -Order of preference

- Use of Parentheses:
- To avoid confusion and to explicitly specify the order of operations, it's recommended to use parentheses in expressions involving multiple operators. For example:
- A and (B or C) specifies that B or C should be evaluated first, and then A and ... should be evaluated based on that result.

Python Comparison Operators

| comparison | Corresponding question |
|-------------------------|--|
| <code>a == b</code> | Is a equal to b ? |
| <code>a != b</code> | Is a not equal to b ? |
| <code>a > b</code> | Is a greater than b ? |
| <code>a >= b</code> | Is a greater than or equal to b ? |
| <code>a < b</code> | Is a less than b ? |
| <code>a <= b</code> | Is a less than or equal to b ? |
| <code>a in b</code> | Is the value a in the list (or tuple) b? |
| <code>a not in b</code> | Is the value a not in the list (or tuple) b? |



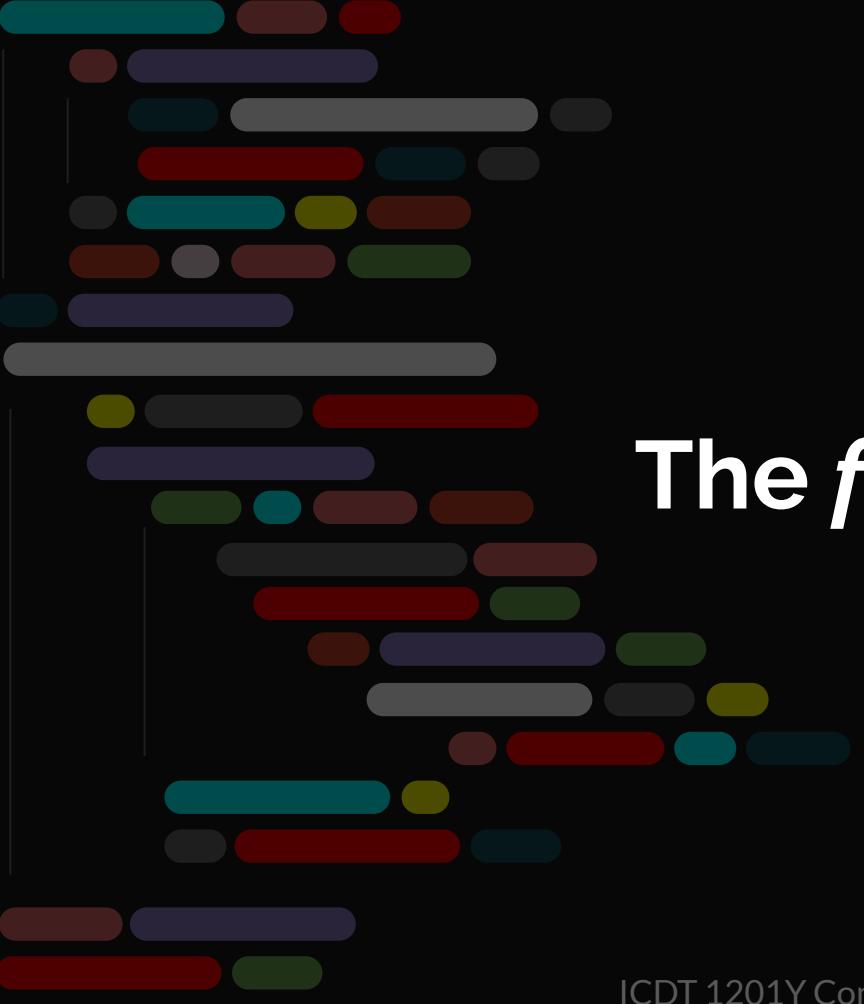
Exercise 4.2

- To be eligible to graduate from ABC University, you must have 120 credits and a CPA of at least 40.
- Write a Python program that allows the input of number of credits and CPA of a student and checks and displays the eligibility to graduate from ABC University.



ICDT1201Y Computer Programming

Week6 Loops



The *for loop*



Learning Objectives

- Understand how the `for` and nested `for` loop work
- Comprehend the use of `break` and `continue` statement in loops.



The for loop

- The `for` statement allows us to iterate through a sequence of values.
- `for <var> in <sequence>:`
 `<body>`
- The loop index variable `var` takes on each successive value in the sequence, and the statements in the body of the loop are executed once for each value.
- The sequence can be a list or a tuple



The for loop

- The keyword `in` :
- The “`in`” and “`not in`” are membership operators.
These operators check either given value is available in sequence or not.
- The “`in`” operator returns Boolean True result if value exist in sequence otherwise returns Boolean False.
- The “`not in`” operator also returns Boolean True / False result but it works opposite to “`in`” operator.



The for loop

- Suppose we want to write a program that can compute the average of a series of numbers entered by the user.
- To make the program general, it should work with any size set of numbers.
- We don't need to keep track of each number entered, we only need know the running sum and how many numbers have been added.



The for loop

- We've run into some of these things before!
 - A series of numbers could be handled by some sort of loop. If there are n numbers, the loop should execute n times.
 - We need a running sum. This will use an accumulator.



The for loop

- Input the count of the numbers, n
- Initialize sum to 0
- Loop n times
 - Input a number, x
 - Add x to sum
- Output average as sum/n



For loops & the *range()* function

- Since a variable often ranges over some sequence of numbers, the *range()* function returns a list of numbers from 0 up to but not including the number we pass to it.
- `range(5)` returns [0,1,2,3,4]
- So we could say:

```
for x in range(5) :  
    print x
```
- (There are more complex forms of *range()* that provide richer functionality...)

For Loops example

```
# average1.py
#     A program to average a set of numbers
#     Illustrates counted loop with accumulator

def main():
    n = eval(input("How many numbers do you have? "))
    sum = 0.0
    for i in range(n):
        x = eval(input("Enter a number >> "))
        sum = sum + x
    print("\nThe average of the numbers is", sum / n)
```

- Note that sum is initialized to 0.0 so that sum/n returns a float!

Results for average1.py

How many numbers do you have? 5

Enter a number >> 32

Enter a number >> 45

Enter a number >> 34

Enter a number >> 76

Enter a number >> 45

The average of the numbers is 46.4



The for loop

- The `for` loop is a definite loop, meaning that the number of iterations is determined when the loop starts.



Example using for loop

Example1:

#Print each element in a fruit list:

```
fruits = ["mango", "apple", "grapes", "cherry"]  
for x in fruits:  
    print(x)
```

output: mango
 apple
 grapes
 cherry

Example2:

```
stateName = "Ohio"  
for letter in stateName :  
    print(letter)
```

The variable `letter` takes each of the values "O", "h", "i", "o" in turn.

The output of the code is

O
h
i
o

Nested for loop

- A nested for loop is a for loop inside another for loop

Example 1:

```
for i in range(1, 3): # Outer loop
    for j in range(1, 3): # Inner loop
        print(f"({i}, {j})")
```

Output:

```
(1, 1)
(1, 2)
(2, 1)
(2, 2)
(3, 1)
(3, 2)
```

Example 2:

```
city = ["Jaipur", "Delhi", "Mumbai"]
fruits = ["apple", "mango", "cherry"]
for x in city:
    for y in fruits:
        print(x, ":", y)
```

output:

```
Jaipur : apple
Jaipur : mango
Jaipur : cherry
Delhi : apple
Delhi : mango
Delhi : cherry
Mumbai : apple
Mumbai : mango
Mumbai : cherry
```



UnConditional Control Construct



Control flow tools

- **Control Flow Tools:** Python provides additional tools like break, continue, and pass to modify the flow of control within loops and conditional statements.
- These tools enhance the flexibility and efficiency of our programs.



Loops: break and continue Statements

- `break` exits the current loop without evaluating the condition.
- `continue` skips the remainder of the current iteration, evaluates the condition again and continues the loop (if the condition is True)



Break statement

- The break statement enables to skip over a part of code that used in loop even if the loop condition remains true.
- It terminates to that loop in which it lies. The execution continues from the statement which find out of loop terminated by break.

Example 1

- Exit the loop when x is "banana":

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    if x == "banana":  
        break  
    print(x)
```

output:
apple

Example 2

```
n=1
while (n<=5) :
    print ("n=", n)
    k=1
    while (k<=5) :
        if (k==3) :
            break
        print ("k=", k, end=" ")
        k+=1
    n+=1
    print ()
```

Output

- Output:

n= 1

k= 1 k= 2

n= 2

k= 1 k= 2

n= 3

k= 1 k= 2

n= 4

k= 1 k= 2

n= 5

k= 1 k= 2



Continue statement

- With the help of continue statement, some of statements in loop, skipped over and starts the next iteration.
- It forcefully stop the current iteration and transfer the flow of control at the loop controlling condition.

Example 1

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        continue
    print(x)
```

output:
apple
cherry

Example 2

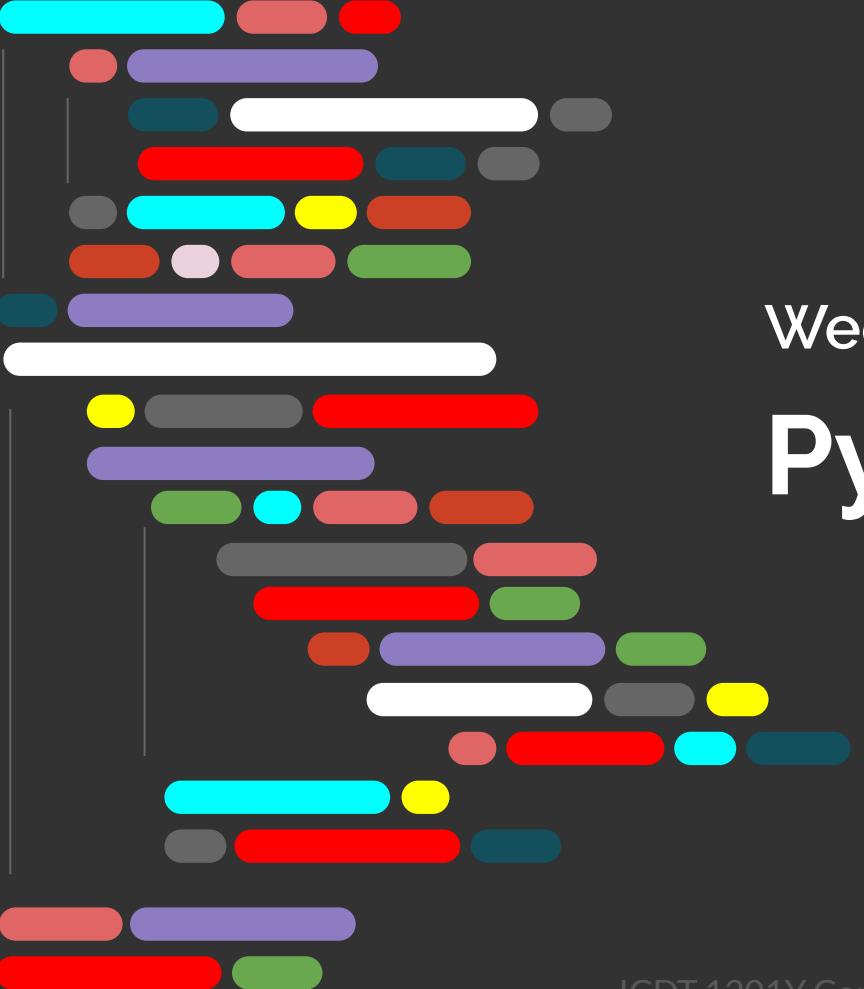
```
i = 0
while i <=10:
    i+=1
    if (i%2==1):
        continue
    print(I, end=" ")
```

output: 2 4 6 8 10



References

- Python Programming, 2/e John Zelle



Week 7

Python Lists



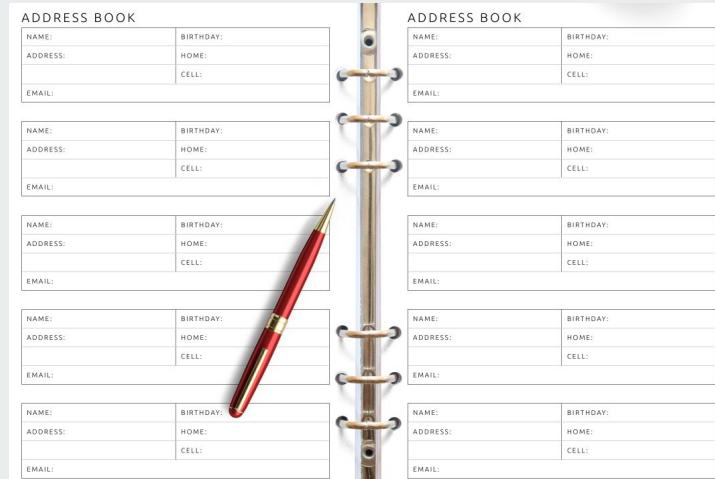
Learning Objectives

- What is a Python List?
- Why are Python Lists important?
- Python List syntax and examples.
- Accessing elements in List.
- Retrieving data from Lists.
- Working with Lists.
- Traversing the List.
- List Comprehension.

Python Lists

- Lists are among the most commonly used *data structures* in Python.
- A data structure is a container of data. It provides a way of storing, organising, accessing, processing, and retrieving data.

A common every day item that can be considered an example of a data structure is an address book, where data is kept in a tabular format.





Python Lists

- Lists can be used to store data.
- Lists are *mutable*. This means that after you create a list, you can modify its contents. You can add, remove, or change elements in the list without creating a new list.



Python Lists

- A **list** in Python is an ordered collection of items, which can be of ***any data type***.
- Any data type?
 - Remember simple variables! They can only be of one data type.
 - `x = 36.5`, This implies that variable x can only hold one type of data, in this case, a floating point number.
 - Moreover, x can contain only one piece of data at a time; here it is 36.5.
 - But Python lists can contain more than one data and each data can be of a different type.
- Why, therefore, do we need a data structure like a list?



Python Lists: why do we need them?

- Recall from all the small Python programs you have written so far:
 - they've all involved dealing with one bit of information at a time in most of the cases.
- If that's all that Python allowed us to do, then Python wouldn't be a very helpful tool for writing interesting programs.



Python Lists: why do we need them?

- Consider a situation where you are being asked to write a program that stores the names of 100 students. How will you do that using simple variables?
- Most probably, you will store each student name in a variable as follows:

| VARIABLE | STUDENT NAME |
|----------------|--------------------|
| studentName1 | = "George Orwell" |
| studentName2 | = "Charles Dicken" |
| studentName3 | = "Dan Brown" |
| . | |
| . | |
| . | |
| studentName100 | = "Carl Lewis" |



Do you find this
to be a good
practice?



Python Lists: why do we need them?

- Why is this a bad practice?

| VARIABLE | STUDENT NAME |
|----------------|--------------------|
| studentName1 | = "George Orwell" |
| studentName2 | = "Charles Dicken" |
| studentName3 | = "Dan Brown" |
| . | |
| . | |
| . | |
| studentName100 | = "Carl Lewis" |

```
studentName1 = "George Orwell"  
studentName2 = "Charles Dicken"  
studentName3 = "Dan Brown"  
. . .  
studentName100 = "Carl Lewis"
```

- Imagine you have to print all 100 student names on the screen.
 - Will you write 100 `print()` functions?
- Or suppose you have to ask a user to input the student names. How many `input()` functions will you write?
- Already, you can see the problem here.



Python Lists: why do we need them?

- In the previous code, had there been only, say, three students, it would have been OK to code it that way.
- But when there are a large number of students, the vast majority of the code will be given over to storing variables, and it will become completely unmanageable.
- And if we were to try and write a program that could process an unknown number of input sequences (for instance, by reading them from a file), we wouldn't be able to do it.



Python Lists to the rescue

- The Python List is one of the most frequently used and versatile data structures used in Python.
 - In Python, a list is a special object that can hold a number of other objects.
 - Lists are very similar to arrays in other programming languages like Java, C, C++, etc.
 - A list implements the sequence protocol, and also allows a programmer to add and remove objects from the sequence.
 - List literals are written within square brackets [] with the first element at index 0 (like strings).
 - There are many methods associated with a list in Python.



Python Lists to the rescue

- As pointed out, a Python list is a type of structure that can hold many pieces of information at the same time. This enables our programs to process multiple pieces of data.
- The list is the most versatile data type available in Python and can be written as a list of comma-separated values (items) between square brackets.
- A list is a mutable sequence of any kind of element.
- The interesting and important thing about a list is that items in it need not be of the same type.



Creating a Python List: the syntax

Lists are created by placing elements inside square brackets [], separated by commas.

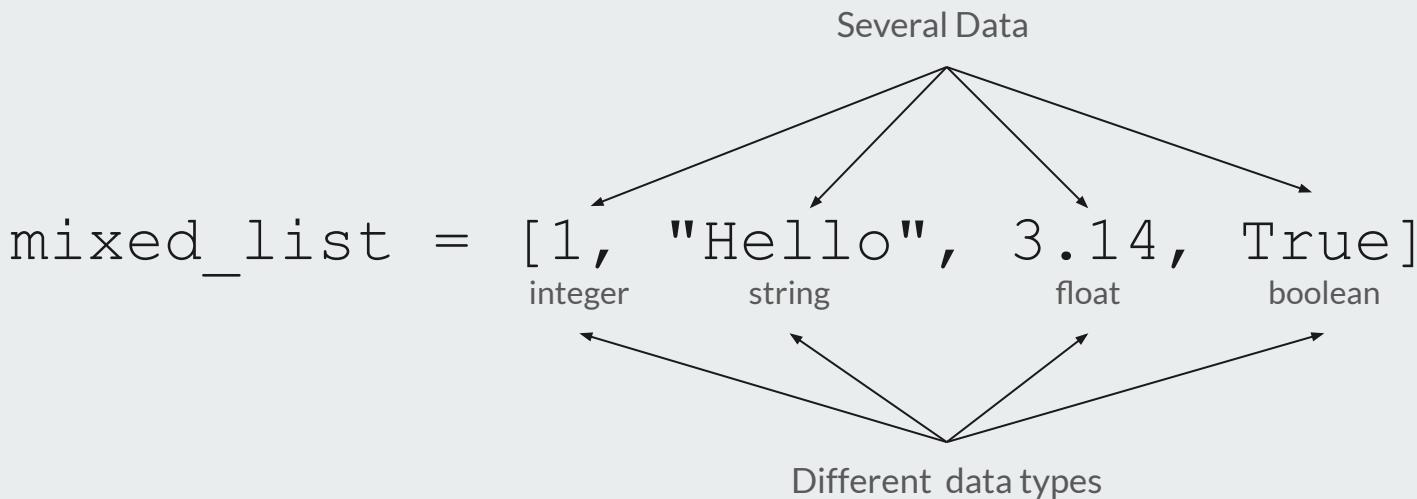
```
list_variable = [element1, element2, ... elementn]
```

Python Lists: Examples

```
list1 = ['maths', 'French', 1997, 2000]  
list2 = [1, 2, 3, 4, 5 ]  
list3 = ["a", "b", "c", "d"]  
a_list = [1, 2, 3, 4]  
b_list = ['a','b','c','d']  
c_list = ['one', 'two', 'three', 'four']  
d_list = [1,2,'three','four']  
my_list = [1, 2, 3, 4, 5]
```

Python Lists: Examples

- Notice how a list can contain several data and each data can be of a different type.





Python Lists: Examples

- We can have lists of numbers.
- We can have lists of strings.
- We can also have lists with mixed types.
- Each individual item in a list is called an element.



Python Lists: Examples

- Regarding our earlier problem of storing 100 student names, the Python list will look like this:

```
studentList = ["George Orwell",
"Charles Dickens", "Dan Brown", ...  
...  
..., "Carl Lewis"]
```



Python Lists

- Type the following code and see the output

```
mylist = ["apple", "banana", "cherry"]  
print(type(mylist))
```



Python Lists: Accessing the Elements

- According to you, how do we access the elements in the list? Say you want to print the student names on the screen. How will you proceed?
- Answer: Use loops.



Lists and Loops

- Given that a list contains many items, we use a loop to access each element – more specifically, a **for loop**.
- When processing a list, we need to execute the same action for each element in the list.
 - It is clear that we may need multiple lines of code consisting of essentially the same statement being executed multiple times, with some slight variations.
- This idea of **repetition-with-variation** is incredibly common in programming problems, and it is very useful when processing Python lists. – Using a **for loop**.



Retrieving elements from the list

To get a single element from the list, write the variable name followed by the index of the element you want in square brackets:

```
students = ["John Smith", "Mary Fox", "Tom Clinton"]
print(students[0])
student_marks = [64, 56, 82]
highest_mark = student_marks[2]
```

Retrieving elements from the list

- If we want to go in the other direction, i.e., we know which element we want but don't know the index, we can use the `index()` method:

```
students = ["John Smith", "Mary Fox", "Tom Clinton"]
student_index = students.index("Mary Fox")
# student_index is now 1.
```

- Remember that in Python, we start counting from zero rather than one, so the first element of a list is always at index zero.
- If we give a negative number, Python starts counting from the end of the list rather than the beginning, so it's easy to get the last element from a list:

```
last_student = students[-1]
```

Retrieving elements from the list

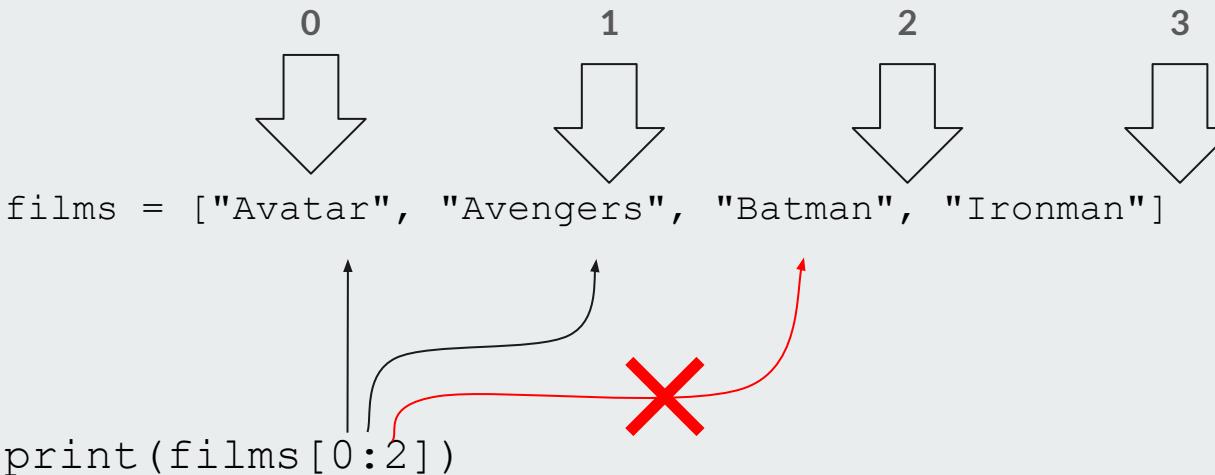
If we want to access more than one element from a list, we can give a start and stop position, separated by a colon, to specify a range of elements:

```
films = ["Avatar", "Avengers", "Batman",
         "Ironman"]
print(films[0:2])
# The code will print films starting with "A",
# i.e., "Avatar" and "Avengers"
```

- Numbers are inclusive at the start and exclusive at the end.

Retrieving elements from the list

- How does the range works



- Numbers are inclusive at the start and exclusive at the end, i.e., the range starts with element at position 0, moves to element at position 1, and stops there, one before the end position.

Class Exercise 1

- Write a Python program that will request a user to input the names of five fruits and use these names to build a list and display the contents of the list.
- Output:

Enter a fruit name: Orange

Enter a fruit name: Apple

Enter a fruit name: Kiwi

Enter a fruit name: Mango

Enter a fruit name: Litchi

```
['Orange', 'Apple', 'Kiwi', 'Mango', 'Litchi']
```



Working with list elements

- To add another element to the end of an existing list, we can use the **append()** method:

```
students = ["John", "Mary", "Tom"]
students.append("Susan")
print(students)
#The new students list becomes
#students["John", "Mary", "Tom", "Susan"]
```

- **append()** is an interesting method because it actually changes the variable on which it's used.
 - In the above example, the student list goes from having three elements to having four.

Working with list elements

- We can get the length of a list by using the `len()` function, just like we did for strings:

```
students = ["John Smith", "Mary Fox", "Tom Clinton"]
print("Initially, there are " + str(len(students)) + "
students.")
students.append("Some Student")
print("Now there are " + str(len(stduents)) + "students.")
```

- **Output**
 - Initially, there are 3 students.
 - Now there are 4 students.



Working with list elements

- We can **concatenate** two lists by using the plus (+) symbol:

```
students = ["John", "Mary", "Tom"]
modules = ["Programming", "Database"]
student_modules = students + modules
print(student_modules)
print(str(len(students)) + " students")
print(str(len(modules)) + " modules")
print(str(len(student_modules)) + " students and modules")
```

- As we can see from the output, this doesn't change either of the two original lists — it makes a brand-new list that contains elements from both:
 - ['John', 'Mary', 'Tom', 'Programming', 'Database']
 - 3 students
 - 2 modules
 - 5 students and modules



Working with list elements

- Since lists are mutable, you can change an element by assigning a new value to a specific index.

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

```
my_list[0] = 10
```

```
#now my_list is [10, 2, 3, 4]
```

```
#notice the replacement
```



Question

- Given the following list, write the appropriate code that will replace the second and third elements in the list with the word, “velociraptor”.

```
carnivores = ["Tyrex", "Diplodocus", "Triceratop"]
```



Working with list elements

- We can use the `insert()` function to insert an element at a specific index in the list.

```
my_list = [1, 2, 3, 4]
my_list.insert(1, "Python")

#this will insert the string Python
#in the list at position index 1. In
#this case, there is no replacement.
#my_list is now [1, "Python", 2, 3, 4]
```

Working with list elements

- We can use the **remove()** function to remove an element from the list.

```
my_list = [1, "Python", 2, 3, 4]
```

```
my_list.remove("Python")
```

```
#this will remove the string Python #from  
the list.
```

```
#my_list is now [1, 2, 3, 4]
```

Question: What happens if we try to remove an item that does not exist in the list?

Working with list elements

- If we want to add elements from a list onto the end of an existing list, changing it in the process, we can use the **extend()** method.
- **extend()** behaves like **append()** but takes a list as its argument rather than a single element.

```
students = ["John", "Mary", "Tom"]
print(students)
modules= ["Programming", "Database"]
students.extend(modules)
print(students)
```

- **Output**

```
['John', 'Mary', 'Tom', 'Programming', 'Database']
```



Working with list elements

- If we want to find whether an element exists in a list, we can use the **index()** method.
 - It returns the position of the **first occurrence** of an element.

```
students = ["John", "Mary", "Tom", "Tom"]
print(students.index("Tom"))
#outputs first occurrence of "Tom" which #is at
position 2 in the list
```

- An error message is displayed in case the searched element is not present in the list.
- Note that a list can contain duplicates.

Working with list elements

The **reverse()** and **sort()** functions work by changing the order of the elements in a list.

reverse() function:

```
students = ["John", "Mary",  
           "Tom"]  
print("Before: ", students)  
students.reverse()  
print("After: ", students)
```

Output

```
Before: ['John', 'Mary', 'Tom']  
After: ['Tom', 'Mary', 'John']
```

sort() function:

```
students      =      ["Mary",      "Bond",  
                     "Pam"]  
print("Before: " + students)  
students.sort()  
print("After: ", students)
```

Output

```
Before: ['Mary', 'Bond', 'Pam']  
After: ['Bond', 'Mary', 'Pam']
```

Slicing lists

- In Python, you can slice and extract elements from a list based on a start and stop index position. Use the slicer, [:].

Example1

```
a_list = ['h','e','l','l','o']
lsc = a_list[1:4]
print(lsc)
```

Output

```
['e','l','l']
```

In a_list[1:4]

The 1 means to start at the second element in the list (note that the slicing index starts at 0).

The 4 means to end at the fifth element in the list but not include it.

The colon in the middle is how Python's lists recognize that we want to use slicing to get objects from the list.

Slicing lists

- Example2

```
str = ['h', 'e', 'l', 'l', 'o']
lsc = str[:3]
print(lsc)
```

Output

```
['h', 'e', 'l']
```

- Example3

```
str = ['h', 'e', 'l', 'l', 'o']
# slice from 4th element, Python starts its lists at 0
rather than 1.
lsc = str[3:]
print(lsc)
```

Exist in list

- We can test if an item exists in a list or not, using the keyword "**in**"

```
str = ['h', 'e', 'l', 'l', 'o']
print('e' in str)
```

Output

True

Not exists

```
str = ['h', 'e', 'l', 'l', 'o']
print('e' not in str)
```

Output

False



Class Exercise 2

- Write a program that will create a list of 5 student marks and then request a user to input a position. The program will then display the element at that position in the list.

Counting the occurrence of an element in a list

- `list.count(x)` returns the number of times `x` appears in the list.

Example

```
some_string = ['h', 'e', 'l', 'l', 'o']
num_count = some_string.count('l')
print(num_count)
```

Output:

2

Delete from list

| Delete Operation | Effect |
|-----------------------------|---|
| <code>del lst[n]</code> | Remove the n^{th} element from <code>lst</code> . |
| <code>del lst[i:j]</code> | Remove the i^{th} through j^{th} elements from <code>lst</code> . |
| <code>del lst[i:j:k]</code> | Remove every k elements from i up to j from <code>lst</code> . |

Delete from list

Example 1

```
students = ["John Smith", "Mary Fox", "Tom Clinton", "Peter Parker"]
del students[2]
print(students)
```

```
['John Smith', 'Mary Fox', 'Peter Parker']
```

Example 2

```
students = ["John Smith", "Mary Fox", "Tom Clinton", "Peter Parker"]
del students[1:3]
print(students)
```

```
['John Smith', 'Peter Parker']
```

Example 3

```
students = ["John Smith", "Mary Fox", "Tom Clinton", "Peter Parker", "Albert
Einstein"]
del lst[0:4:2]
print(students)
```

```
['Mary Fox', 'Peter Parker', 'Albert Einstein']
```

Clearing or Emptying a list

- The **clear()** method can be used to remove all items from the list.

```
numbers = ['one', 'two', 'three', 'four', 'five']
print("Initially list = ", numbers)
numbers.clear()
print("After clearing contents, list = ", numbers)
```

Output:

```
Initially list = ['one', 'two', 'three', 'four', 'five']
After clearing contents, list = []
```

Appending a list inside a list

- The **append ()** method can be used to append a list inside another list.

```
num1 = [1, 2, 3]
num2 = [4, 5, 6]
num1.append(num2)
print(num1)
```

Output:

```
[1, 2, 3, [4, 5, 6]]
```



Getting the maximum and minimum values of a numerical list

- The functions `min()` and `max()` return the minimum and maximum values of a given numerical list, respectively.

```
vals=[ 6, 4, 3, 5, 2, 1 ]
```

```
max(vals)
```

Output: 6

```
min(vals)
```

Output: 1

What happens when we use min() and max() with a list of strings?

```
names = ["John", "Ann", "Mary"]
```

```
max(names)
```

Output: 'Mary'

```
min(names)
```

Output: 'Ann'

Note:

max() returns the string that appears last, and **min()** returns the string that appears first when the list is in ascending order!



Summing up the content of a numerical list

- The function **sum()** returns the sum of the contents of a numeric list.

```
vals=[6, 4, 3, 5, 2, 1]  
sum(vals)
```

Output: 21



Using the * operator with a list

- Using the * operator repeats a list a given number of times.

```
num1 = ['hi']  
num = num1 * 4  
print(num)
```

Output:

```
['hi', 'hi', 'hi', 'hi']
```

```
num1 = [1, 2, 3, 4]  
num=num*2  
print(num)
```

Output:

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



Class Exercise 3

- Write a program that will create a list of names, display the list and:
 - Delete the middle element of the list and display the list again.
 - Display the list in reverse order.
 - Display the list sorted.



Using a loop to traverse a list

- Consider the list of students, as follows:

```
students = ["John Smith", "Mary Fox", "Tom Clinton"]
```

- Suppose we need to print each element of the list as follows:

John Smith **is a student**

Mary Fox **is a student**

Tom Clinton **is a student**

- One way to do it:**

```
print(students[0] + " is a student")
print(students[1] + " is a student")
print(students[2] + " is a student")
```



Using a loop to traverse a list

- But writing those 3 lines seem very repetitive and relies on us knowing the number of elements in the list.
- What we need is a way to say something along the lines of "*for each element in the list of students, print out the element, followed by the words 'is a student.'*"
- We can use Python loops (**for** loop) to allow us to express those instructions.



Using a loop to traverse a list

- The **for** loop syntax to be used with list:

```
for <x> in <list>:
```

use the current element each time, round the loop

where **<x>** is the name we want to use for the current element and **<list>** is the name of the list we want to process.



Using a loop to traverse a list

- The loop variable <x> represents each element in a list:
 - is just a variable name , but it behaves slightly differently from all the other variables we've seen so far.
 - In all previous examples, we create a variable and store something in it, and then the value of that variable doesn't change unless we change it ourselves.
 - In contrast, when we create a variable to be used in a loop, we don't set its value; the value of the variable will be automatically set to each element of the list in turn, and it will be different each time round the loop.
 - only exists inside the loop.
 - It gets created at the start of each loop iteration and disappears at the end,i.e., when the loop has finished running for the last time



Using a loop to traverse a list

- This first line of the loop ends with a colon.
 - All the subsequent lines (just one, in this case) are indented.
 - Indented lines can start with any number of tab or space characters, but they must all be indented in the same way.
 - This pattern, i.e., a line that ends with a colon, followed by some indented lines, is very common in Python, and we'll see it in several more places throughout.
 - The indented block is known as the body of the loop, and the lines inside it will be executed once for each element in the list.
 - To refer to the current element, we use the variable name that we wrote in the first line.



Using a loop to traverse a list

- The body of the loop can contain as many lines as we like and can include all the functions and methods that we've learned about, with one important exception:
 - we're not allowed to change the list while inside the body of the loop.



Using a loop to traverse a list

- So for our example, the following codes will give the desired output:

```
for s in students:  
    print(s + " is a student")
```

Using loop to traverse a list

- Another example

```
apes = ["Homo sapiens", "Pan troglodytes", "Gorilla gorilla"]
for ape in apes:
    name_length = len(ape)
    first_letter = ape[0]
    print(ape + " is an ape. Its name starts with " +
          first_letter)
    print("\tIts name has " + str(name_length) + " letters")
```



Using loop to traverse a list

- The body of the loop in the code above has four statements.
 - two of which are print statements.
 - so each time round the loop we'll get two lines of output. If we look at the output, we can see all six lines (3 elements x 2 print statements):
 - **Output**
 - Homo sapiens is an ape. Its name starts with H
 - Its name has 12 letters
 - Pan troglodytes is an ape. Its name starts with P
 - Its name has 15 letters
 - Gorilla gorilla is an ape. Its name starts with G
 - Its name has 15 letters



Why is a loop is better to process a list?

- To avoid redundancy
 - We only need to write the statements that have to be repeated once inside the block.
 - This also means that if we need to make a change to the code, we only have to make it once rather than doing it for each element separately.
- Another benefit of using a loop is that if we want to add some elements to the list, we don't have to touch the loop code at all.
 - Consequently, it doesn't matter how many elements are in the list, and it's not a problem if we don't know how many are going to be in it at the time when we write the code.



Indentation Errors

- Unfortunately, introducing tools like loops that require an indented block of code also introduces the possibility of a new type of error: an **IndentationError**.
- Refer to the code fragment below, which gives an **IndentationError**:

```
apes = ["Homo sapiens", "Pan troglodytes", "Gorilla gorilla"]
for ape in apes:
    name_length = len(ape)
    first_letter = ape[0]
    print(ape + " is an ape. Its name starts with " + first_letter)
    print("\tIts name has " + str(name_length) + " letters")
```



More on loops with lists

- We can also loop through items in a list by referring to their index number.
- We use the `range()` and `len()` functions to create a suitable iterable.



More on loops with lists

- The following codes print all items in the list by referring to their index number:

```
thislist = ["apple", "banana", "cherry"]  
for i in range(len(thislist)):  
    print(thislist[i])
```

Output:

apple
cherry
banana



Class Exercise 4

- Rewrite the previous codes using the **while** loop.



Another method for creating a list

- A list can also be created by turning a non list object into a list by using the built-in function `list()`. The `list()` method is also called a constructor method.

```
greet="Hello"  
list(greet)
```

Output:

```
['H', 'e', 'l', 'l', 'o']
```



Another method for creating a list

- We can also use the `list()` method as follows to create a list:

```
mylist = list(("ferrari", "bugatti", "maserati"))
# Note the double round brackets.
print(thislist)
```



Copy () methods for lists

- Copy a list using the **copy ()** method.

```
sports_list = ["football", "badminton", "volleyball"]
mylist = sports_list.copy()
print(mylist)
#This will copy the content of of sports_list to #mylist.
```

- Avoid doing this:

```
mylist = sports_list
```

This will make mylist **reference** sports_list and
any changes in the latter will be reflected in the former



List Comprehensions

- List Comprehensions to create new list with dynamic values.
- **List Comprehension vs For Loop in Python**
 - It provides a concise way to apply an operation to the values in a sequence.
 - It creates a new list in which each element is the result of applying a given operation to a value from a sequence (e.g., the elements in another list).
 - It is an alternative way of doing things iteratively.



Syntax for list Comprehension

- [expression for item in list]



List Comprehension

- Iterating through a string using a **for** loop:

```
h_letters = []
for letter in 'human':
    h_letters.append(letter)
print(h_letters)
```

Output

```
['h', 'u', 'm', 'a', 'n']
```



List Comprehension

- Iterating through a string using list comprehension:

```
h_letters = [letter for letter in 'human']
print(h_letters)
```

Output

```
['h', 'u', 'm', 'a', 'n']
```



Another syntax for List Comprehension

- List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

```
newlist = [expression for item in iterable if condition == True]
```

- The return value is a new list, leaving the old list unchanged.
- *condition*: it is like a filter that only accepts the items that evaluate to True .
- *iterable*: can be any iterable object, in our case a list.
- *expression*: it is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list.



List Comprehension Example

- Given the following list of countries, suppose we want to create a new list from it but only with countries that satisfy a given condition. Say we only want countries that have the letter “a” in them.

```
country_list = ["Mauritius", "Seychelles",  
"Maldives", "Singapore"]
```

List Comprehension Example

- If we use loop, the code will look like this:

```
country_list = ["Mauritius", "Seychelles", "Maldives",
"Singapore"]
newlist = []

for x in country_list:
    if "a" in x:
        newlist.append(x)

print(newlist)
```

List Comprehension Example

- But with list comprehensive, the code is shorter:

```
country_list = ["Mauritius", "Seychelles", "Maldives",
"Singapore"]
newlist = [x for x in country_list if "a" in x]
print(newlist)
```



List Comprehension Example

- Use list comprehensive to make a new list without Singapore in it:

```
country_list = ["Mauritius", "Seychelles", "Maldives",
"Singapore"]
newlist = [x for x in country_list if x != "Singapore"]
print(newlist)
```

The condition `x != "Singapore"` will return True for all elements other than "Singapore", making the new list contain all countries except "Singapore".



List Comprehension Example

- If you do not include the condition, the new list will contain all elements from the original list.

```
country_list = ["Mauritius", "Seychelles", "Maldives",
"Singapore"]
newlist = [x for x in country_list]
print(newlist)
```

List Comprehension Example

- The iterable so far in our code has been a list, but we can use any other iterable in the code. An example, we can use the range() function.

```
newlist = [x for x in range(10)]  
print(newlist)  
#This will print [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```



List Comprehension More Examples

- We can use the list comprehensive as follows:

```
newlist = [x.upper() for x in country_list]  
print(newlist)  
#This will print ["MAURITIUS",  
#"SEYCHELLES", "MALDIVES", "SINGAPORE"]
```

List Comprehension More Examples

- We can use the list comprehensive as follows:

```
newlist = ["UoM" for x in country_list]  
print(newlist)  
#This will print ["UoM", "UoM", "UoM",  
"UoM"]
```



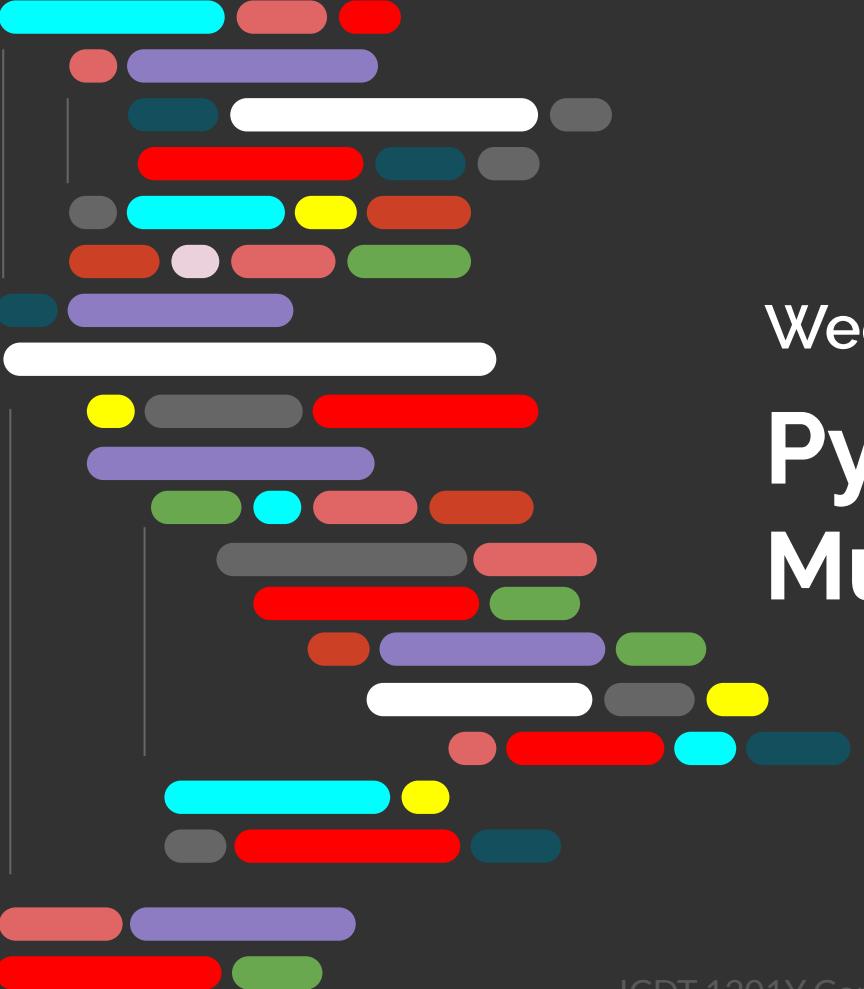
List Comprehension More Examples

```
country_list = ["Mauritius", "Seychelles", "Maldives",
"Singapore"]
newlist = [x if x != "Maldives" else "Rodrigues" for x in
country_list]
print(newlist)
```

Type the code and see the result.



Have fun coding in Python



Week 7: Part 2

Python: Multi-dimensional List



Learning Objectives

- Multi-dimensional List vs. One-dimensional List
- Multi-dimensional List inner structure and syntax
- Multi-dimensional List declaration and initialisation
- Multi-dimensional List: populating the list
- Multi-dimensional List: displaying the content
- Multi-dimensional List: More examples



Python List: What you have learned so far

- Up till now, we have only considered one dimensional lists.
 - What does this mean?
 - Consider the following list:

```
my_list = [Brinda, Zohra, Kevin, Ali]
```
 - This is considered to be a one dimensional list as it contains only one row of data.

Python List: One Dimensional (1D) List

- One dimensional, (1D) list explained.

```
my_list = [Brinda, Zohra, Kevin, Ali]
```

Visual representation of what the above 1D list looks like:

| Index Number ➔ | 0 | 1 | 2 | 3 |
|-------------------|--------|-------|-------|-----|
| One row of data ➔ | Brinda | Zohra | Kevin | Ali |

Python List: One Dimensional (1D) List

- `print(my_list)` will print the list in one line as follows:
Output: ['Brinda', 'Zohra', 'Kevin', 'Ali']
- You can choose to print the list using a `for` loop:

```
for x in my_list:  
    print(x)
```

Output:

Brinda
Zohra
Kevin
Ali



Notice the output for the `for` loop is 4 successive lines of data. This is because each time the loop executes, it prints the data element onto a new line.



Python List: Multi-Dimensional List

- Some data that we want to store and manipulate can occupy more than one dimension.
- e.g., a table in which data is kept in rows and columns.

Python List: Multi-Dimensional List

- A table containing data

The diagram illustrates a multi-dimensional list as a table. A brace on the left side is labeled "Rows" and spans all five rows of the table. A brace at the top is labeled "Columns" and spans both columns of the table. The table itself has two columns: "Student Name" and "Sports". The data is as follows:

| Student Name | Sports |
|--------------|----------|
| Brinda | Fencing |
| Zohra | Shotput |
| Kevin | Swimming |
| Ali | Archery |

Python List: Multi-Dimensional List

- Table cell addresses:

| | Column 0 | Column 1 |
|--------|--------------------|--------------------|
| Row 0: | Row[0], Coloumn[0] | Row[0], Coloumn[1] |
| Row 1: | Row[1], Coloumn[0] | Row[1], Coloumn[1] |
| Row 2: | Row[2], Coloumn[0] | Row[2], Coloumn[1] |
| Row 3: | Row[3], Coloumn[0] | Row[3], Coloumn[1] |
| Row 4: | Row[4], Coloumn[0] | Row[4], Coloumn[1] |



Python List: Multi-Dimensional List

- A multi-dimensional list can contain data just a table.
- In this case, we can use a two-dimensional, 2D list that is made up of i numbers of **rows** and j numbers of **columns**.

Python List: Multi-Dimensional List

- 2D List:

- $i = 5$ (rows)

- 0, 1, 2, 3, 4

- $j = 2$ (columns)

- 0, 1

| | i | j |
|---|--------|--------|
| 0 | [0][0] | [0][1] |
| 1 | [1][0] | [1][1] |
| 2 | [2][0] | [2][1] |
| 3 | [3][0] | [3][1] |
| 4 | [4][0] | [4][1] |

Python List: Multi-Dimensional List

- Accessing elements in a 2D list:
 - We write the list name followed by the row number in square brackets and the column number in square brackets.
 - E.g.,

```
print(gamelist[0][0])
```

will output Samantha

List: gamelist

| | 0 | 1 |
|---|---------|----------|
| 0 | Samanta | Chess |
| 1 | Brinda | Scrabble |
| 2 | Zohra | Uno |
| 3 | Kevin | Ludo |
| 4 | Ali | Monopoly |

Python List: Multi-Dimensional List

- How does a 2D list look?

- So the gamelist from the previous slide will be:

```
gamelist = [ ["Samantha", "Chess"], ["Brinda", "Scrabble"], ["Zohra", "Uno"], ["Kevin", "Ludo"], ["Ali", "Monopoly"] ]
```

Python List: Multi-Dimensional List

- We can as well write the 2D list as follows:

```
gamelist = [ ["Samantha", "Chess"],  
             ["Brinda", "Scrabble"],  
             ["Zohra", "Uno"],  
             ["Kevin", "Ludo"],  
             ["Ali", "Monopoly"] ]  
  
                                         } 5 rows  
  
                                         } 2 Columns
```



Python List: Multi-Dimensional List

- Another 2D list:

```
product_list = [ ["Product", "Qty", "Unit Price"],  
                 ["Fountain Pen", 100, 1000.00],  
                 ["Notebook", 50, 500.00],  
                 ["Bookmarks", 200, 25.00],  
                 ["Stickers", 500, 10.00] ]
```



Multi-Dimensional List: Initialisation

- 2D list declaration and initialisation:

```
mylist = [      ["Car", "Colour", "Price"],  
             ["Maserati", "red", 10000000.00],  
             ["Bugatti", "blue", 50000000.00],  
             ["Ferrari", "yellow", 25000000.00],  
             ["Lamborghini", "green", 15000000.00]]
```

Typing this code directly in VS Code means that you are declaring and initialising a 2D list with name mylist and the initialised values → those who are in the square brackets

Multi-Dimensional List: More Example

- 2D list : matrix example.

```
matrix = [ [1, 2, 3, 4],  
          [4, 3, 2, 1],  
          [7, 8, 0, 1] ]
```

```
matrix[0][0] = 1      matrix[0] is [1, 2, 3, 4]  
matrix[1][2] = 2      matrix[1] is [4, 3, 2, 1]  
...                  matrix[2] is [7, 8, 0, 1]
```



Multi-Dimensional List: Populating

- Populating a 2D list:
 - We start with the first row, i.e., $i = 0$, and we fill in the columns in that row, $j = 0$ and $j = 1$.
 - We then proceed to the next row, $i = 1$, and fill in the columns in that row, $j = 0$ and $j = 1$.
 - We repeat until we reach the final row, $i = 4$, and fill in the associated columns, $j = 0$ and $j = 1$.

Multi-Dimensional List: Populating

The code for populating the 2D list:

```
num_rows = int(input("Enter number of rows: "))
num_columns = int (input("Enter number of columns: "))
my2dList = []
for i in range(num_rows):
    cols = []
    for j in range(num_columns):
        data = input("Enter Data: ")
        cols.append(data)
    my2dList.append(Cols)
```

Multi-Dimensional List: Populating

Explanation:

```
num_Rows = int(input("Enter number of rows: ")) num_Columns = int  
(input("Enter number of columns: "))  
  
my2dList = []  
  
for i in range(num_Rows): # Loop through each row, starting with row [0].  
    cols = [] # Each time we are in a row, we start with an empty list to store  
              # data in each column  
  
    for j in range(num_Columns):  
        data = input("Enter Data: ")  
        cols.append(data) # Appending data to cols for each row  
  
    my2dList.append(Cols) # When we exit the inner loop we append cols to the  
                          # list
```

Python List: Multi-Dimensional List

Suppose the user inputs the following data:

Enter number of rows: **2**

Enter number of Columns: **3**

Enter Data: x

Enter Data: y

Enter Data: z

Enter Data: a

Enter Data: b

Enter Data: c

If we then `print(my2dList)` , we get the following output:

```
[['x', 'y', 'z'], ['a', 'b', 'c']]
```

Multi-Dimensional List: Displaying Content of

We can also display the content of the list using nested **for** loops:

```
for i in range(len(my2dList)) :  
    for j in range(len(my2dList[0])) :  
        print (my2dList[i][j])
```



Class Exercise

Rewrite the output using the for loops but with the data neatly displayed like in a table

More...to try by yourself

```
matrix = [] # Create an empty list  
numberOfRows = eval(input("Enter the number of rows: "))  
numberOfColumns = eval(input("Enter the number of columns: "))  
  
for row in range(0, numberOfRows):  
    matrix.append([]) # Add an empty new row  
    for column in range(0, numberOfColumns):  
        value = eval(input("Enter an element and press Enter: "))  
        matrix[row].append(value)  
  
print(matrix)
```

More...to try by yourself

```
import random

matrix = [] # Create an empty list

numberOfRows = eval(input("Enter the number of rows: "))
numberOfColumns = eval(input("Enter the number of columns: "))
for row in range(0, numberOfRows):
    matrix.append([]) # Add an empty new row
    for column in range(0, numberOfColumns):
        matrix[row].append(random.randrange(0, 100))

print(matrix)
```

More...to try by yourself

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] # Assume a list is given  
  
for row in range(0, len(matrix)):  
  
    for column in range(0, len(matrix[row])):  
  
        print(matrix[row][column], end = " ")  
  
    print() # Print a newline
```

More...to try by yourself

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
total = 0  
  
for row in range(0, len(matrix)):  
    for column in range(0, len(matrix[row])):  
        total += matrix[row][column]  
  
print("Total is " + str(total)) # Print the total
```

More...to try by yourself

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
total = 0  
  
for column in range(0, len(matrix[0])):  
    for row in range(0, len(matrix)):  
        total += matrix[row][column]  
  
print("Sum for column " + str(column) + " is " + str(total))
```

More...to try by yourself

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] # Assume a list is given  
maxRow = sum(matrix[0]) # Get sum of the first row in maxRow  
  
indexOfMaxRow = 0  
for row in range(1, len(matrix)):  
    if sum(matrix[row]) > maxRow:  
        maxRow = sum(matrix[row])  
        indexOfMaxRow = row  
  
print("Row " + str(indexOfMaxRow))
```

More...to try by yourself

```
import random

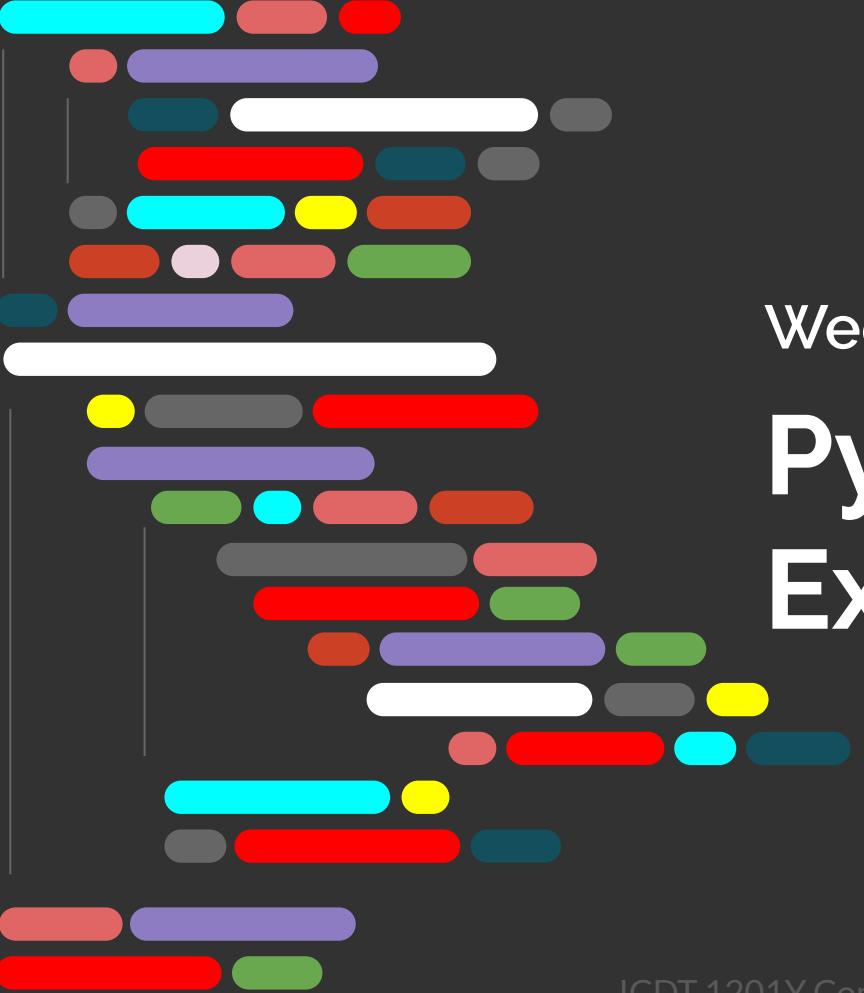
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] # Assume a list is given

for row in range(0, len(matrix)):
    for column in range(0, len(matrix[row])):
        i = random.randrange(0, len(matrix))
        j = random.randrange(0, len(matrix[row]))
        # Swap matrix[row][column] with matrix[i][j]
        matrix[row][column], matrix[i][j] = \
            matrix[i][j], matrix[row][column]

print(matrix)
```



Have fun coding in Python



Week 8

Python Files and Exceptions



Topics

- Introduction to File Input and Output
- Using Loops to Process Files
- Using the `with` Statement to Open Files
- Processing Records
- Exceptions



Introduction to File Input and Output

- For a program to retain data between the times it is run, you must save the data.
 - Data is saved to a file, typically on computer disk.
 - Saved data can be retrieved and used at a later time.
- “Writing data to”: saving data on a file
- Output file: a file that data is written to



Introduction to File Input and Output

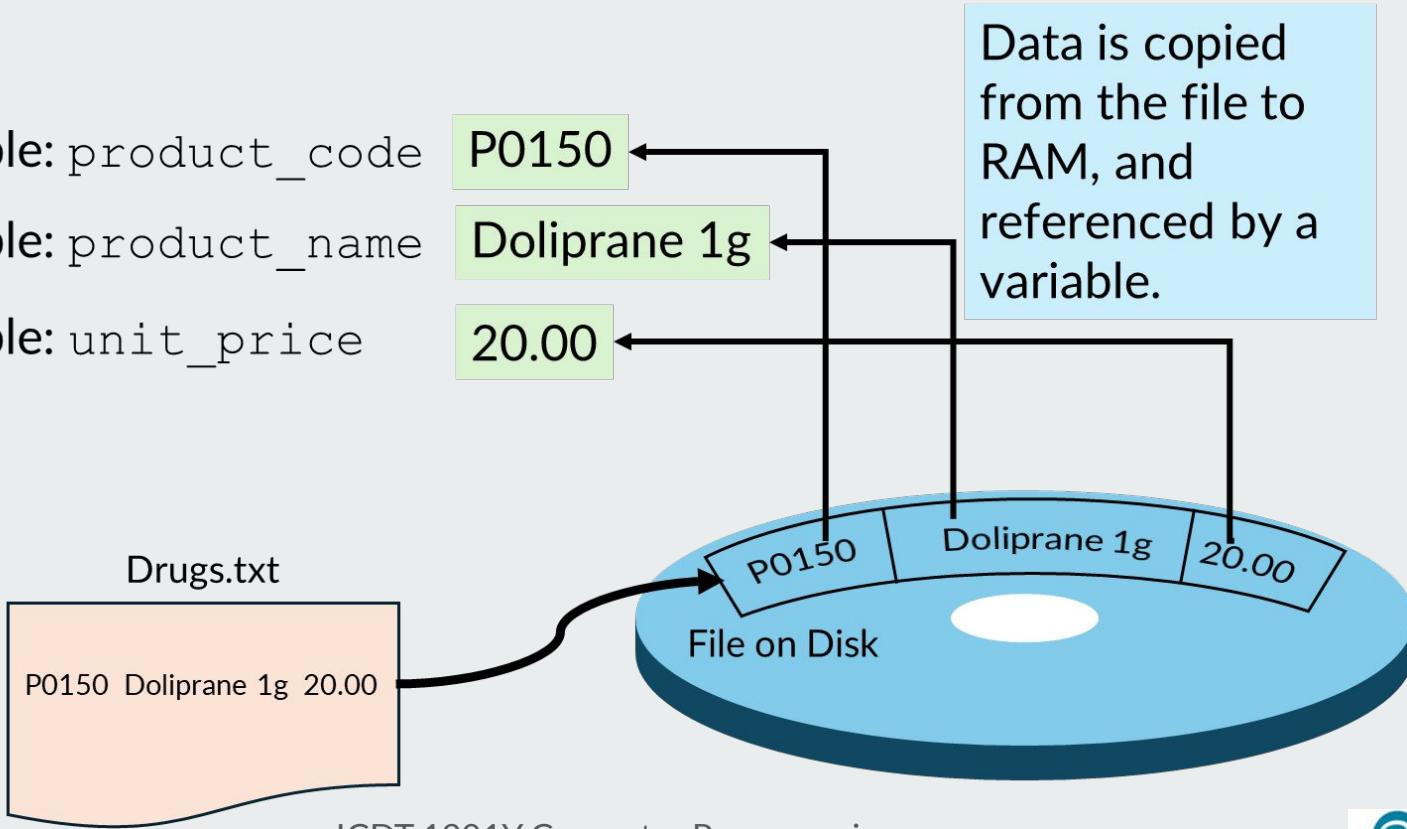
- “Reading data from”: process of retrieving data from a file
- Input file: a file from which data is read
- Three steps when a program uses a file
 - Open the file
 - Process the file
 - Close the file

File Data

Variable: product_code

Variable: product_name

Variable: unit_price





Types of Files and File Access Methods

- In general, two types of files
 - **Text file:** contains data that has been encoded as text.
 - **Binary file:** contains data that has not been converted to text.
- Two ways to access data stored in file
 - **Sequential access:** file read sequentially from beginning to end, can't skip ahead
 - **Direct access:** can jump directly to any piece of data in the file

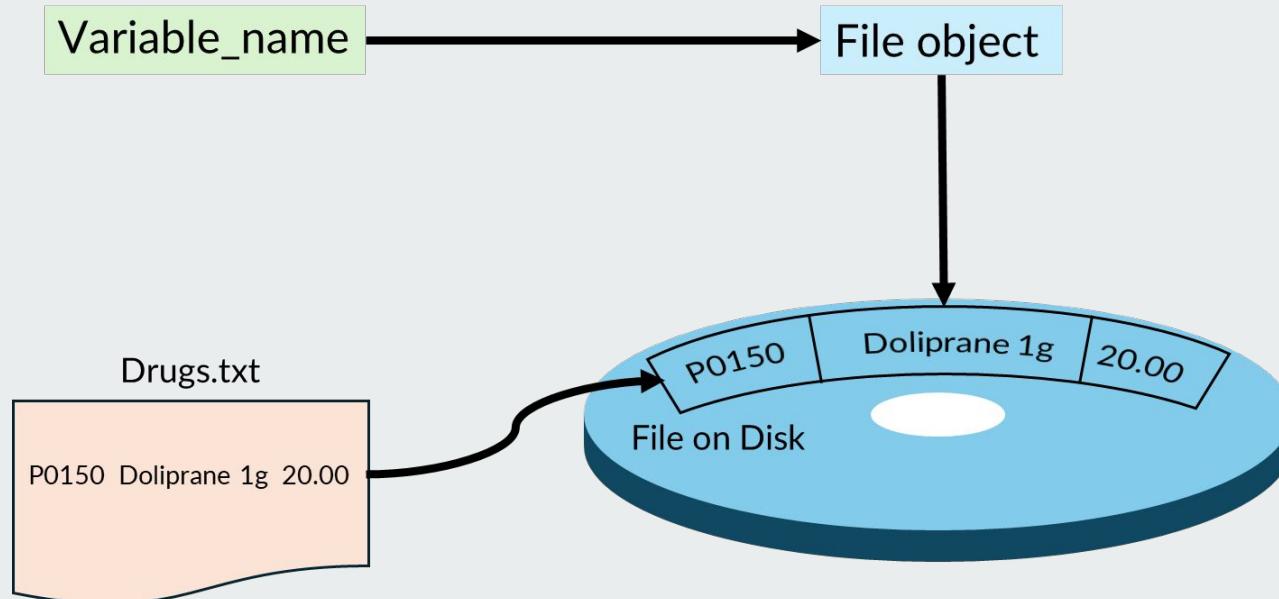


Filenames and File Objects

- **Filename extensions:** short sequences of characters that appear at the end of a filename preceded by a period
 - Extension indicates type of data stored in the file
- **File object:** object associated with a specific file
 - Provides a way for a program to work with the file: file object referenced by a variable

Filenames and File Objects

- A variable name references a file object that is associated with a file.





Opening a File

- **open function:** used to open a file
 - Creates a file object and associates it with a file on the disk.
 - General format:

```
file_object = open(filename, mode)
```
- **mode:** string specifying how the file will be opened
 - Example: reading only ('`r`'), writing ('`w`'), and appending ('`a`')



Specifying the Location of a File

- If `open` function receives a filename that does not contain a path, it assumes that the file is in the same directory as the program.
- If the program is running and a file is created, it is created in the same directory as the program.
 - Can specify alternative path and file name in the `open` function argument
 - Prefix the path string literal with the letter `r`.



Writing Data to a File

- Method: a function that belongs to an object
 - Performs operations using that object.
- File object's `write` method used to write data to the file
 - Format: `file_variable.write(string)`
- Files should be closed using the `fileobject close` method.
 - Format: `file_variable.close()`



Reading Data From a File

- **read method:** file object method that reads entire file contents into memory
 - Only works if the file has been opened for reading.
 - Contents returned as a string.
- **readline method:** file object method that reads a line from the file
 - Line returned as a string, including ' \n '
- **Read position:** marks the location of the next item to be read from a file.



Concatenating a newline to and stripping it from a string

- In most cases, data items written to a file are values referenced by variables.
 - Usually necessary to concatenate a '\n' to data before writing it.
 - Carried out using the + operator in the argument of the `write` method
- In many cases, you need to remove '\n' from the string after it is read from a file.
 - `rstrip` method: string method that strips specific characters from end of the string



Appending Data to an Existing File

- When opening a file with 'w' mode, if the file already exists, its previous content is overwritten.
- To append data to a file, use the 'a' mode.
 - If file exists, it is not erased, and if it does not exist, it is created.
 - Data is written to the file at the end of the current contents.



Writing and Reading Numeric Data

- Numbers must be converted to strings before they are written to a file.
- `str` function: converts value to string
- Numbers are read from a text file as strings.
 - Must be converted to numeric type in order to perform mathematical operations.
 - Use `int()` and `float()` functions to convert string to numeric value.

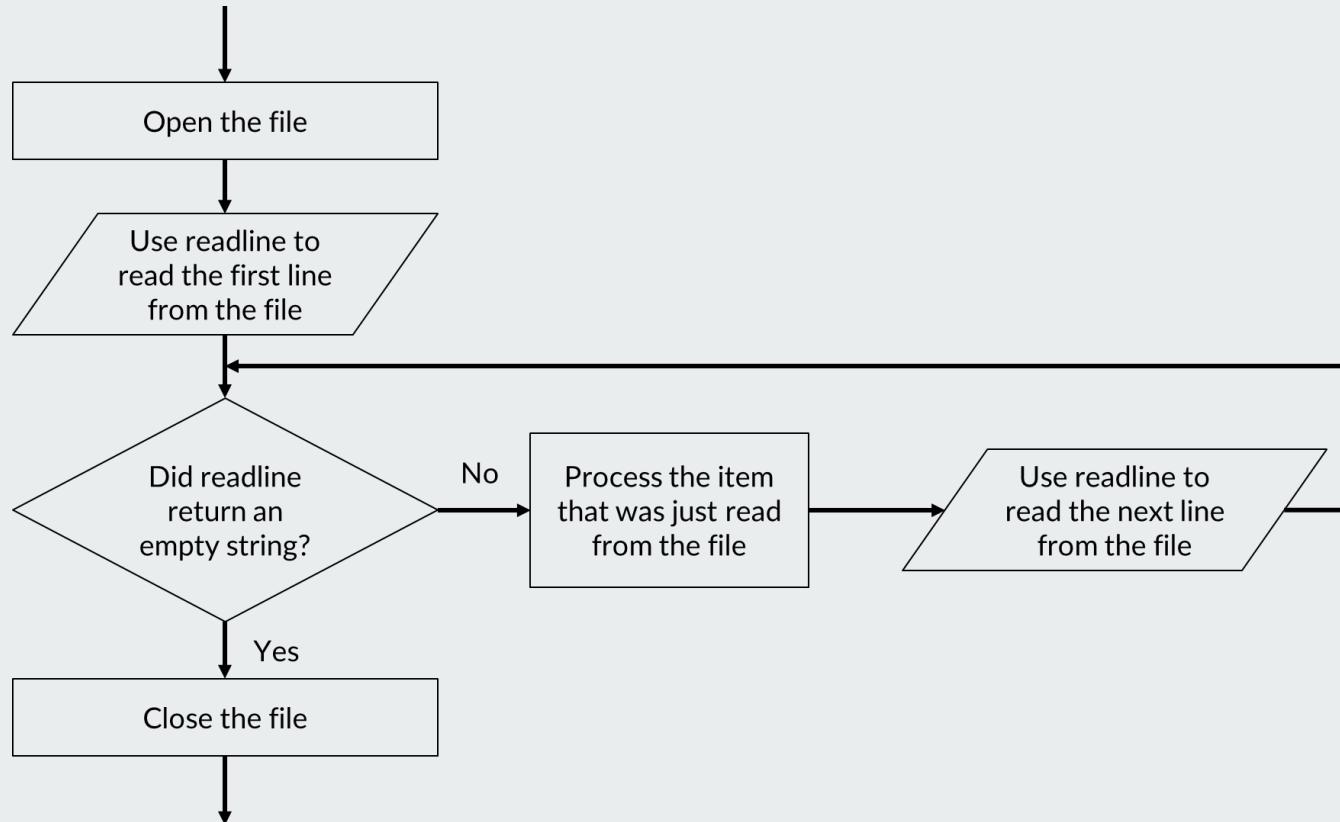


Using Loops to Process Files

- Files typically used to hold large amounts of data
 - Loop typically involved in reading from and writing to a file
- Often the number of items stored in a file is unknown.
 - The `readline` method uses an empty string as a sentinel when the end of the file is reached.
 - Can write a while loop with the condition

```
while line != ''
```

General Logic for Detecting the End of a File





Using Python's `for` Loop to Read Lines

- Python allows the programmer to write a `for` loop that automatically reads lines in a file and stops when the end of the file is reached.
 - Format:

```
for line in file_object:  
    statements
```
 - The loop iterates once over each line in the file.

Using the `with` Statement to Open Files

- The `with` statement can be used to open a file and automatically close the file. General format:

```
with open(filename, mode) as file_variable:  
    statement  
    statement  
    etc.
```

- filename* is a string specifying the name of the file.
- mode* is a string specifying the mode, such as '`r`', '`w`', or '`a`' .
- file_variable* is the name of the variable that will reference the file object.
- The indented block of statements is known as the `with` suite.

Using the `with` Statement to Open Files

- Example:

```
with open('myfile.txt', 'w') as output_file:  
    statement  
    statement  
    etc.
```

- This example opens a file named `myfile.txt`
- Code inside the `with` suite can use `output_file` to write data to the file.
- When the code inside the `with` suite has finished, the file is automatically closed.

Using the `with` Statement to Open Files

- Example:

```
with open('myfile.txt', 'r') as input_file:  
    statement  
    statement  
    etc.
```

- This example opens a file named `myfile.txt`
- Code inside the `with` suite can use `input_file` to read data from the file.
- When the code inside the `with` suite has finished, the file is automatically closed.

Using the `with` Statement to Open Files

- Opening multiple files with a `with` statement:

```
with open('file1.txt', 'r') as input_file,  
      open('file2.txt', 'w') as output_file:  
    statement  
    statement  
    etc.
```

- This example opens a file named `file1.txt` for reading and a file named `file2.txt` for writing.
- Code inside the `with` suite can use `input_file` to read data from `file1.txt` and `output_file` to write data to `file2.txt`.
- When the code inside the `with` suite has finished, both files are automatically closed.



Processing Records

- **Record:** set of data that describes one item
- **Field:** single piece of data within a record
- Write records to sequential access file by writing the fields one after the other.
- Read record from sequential access file by reading each field until record complete.



Processing Records

- When working with records, it is also important to be able to:
 - Add records
 - Display records
 - Search for a specific record
 - Modify records
 - Delete records



Exceptions

- **Exception:** error that occurs while a program is running
 - Usually causes the program to abruptly halt
- **Traceback:** error message that gives information regarding line numbers that caused the exception.
 - Indicates the type of exception and a brief description of the error that caused the exception to be raised.



Exceptions

- Many exceptions can be prevented by careful coding.
 - Example: input validation
 - Usually involve a simple decision construct.
- Some exceptions cannot be avoided by careful coding.
 - Examples
 - Trying to convert non-numeric string to an integer
 - Trying to open for reading a file that doesn't exist



Exceptions

- Exception handler: code that responds when exceptions are raised and prevents the program from crashing.
- In Python, written as `try/except` statement
 - General format:

```
try:
```

```
    statements
```

```
except exceptionName:
```

```
    statements
```

- Try suite: statements that can potentially raise an exception
- Handler: statements contained in `except` block



Exceptions

- If statement in try suite raises exception:
 - Exception specified in except clause:
 - Handler immediately following except clause executes
 - Continue program after try/except statement
 - Other exceptions:
 - Program halts with traceback error message
- If no exception is raised, handlers are skipped



Handling Multiple Exceptions

- Often code in a `try` suite can throw more than one type of exception.
 - Need to write an `except` clause for each type of exception that needs to be handled.
- An `except` clause that does not list a specific exception will handle any exception that is raised in the `try` suite.
 - Should always be last in a series of `except` clauses.



Displaying an exception default error message

- Exception object: object created in memory when an exception is thrown.
 - Usually contains a default error message pertaining to the exception.
 - Can assign the exception object to a variable in an `except` clause
 - Example: `except ValueError as err:`
 - Can pass an exception object variable to the `print` function to display the default error message.



The else Clause

- `try/except` statement may include an optional `else` clause, which appears after all the `except` clauses
 - Aligned with `try` and `except` clauses
 - Syntax similar to `else` clause in decision structure
 - **Else suite:** block of statements executed after statements in try suite, only if no exceptions were raised.
 - If an exception was raised, the `else` suite is skipped.



The finally Clause

- `try/except` statement may include an optional `finally` clause, which appears after all the `except` clauses.
 - Aligned with `try` and `except` clauses
 - General format:

```
finally:  
    statements
```
- **Finally suite:** block of statements after the `finally` clause
 - Execute whether an exception occurs or not.
 - Purpose is to perform cleanup before exiting



What If an Exception Is Not Handled?

- Two ways for exceptions to go unhandled:
 - No except clause specifying exception of the right type
 - Exception raised outside a try suite
- In both cases, an exception will cause the program to halt.
 - Python documentation provides information about exceptions that can be raised by different functions.

starting out with >>>

PYTHON®

SIXTH EDITION



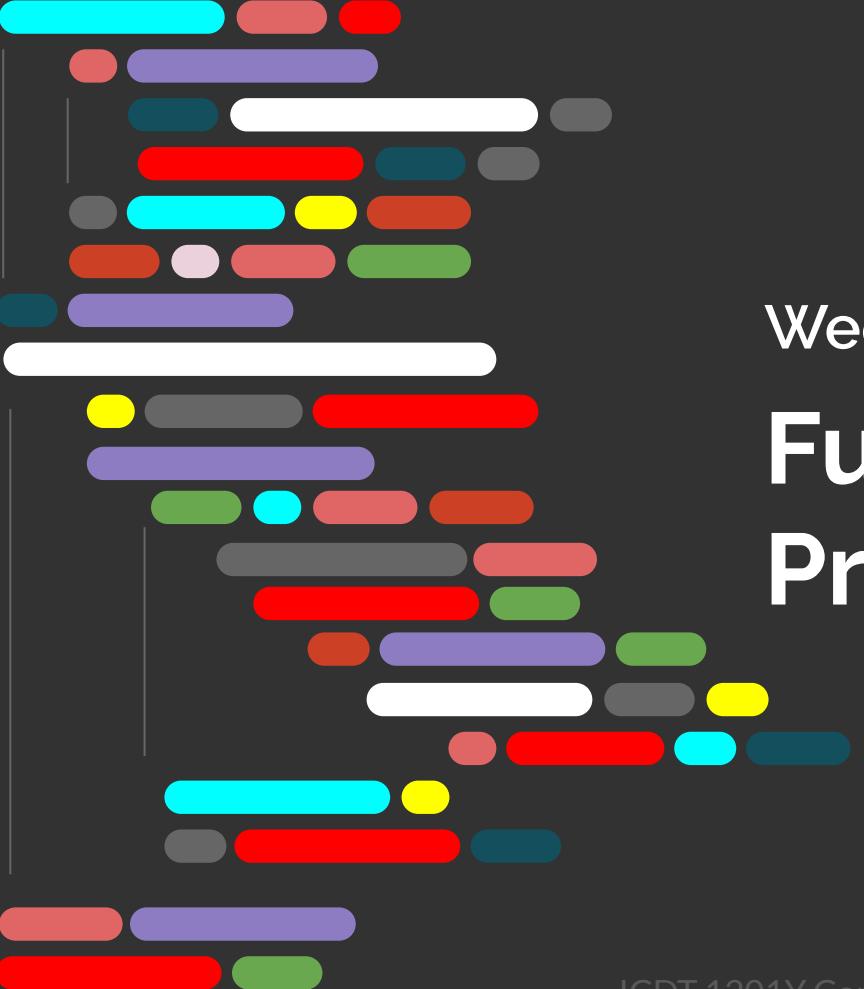
TONY GADDIS

Content taken from
**Starting out with
Python, 6th Ed**





Have fun coding in Python



Week 9

Functions & Modular Programming



Learning Objectives

- Introduction
- Defining and Calling a function
- Parameters and Arguments
- Void functions
- Value Returning functions
- Scope
- Modular Programming
- Modules



Functions

- You are working on an application that requires you to enter the marks for 20 students in four different subjects and calculate the average mark for each student and output their corresponding grade.
- How do you go about solving it?



Functions

- A function is a group of statements within a program that performs a specific task.
 - Usually one task of a large program
- Modularized Programming: program wherein each task in the program is in its own function



Defining and Calling functions

Two steps involved when using functions

1. Define the function
 - Functions are made up of a header and a body
 - This entails implementation of the function
2. Call the function
 - If the function is not called, it never executes.

Defining a function

- Functions are made up of a function header and function body/block:

```
def function_name() : ← header
    statement
    statement ← body
```



Conventions when defining a function name

- Name should be meaningful and descriptive of the task at hand.
- Cannot use keywords
- Cannot contain spaces
- First character needs to be a letter or underscore
- All other characters must be a letter, number or underscore.



Calling a function

- To execute a function it needs to be called/invoked.
- We call a function by writing the **name** of a function, followed by the list of parameters if any.
- When a function is called, the interpreter jumps to the function and executes the body
- After function has executed, interpreter jumps back to the line in the program where the function was called.

Calling a function

```
1 1
2 #defining and implementing the function
3 #to print the sum of two numbers
4 def calculate_sum(num1, num2):
5     print("Sum of marks:", num1 + num2) ↓
6
7
8 2 mark1 = 68
9 mark2 = 89
10 #calling the function so that it executes
11 calculate_sum(mark1, mark2) ↓ 4
```



Parameters

- The variables defined in the function header, in between parentheses, are called formal parameters.

```
def function_name(param1, param2):  
    statement  
    Statement
```

- A function can have zero, one or multiple parameters.



Actual Parameters, aka, Arguments

- When a function is called, you pass a value to the formal parameter.
- This value is referred to as an actual parameter or argument.
- When invoking a function with **multiple parameters**, arguments need to be **passed by position** to corresponding parameters.
- That is, the first parameter receives value of first argument, the second parameter receives value of second argument, etc.

Default Arguments

In a function definition, you can provide a default argument for a parameter.

```
def show_tax(price, tax_rate=0.07):  
    tax = price * tax_rate  
    print(f'The tax is {tax}.')
```

Default argument



Default Arguments

When we call the function, we must pass an argument for the price parameter, but **we have the option of omitting the argument for the tax_rate parameter.**

```
show_tax(100)
```

In this function call, 100 is assigned to price and 0.07 will be assigned to tax_rate.

Default Arguments

```
show_tax(100, 0.08)
```

The function `show_tax` called as above, passes the value 100 to the `price` parameter and 0.08 to the `tax_rate` variable.



Default Arguments

- In a function's parameter list, the parameters without default arguments must appear first, followed by the parameters with default arguments.
- This is an invalid function:

```
def show_tax(price=10, tax_rate):  
    tax = price * tax_rate  
    print(f'The tax is {tax}.')
```



Default Arguments

- You can also provide default arguments for all of a function's parameters.
- We can call the function without passing any arguments.
- The default values of the parameter will be used during execution of the function body.



Void functions

- Void functions are functions that perform a specific task when called.
- Results are normally printed directly in the function.

E.g. calculate_sum on slide 10



Value Returning function

- A Value Returning function works the same as a void function but also returns a value using the return keyword.
- This implies that when the function finishes execution, the function will return a value to the part of the code that called the function. The returned value is accessible for further processing.
- Functions can return Integer, Float, Boolean and other types of values.

Value Returning functions

```
1
2     # defining and implementing the function
3     # to print the sum of two numbers
4     def calculate_sum(num1, num2):
5         sum = num1+num2
6         return sum
7
8     # calling the function so that it executes
9     marks_sum = calculate_sum(68, 89)
10    # note the use of a variable to hold
11    # the value returned by the function
12    # this way, you can do further processing
13    # with the returned value
14    print(marks_sum)
```



Returning Multiple Values

- In Python, a function can return multiple values.

```
return expression1, expression2
```

- When you call such a function, you need a separate variable on the left to receive each value.



Class Exercise 1

Using void functions, write a program that takes as input a depth in km and computes and displays the temperature inside the earth in Celsius and Fahrenheit. The relevant formulas are:

$$\text{Celsius} = 10 \times (\text{depth}) + 20$$

$$\text{Fahrenheit} = 1.8 \times (\text{Celsius}) + 32$$

Class Exercise 2

Using value-returning functions, write a program that takes as input a depth in km and computes and displays the temperature inside the earth in Celsius and Fahrenheit. The relevant formulas are:

$$\text{Celsius} = 10 \times (\text{depth}) + 20$$

$$\text{Fahrenheit} = 1.8 \times (\text{Celsius}) + 32$$



Scope of Variables

- Scope refers to the part of a program where a variable can be referenced.



Local Scope

- A local variable is declared inside a function and can be accessed/referenced anywhere **inside** the function.
- The scope of a function begins at the creation of a function until the end of the function that contains the variable.
- A local variable cannot be referenced outside the function where it is declared.
- Supports the fact that same variable names can be used in different functions in the same program without any clashes.



Global Scope

- Python allows you to declare global variables.
- Global variables are created outside functions and are accessible to all functions in their scope.



Benefits of Using Functions

- No redundancy/duplication of code
- Enables code reuse
- Increases maintainability of code
- Faster Development
- Good teamwork dynamics



Modular Programming

- With functions came the concept of Modular programming where, **to keep code well organized**, we started **grouping related functions together** to form **Modules**.
- Modules also showed the same benefits as functions.



Storing functions in Modules

- A module is a file that contains function definitions but no function calls.
- It is a python file with .py extension
- Modules need to be imported using the `import` statement in your python file, from where you will also call the functions in modules.



Python Modules/Libraries

- There is a list of python modules or libraries that are available already e.g.
 - Random
 - Math



Import Statement

- To use these libraries/modules you need to install them and then import them into your very own python files.
- Remember that all your import statements should make up the first few lines of your python file.

Call functions from module

Once you have imported the module, you can now invoke/call the functions from the modules in your own python files.

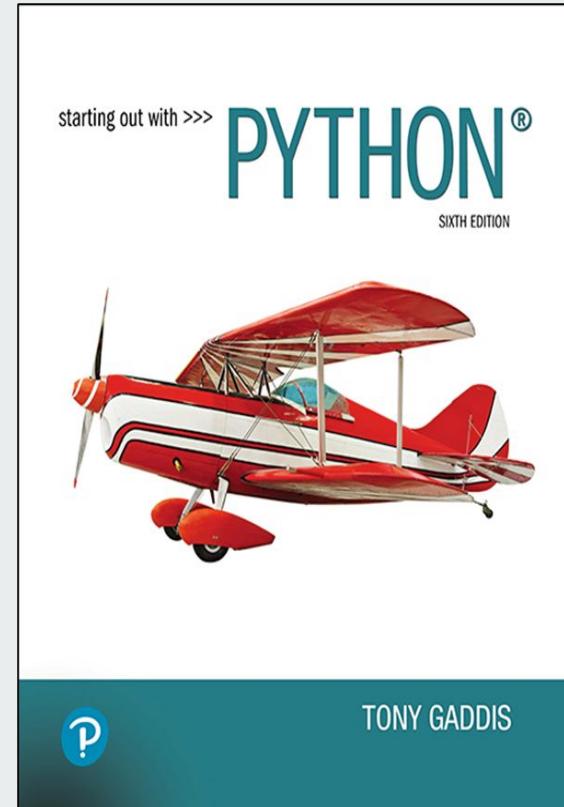
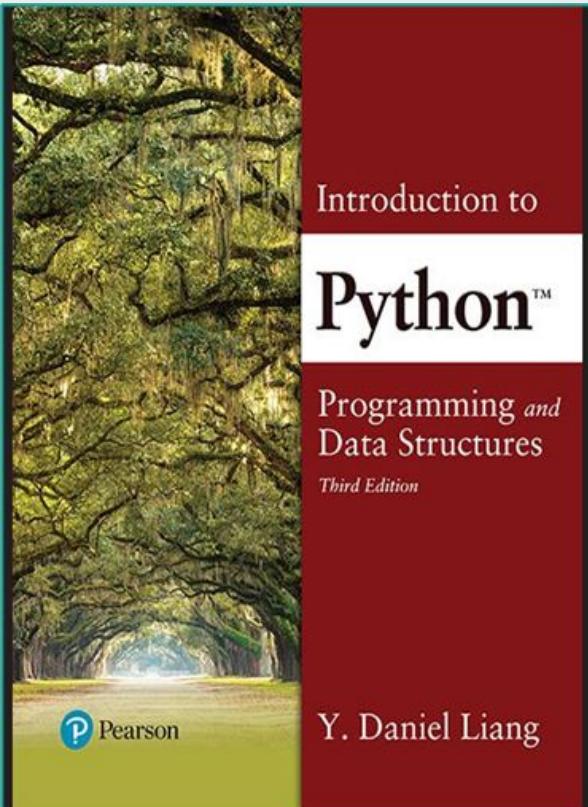
```
1 import random  
2  
3 #generate a random integer between 0 and 100  
4 rand_number = random.randint(0, 100)  
5 print("Randomly generated value = ", rand_number)
```

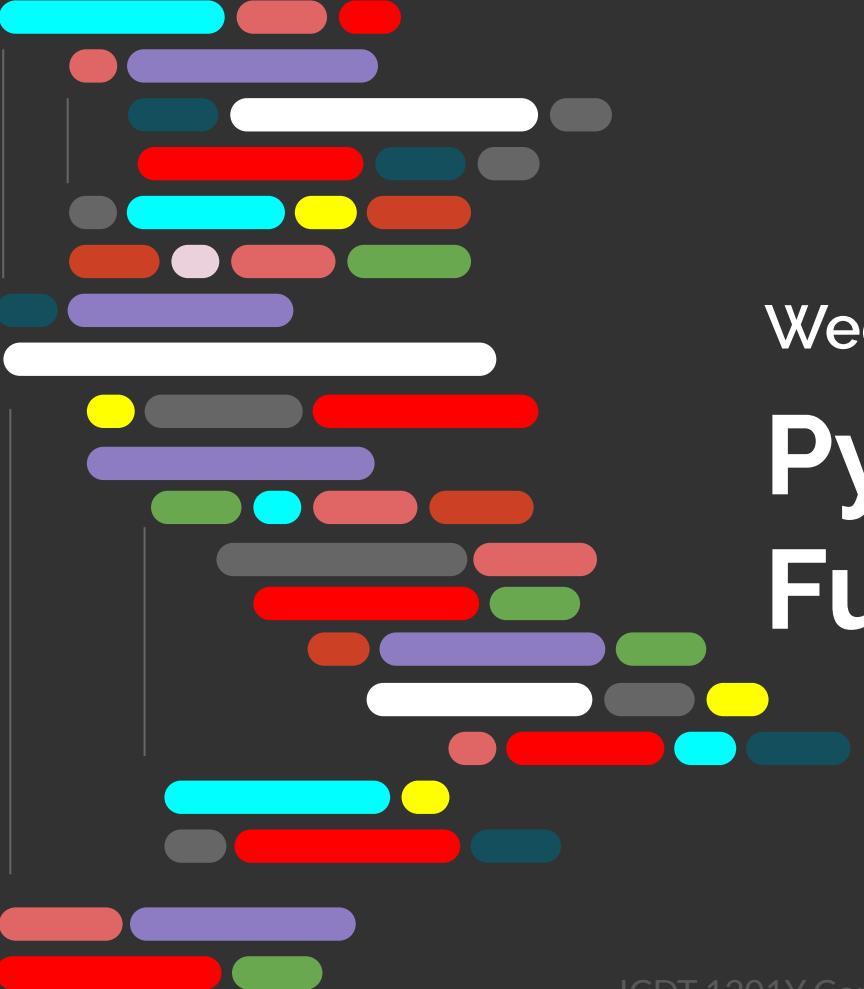


Happy Coding!

You are now set up to create your own functions and build your own modules.

References





Week 10

Python Recursive Functions



Agenda

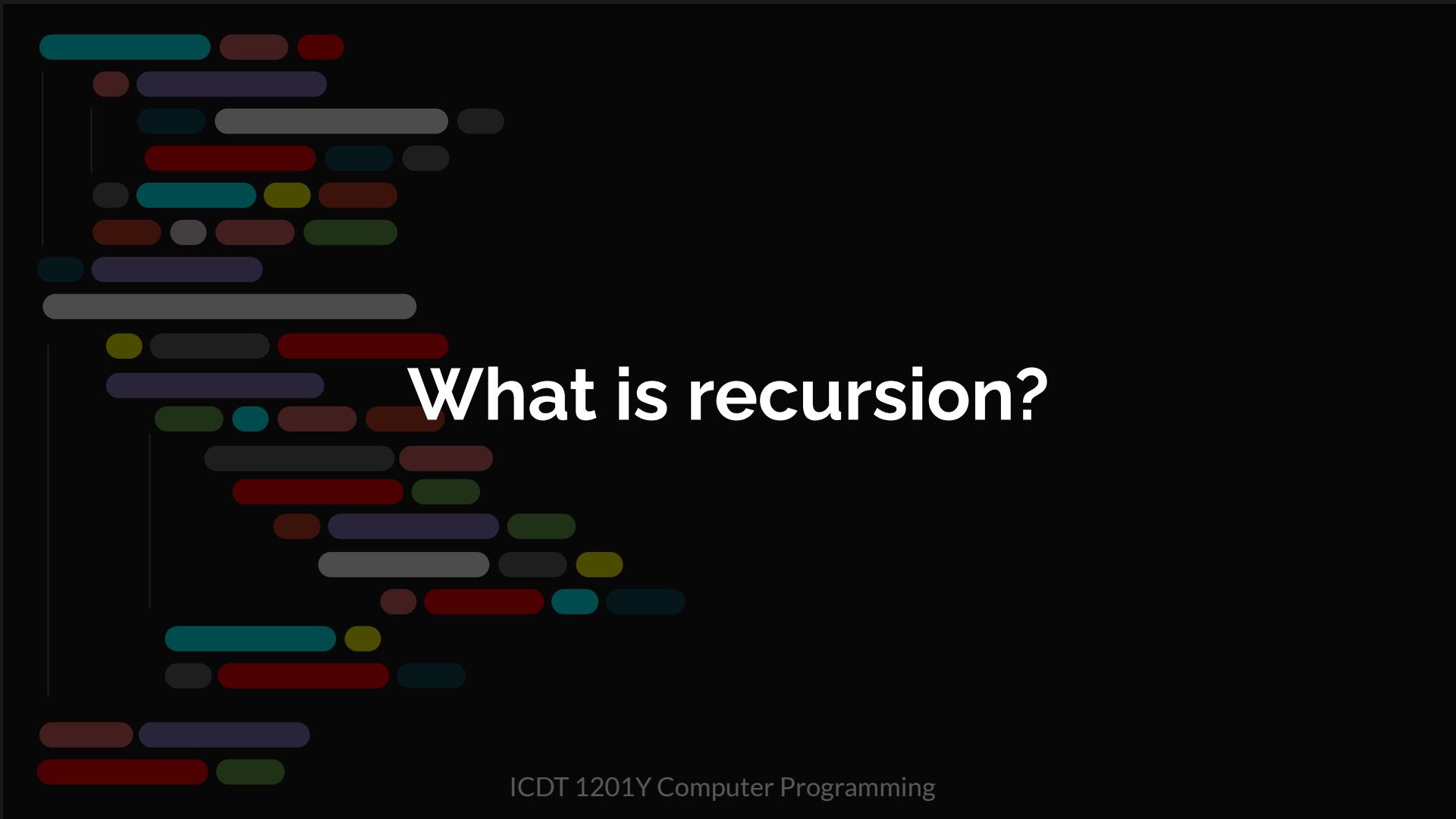
- What is recursion?
- When can recursion be used?
- How to write recursive functions?
- Flow of execution of recursive functions.



Learning Objectives

At the end of this session, you will be able to:

- Understand what is recursion.
- Know when to use recursion.
- Write programs with recursive functions.
- Trace programs with recursive calls.



What is recursion?



Recursion

- It is a process by which a function calls itself repeatedly until some specified condition is satisfied.
- A function that uses recursion is called a **recursive function**.
 - Each action stated in terms of the previous result.
 - Must include a stopping condition.
 - To prevent infinite recursion, you need an if-else statement where one branch makes a recursive call, and the other does not.



Recursion explanation

- Say you want to calculate the factorial of a number:
- The number you want to calculate the factorial of is eventually a positive integer.
- Let us call the number n .
- To calculate the factorial, you need the formula, which is:

$$n! = 1 \times 2 \times 3 \times \dots \times n$$

- The next slide explains how to use recursion for calculating the factorial of a number.

Recursion explanation

Let us call the factorial of a number, n as $\text{fact}(n)$. Therefore,

$$\text{fact}(n) = n! = 1 \times 2 \times 3 \times \dots \times n$$

$$\text{fact}(1) = 1! = 1$$

$$\text{fact}(2) = 2! = 1 \times 2 = 2$$

$$\text{fact}(3) = 3! = 1 \times 2 \times 3 = 6$$

$$\text{fact}(4) = 4! = 1 \times 2 \times 3 \times 4 = 24$$

Can you see a pattern forming?

Recursion explanation

The pattern:

$$\begin{aligned} \text{fact}(1) &= 1! & = 1 \\ \text{fact}(2) &= 2! = 1 \times 2 & = 2 \\ \text{fact}(3) &= 3! = 1 \times 2 \times 3 & = 6 \\ \text{fact}(4) &= 4! = 1 \times 2 \times 3 \times 4 & = 24 \end{aligned}$$

The diagram illustrates the recursive pattern for calculating factorials. It shows four examples: fact(1) = 1!, fact(2) = 2!, fact(3) = 3!, and fact(4) = 4!. Each example is presented as a product of numbers. In fact(2), the number 1 is highlighted in a blue box. In fact(3), the product 1 x 2 is highlighted in an orange box. In fact(4), the product 1 x 2 x 3 is highlighted in a green box. Blue, orange, and green arrows point from the highlighted part of one equation to the next, showing how each term is built upon the previous one.

The factorial of a number consists of multiplying the number with the factorial of the number just before it

Recursion explanation

The pattern:

The factorial of a number consists of multiplying the number with the factorial of the number just before it. We can therefore rewrite the equations as follows:

$$\text{fact}(1) = 1! = \mathbf{1}$$

$$\text{fact}(2) = 2! = \mathbf{1!} \times 2 = \mathbf{2} \rightarrow \text{fact}(2) = 2! = \text{fact}(1) \times 2$$

$$\text{fact}(3) = 3! = \mathbf{2!} \times 3 = \mathbf{6} \rightarrow \text{fact}(3) = 3! = \text{fact}(2) \times 3$$

$$\text{fact}(4) = 4! = \mathbf{3!} \times 4 = \mathbf{24} \rightarrow \text{fact}(4) = 4! = \text{fact}(3) \times 4$$

Recursion explanation

The pattern:

$$\text{fact}(2) = 2! = \text{fact}(1) \times 2 \rightarrow \text{fact}(2) = 2! = \text{fact}(2-1) \times 2$$

$$\text{fact}(3) = 3! = \text{fact}(2) \times 3 \rightarrow \text{fact}(3) = 3! = \text{fact}(3-1) \times 3$$

$$\text{fact}(4) = 4! = \text{fact}(3) \times 4 \rightarrow \text{fact}(4) = 4! = \text{fact}(4-1) \times 4$$

We can therefore safely
write:

$$\text{fact}(n) = n! = \text{fact}(n-1) \times n$$

Recursion explanation

We end up with:

$$\text{fact}(n) = n! = \text{fact}(n-1) \times n$$

Let us see how to write a function to calculate the factorial of a number, n .

We are going to write the function using two ways:

- Using loop
- Using recursion.



Writing function to calculate factorial

Using loop:

```
def fact(n):  
    f = 1  
    while n > 0:  
        f = f * n  
        n = n - 1  
    return f
```

Writing function to calculate factorial

Using recursion:

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

Recall the formula we ended up with!



Recursive function to calculate factorial

How does it work? In the recursive function, there are the following features:

- A Base Case
 - Defines the stopping condition
- Smaller-caller Case
 - Function calls smaller version of itself
- General Case
 - If the recursive function works properly, the problem will be solved



Recursive function to calculate factorial

In the `fact(n)` recursive function, there is:

- The Base Case (stopping condition):
 - The function needs to stop at a condition. In our case, this is when $n=0$.
 - Since $\text{fact}(0) = 1$, this implies that we can use $n==0$ as the base case.
- Smaller-caller Case
 - Function calls smaller version of itself
 - $\text{fact}(n) = \underline{\text{fact}(n-1)} * n$
- General Case
 - If the recursive function works properly, the problem will be solved



A **recurrence relation** is an equation that recursively defines a sequence, once one or more initial terms are given. Each further term of the sequence is defined as a function of the preceding terms.

Factorial Recursive Function: Execution

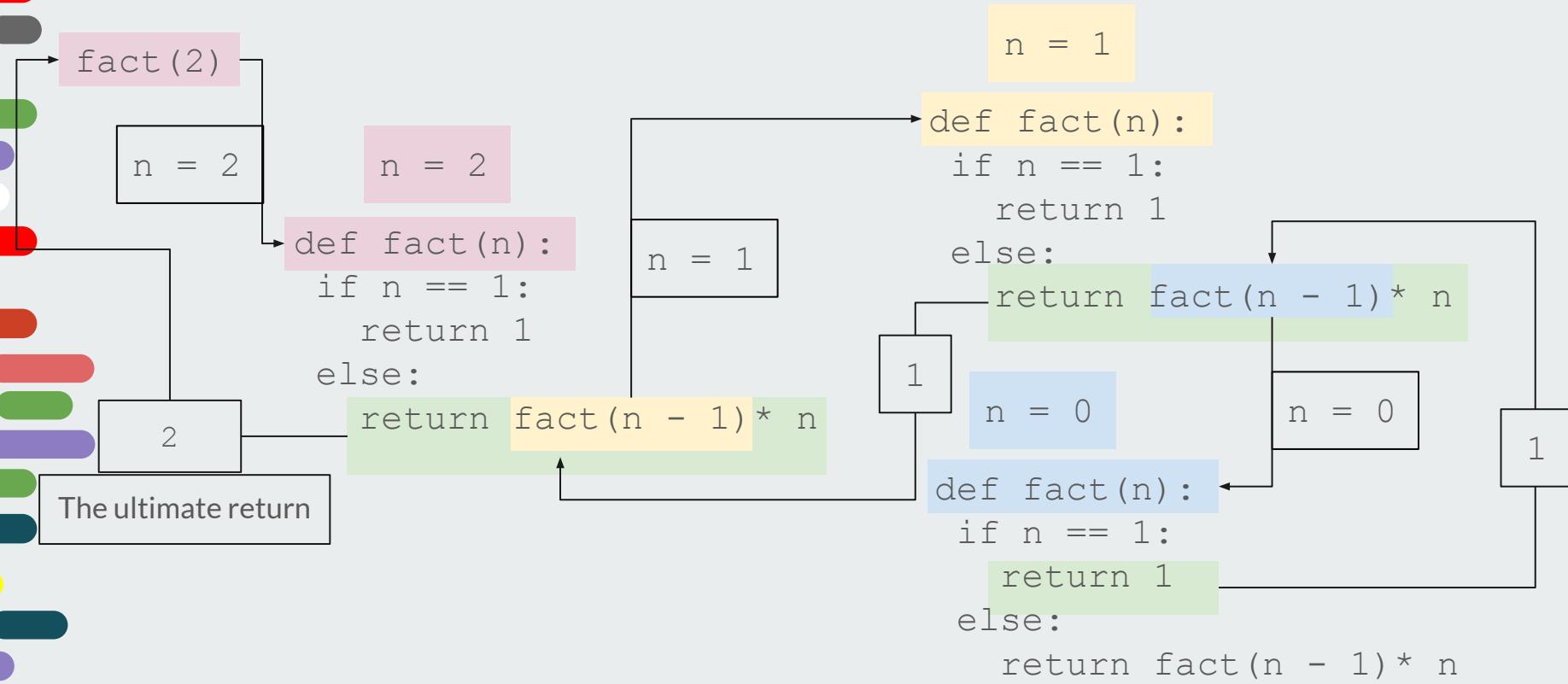
Explanation

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

Base Case
(Stopping case):
Stops the recursion

In the return part of the code, the function calls itself back (Smaller-caller Case) with $(n-1)$ parameter. This will continue happening recursively until the value of n reaches 0 which will trigger the base case (stopping case).

Factorial Recursive Function: Execution





Recursive Calls

- Recursive call
 - Records the location to which it will return to.
 - Re-enters the function code at the beginning.
 - Allocates memory for local data for this new invocation of the function.
- In the factorial function, the parameter argument is the only local variable

Activity

Write a function 'computerPower' that raises any integer value 'x' to an arbitrary power 'n' using:

- Iteration (loop)
- Recursion

Iterative $2^5 = 2 * 2 * 2 * 2 * 2$ (5 times)

Iterative Power Function

```
def computePower(x, n):  
    ans = 1;  
    for i in range(n):  
        ans = ans * x  
    return ans
```

Recursive Power Function

```
def computePower(x, n):  
    if n == 0:  
        return 1;  
    else:  
        return x * computePower(x, n-1)
```

Exercise

Write the recursive function for fibonacci, which can be defined by the following recurrence relations:

- $\text{fibonacci}(0) = 1$
- $\text{fibonacci}(1) = 1$
- $\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$



Recursive function - Countdown

Write a recursive program that displays the countdown sequence from a number n.

E.g., n = 5

Display:

5, 4, 3, 2, 1



ICDT1201Y - Week 11

String Manipulation



Learning Objectives

- Strings
- Iteration
- Repetition and Concatenation
- Slicing
- String Methods
- Splitting



Strings

- print("This is a string")
- Strings are a **sequence of characters**
- Characters are all alphabets(upper and lower), numbers, symbols and spaces
- Order of the characters matter, so a string is referred to as a **sequence**
- Many of the functions and methods that work with sequences work with strings



Accessing Individual Characters in a String

- *stringtext* = "this is a string"
- Iteration
 - Using a for loop

```
for character in stringtext:
```
- Indexing
 - Each character has an index depending on its position in the string. Indices start from 0.

```
character = stringtext[2]
```



Accessing Individual Characters in a String

- IndexError exception
 - Happens when you use an index that is out of range for the string
- `len(stringtext)`
 - Function **len** can be used to obtain the length of a string so as to prevent loops from iterating beyond the end of a string.



String Concatenation

- Appending one string to the end of another string
 - Use the `+` operator to produce a string that is the combination of its operands
 - augmented assignment operator `+=` can also be used to concatenate strings



Strings are immutable

- Once they are created, they cannot be changed
- You cannot change assign new values for a string
- `string[index] = new_character` will raise an exception
- **Concatenation doesn't actually change the existing string, but rather creates a new string and assigns the new string to the previously used variable**

Slicing

- Slice: span of items taken from a sequence, known as substring
- Used previously when slicing lists

string[start : end]

- Expression will return a string containing a copy of the characters from *start* up to, but not including, *end*
 - If *start* not specified, 0 is used for start index
 - If *end* not specified, `len(string)` is used for end index
 - Slicing expressions can include a step value and negative indexes relative to end of strings



Manipulating Strings

- Searching
- Search and Replace
- Testing
- Modification
- Repetition
- Splitting



String Methods

- Strings in Python have several different methods which can be categorised into the types of operations mentioned in the slide above.
mystring.method(arguments)
- Some methods test for specific characteristics and will therefore return true if a condition is True or False otherwise.



Search and Replace Strings

Searching

- Use the `in` / `not in` operators to determine whether a string is or is not contained in another string

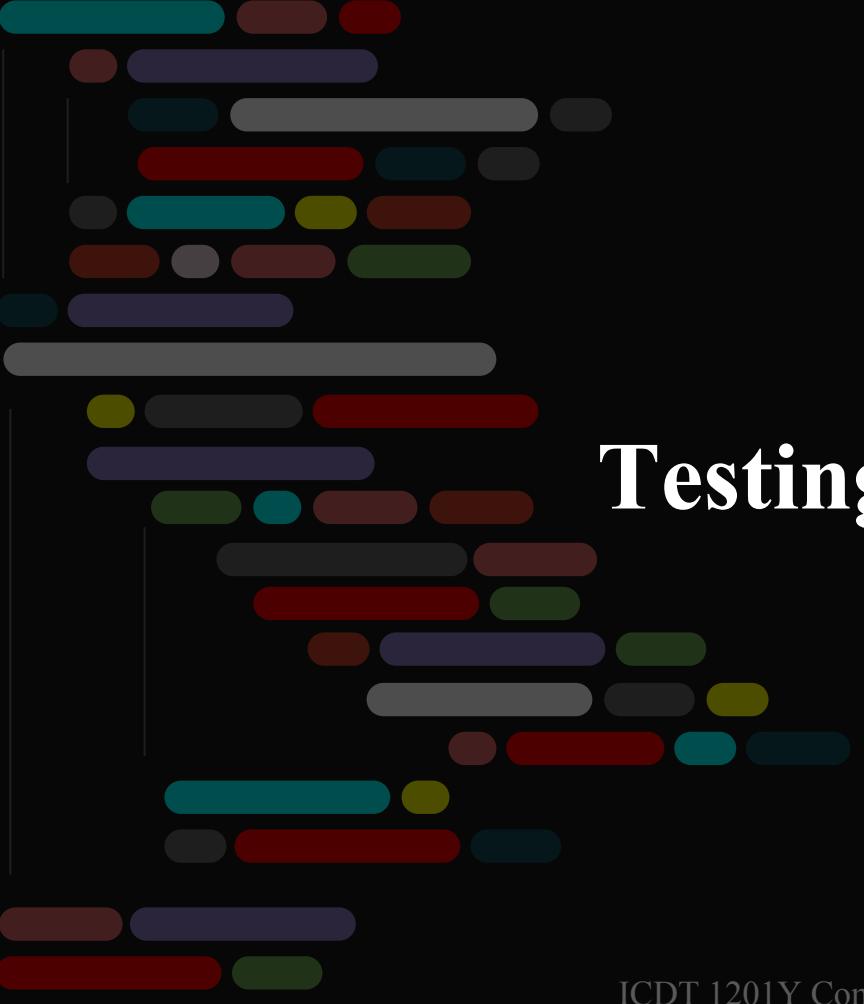
string1 in string2

```
sentence = "I love programming in Python"  
is_present = "Python" in sentence
```

`is_present` will contain the boolean value: True

Search and Replace Methods

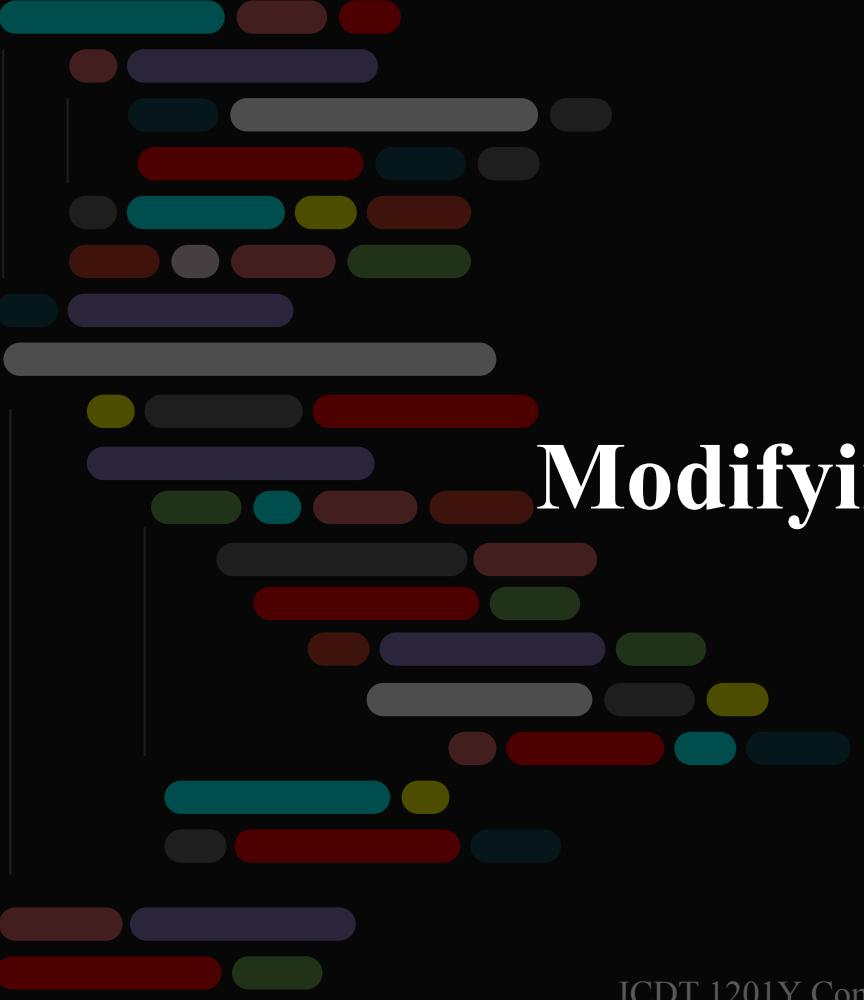
| Method | Description |
|--------------------------------------|--|
| <code>endswith (substring)</code> | The <i>substring</i> argument is a string. The method returns true if the string ends with <i>substring</i> . |
| <code>find (substring)</code> | The <i>substring</i> argument is a string. The method returns the lowest index in the string where <i>substring</i> is found. If <i>substring</i> is not found, the method returns ... |
| <code>replace (old, new)</code> | The <i>old</i> and <i>new</i> arguments are both strings. The method returns a copy of the string with all instances of <i>old</i> replaced by <i>new</i> . |
| <code>startswith (substring)</code> | The <i>substring</i> argument is a string. The method returns true if the string starts with <i>substring</i> . |



Testing Strings

Testing Methods

| Method | Description |
|-----------|--|
| isalnum() | Returns true if the string contains only alphabetic letters or digits and is at least one character in length. Returns false otherwise. |
| isalpha() | Returns true if the string contains only alphabetic letters and is at least one character in length. Returns false otherwise. |
| isdigit() | Returns true if the string contains only numeric digits and is at least one character in length. Returns false otherwise. |
| islower() | Returns true if all of the alphabetic letters in the string are lowercase, and the string contains at least one alphabetic letter. Returns false otherwise. |
| isspace() | Returns true if the string contains only whitespace characters and is at least one character in length. Returns false otherwise. (Whitespace characters are spaces, newlines (\n), and tabs (\t).) |
| isupper() | Returns true if all of the alphabetic letters in the string are uppercase, and the string contains at least one alphabetic letter. Returns false otherwise. |



Modifying Strings

String Modification Methods

| Method | Description |
|-----------------------|---|
| lower() | Returns a copy of the string with all alphabetic letters converted to lowercase. Any character that is already lowercase, or is not an alphabetic letter, is unchanged. |
| lstrip() | Returns a copy of the string with all leading whitespace characters removed. Leading whitespace characters are spaces, newlines (\n), and tabs (\t) that appear at the beginning of the string. |
| lstrip(<i>char</i>) | The <i>char</i> argument is a string containing a character. Returns a copy of the string with all instances of <i>char</i> that appear at the beginning of the string removed. |
| rstrip() | Returns a copy of the string with all trailing whitespace characters removed. Trailing whitespace characters are spaces, newlines (\n), and tabs (\t) that appear at the end of the string. |
| rstrip(<i>char</i>) | The <i>char</i> argument is a string containing a character. The method returns a copy of the string with all instances of <i>char</i> that appear at the end of the string removed. |
| strip() | Returns a copy of the string with all leading and trailing whitespace characters removed. |
| strip(<i>char</i>) | Returns a copy of the string with all instances of <i>char</i> that appear at the beginning and the end of the string removed. |
| upper() | Returns a copy of the string with all alphabetic letters converted to uppercase. Any character that is already uppercase, or is not an alphabetic letter, is unchanged. |



String Modification Methods

- Remember strings are immutable.
- The methods listed on the previous slide all return a **copy of the string, to which modifications have been made.**
- String comparisons are case-sensitive
 - Uppercase characters are distinguished from lowercase characters.
 - “Text” is not equal to “text”

Repetition

Repetition Operator

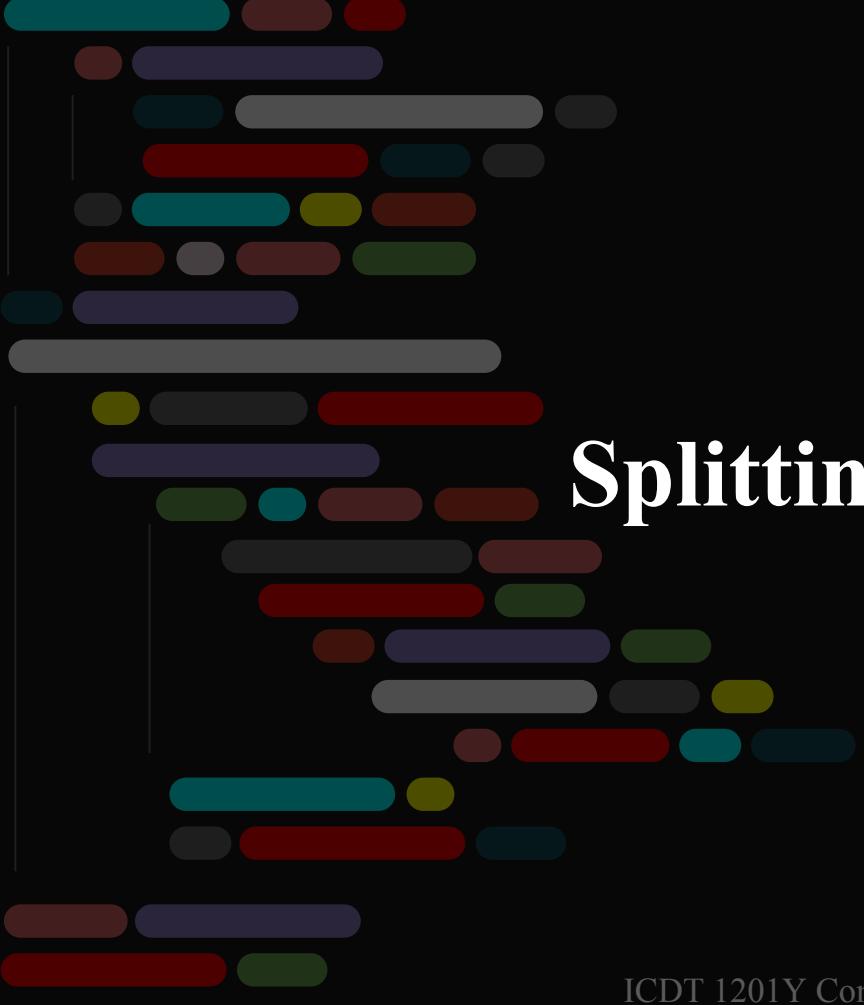
- Repetition operator: makes multiple copies of a string and joins them together
- The * symbol is a repetition operator when applied to a string and an integer
 - String is left operand; number is right

*string_to_copy * n*

```
text = "s" * 4
```

```
print(text)
```

Output: ssss



Splitting a string



Split a string

- **split method:** breaks the string into a list of character/substrings based on the separator/delimiter.
 - By default, uses space as separator
 - Can specify a different separator by passing it as an argument to the `split` method

Examples

```
>>> my_string = 'One two three four'  
>>> word_list = my_string.split()  
>>> word_list  
['One', 'two', 'three', 'four']  
>>>
```

```
>>> my_string = '1/2/3/4/5'  
>>> number_list = my_string.split('/')  
>>> number_list  
['1', '2', '3', '4', '5']  
>>>
```



String Tokens

- **Tokenizing** is the process of breaking a string into tokens
- When you tokenize a string, you extract the tokens and store them as individual items
- In Python you can use the `split` method to tokenize a string

String Tokens

```
>>> my_string = '1/2/3/4/5'  
>>> number_list = my_string.split('/')  
>>> number_list  
['1', '2', '3', '4', '5']  
>>>
```

String contains the 1, 2, 3, 4 and 5 tokens and the delimiter is /

Example

```
>>> my_address = 'www.example.com'  
>>> tokens = my_address.split('.')>>> tokens  
['www', 'example', 'com']  
>>>
```

Activity

Write Python code for the below:

1. Create a variable breakfast and assign it string value “egg”
2. Print the value of your variable in all uppercase characters
3. Print the value of your variable in all lowercase characters
4. If the value of your variable starts with “eg”, print “egg start with eg”
5. Print a new variable which stores the value “egg” repeated 6 times
6. Variable “txt” is assigned value “I like bananas”. Replace “bananas” with “apples” and print the new sentence.



Summary

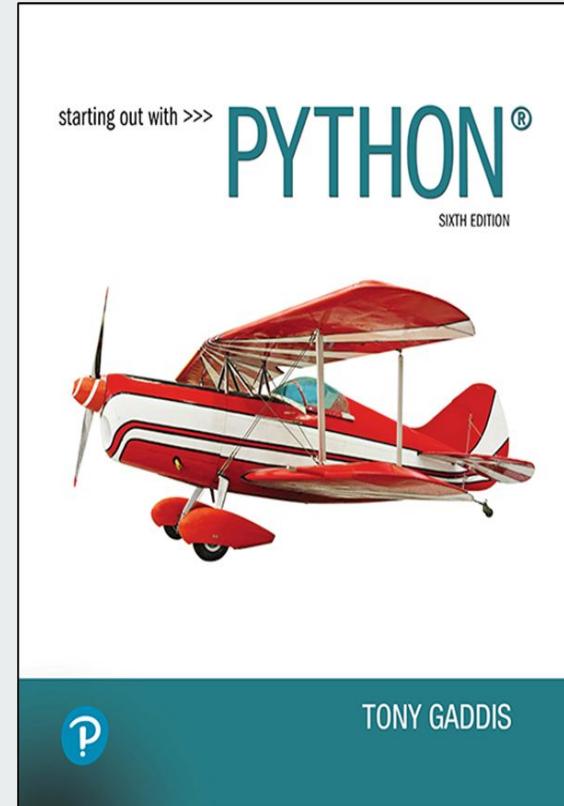
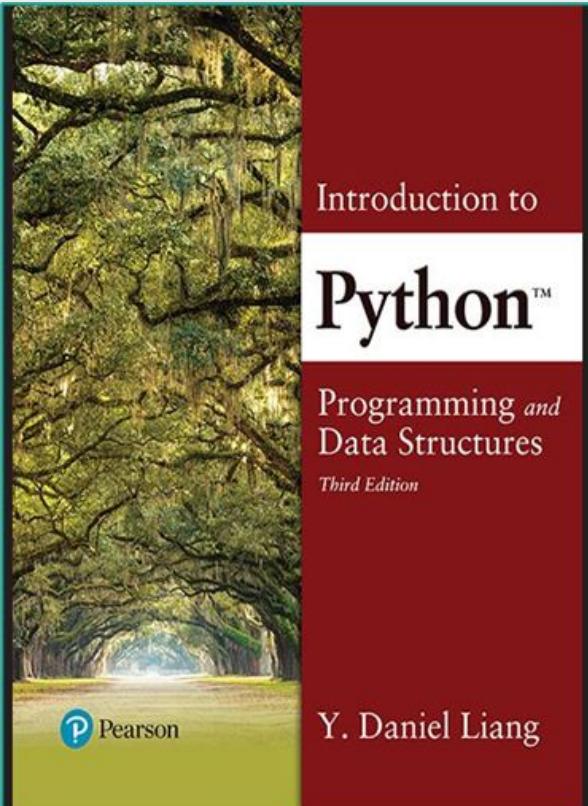
- **This chapter covered:**
 - String operations, including:
 - Methods for iterating over strings
 - Repetition and concatenation operators
 - Strings as immutable objects
 - Slicing strings and testing strings
 - String methods
 - Splitting a string



Assessment Criteria

- Ability to do string operations using String methods,
- Slicing,
- Repetition and Concatenate operators

References





Week 12
ICDT1201Y – Computer
Programming

Program Testing

ICDT 1201Y Computer Programming



Learning Objectives

By the end of this lecture, students should be able to:

- Understand the need for program testing
- Differentiate between different types of testing
- Understand the difference between manual testing and test automation
- Understand how to use **Python Unittest**



Agenda

- Software Testing
- Static Testing
- Dynamic Testing
- Debugging Python codes



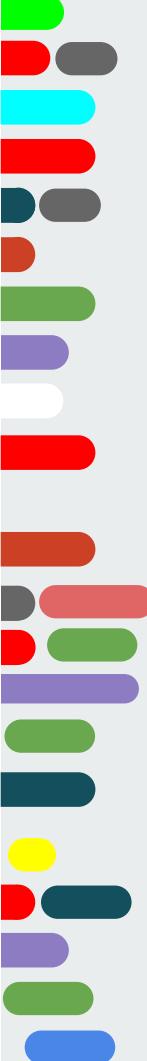
Software Testing

- Testing is an important phase in software development. The goal of testing is assess the functionality and behavior of a software program.
- Software testing can be categorized as:
 - Static testing
 - Dynamic testing



Static Testing

- Static testing is the review of a program's code and documentation. Static testing is performed without having the code getting executed.
- Popular forms of static testing:
 - Peer review
 - Peer code review is where another person or a group of person in the development team read and assess the code written by another development in view to improve quality.
 - Code Analysis Tools
 - Code analysis tools are other software that takes as input source code and evaluates it against some defined set of rules. These rules matches standards and good practices.



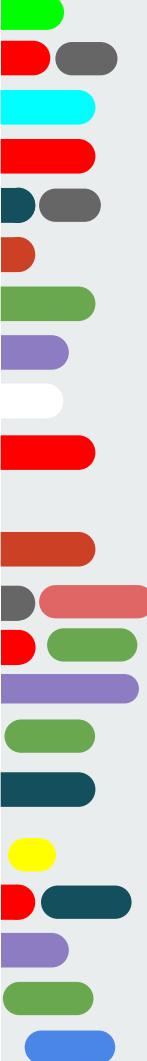
Dynamic Testing

- Dynamic testing is the evaluation of the software with corresponding input data while running the software.
- Dynamic testing examines the output with given input values by executing test cases.
- Different forms of dynamic testing are **Unit tests, integration tests, system tests/functional test, acceptance tests, performance tests, security tests.**
- Dynamic testing can be done through **manual testing or test automation.**
- Dynamic testing includes: **White box testing & Black box testing**



Unit Testing

- The earlier a bug is detected, the less costlier it is to correct it. Unit Testing is the first dynamic testing that happens during the development of a software, and therefore it is very crucial.
- Unit testing involve testing the good functioning of small pieces of source code with associated data to determine whether each of these piece of codes is working as expected.



Unit Testing

- Unit testing is often performed by writing dummy programs that calls functions with different types of test data to check the output of the functions.
- Three types of test data exist:
 - Normal data
 - Extreme data
 - Abnormal data



Normal test data

- Normal test data are data that real users will typically enter in the system for processing.
- The system would normally accept normal test data, process it, and give corresponding output which can be checked against expected output.
- Ideally enough normal test data should be used to have all control flows in the program run at least once.
- Example: for a program that accepts marks as percentages, normal test values would include 10, 35, 50, 65, 90.



Extreme test data

- Extreme test data are normal data but at the absolute limit of the normal range.
- Very often extreme test data shows incorrect processing, and needs code rework.
- With extreme test data, the system should behave normally; that is, process the data and give expected output.
- Example: for a program that accepts marks as percentages, extreme test values would be 0 and 100.



Abnormal test data

- Abnormal test data are data that should not normally be accepted by the system - the values are invalid.
- The system should reject any abnormal test data.
- Abnormal values are used in testing to make sure that invalid data does not break the system.
- Example: for a program that accepts marks as percentages, abnormal test values would include -1, 101, 210, -13.



Testing Tools

- Testing tools are software that helps ease developers/testers in performing the testing process.
- Testing tools range from management tools to software that can analyze codes provide recommendations.
- Unit testing tools can be used to program unit test cases with different inputs and check for expected results.



Testing Tools

- Unit testing tools allow testing of multitude of possibilities without much human intervention.
- Test scripts generated from testing tools can be re-executed multiple times to ensure functionalities continue to work correctly after code changes.
- These test scripts can be made to run automatically when there are changes in code or periodically (Test Automation) by performing some configurations in other software (e.g. in the compiler).



Test Plans/Test Cases

- Before embarking in Unit Testing, it is good practice to first have:
 - Unit Test Plan
 - A document that describes at high level how to carry the unit test. It normally includes the functionality to be tested, the roles and responsibilities of the stakeholders, description of the testing environment and the assumptions.
 - Unit Test Cases
 - A unit test case is a scenario on how to test one part of the program. Unit tests should include a complete set of unit test cases that cover all different possibilities.



Unit Test Cases

- Unit Test Cases would normally include the following data (+ any other information that may be useful):
 - Test Case Number
 - Test Case Description
 - Execution Steps
 - Input Data
 - Expected Result
 - Tested by
 - Test Date
 - Actual Result
 - Remarks



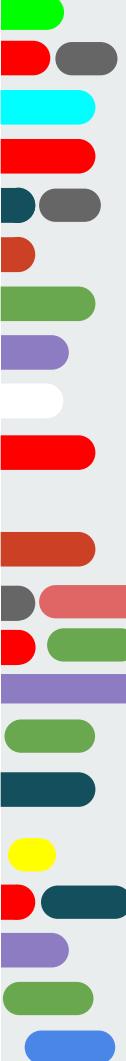
Example

- Consider a login screen that checks the username and password of a user. In case the username is unknown, an error message is displayed “Unknown user”. In case the password does not match a username, “Incorrect Password” error message is displayed. For a successful login, the system navigates to a main screen. The screen has a username textbox, password textbox and a login button.



Example – Test Cases

| # | Description | Execution Steps | Input Data | Expected Result | Tested by | Test Date | Result (OK/KO) | Remarks |
|---|--------------------------------------|--|--------------------------------------|------------------------------------|--|-----------|----------------|---------|
| 1 | No data entered | Press on login-button without entering any data | Username: "" Password: "" | Error message "Unknown User" | | | | |
| 2 | Incorrect user | Enter an incorrect username and press login-button | Username: "Paul" Password: "" | Error message "Unknown User" | | | | |
| 3 | Correct user with incorrect password | Enter a correct username with an incorrect password and press login-button | Username: "John" Password: "abc" | Error message "Incorrect Password" | | | | |
| 4 | Valid username and password | Enter a valid username and password and press login-button | Username: "John" Password: "Phnx" | Main screen is displayed | *note: - a valid username/password: John/Phnx - an invalid username: Paul | | | |



Classwork

Consider a program that allows a test mark and an exam mark of a student to be entered, and the program displays the grades. The test mark can range from 0 to 30 and the exam mark can range from 0 to 70. The system should not allow numbers outside these range and should display error messages. Once the marks are entered, the total mark is computed by adding the test mark with the exam mark.



Classwork

The grade is then computed as follows.

| Total Marks | Test Marks | Exam Marks | Grade |
|------------------|-------------|-------------|-------|
| $X < 40$ | - | - | F |
| $40 \leq X < 50$ | - | - | E |
| $50 \leq X < 60$ | - | - | D |
| $60 \leq X < 65$ | - | $X \leq 50$ | D |
| $60 \leq X < 65$ | - | $X > 50$ | C |
| $65 \leq X < 70$ | - | $X \leq 50$ | C |
| $65 \leq X < 70$ | - | $X > 50$ | B |
| $X \geq 70$ | $X < 15$ | - | B |
| $X \geq 70$ | - | $X \leq 50$ | B |
| $X \geq 70$ | $X \geq 15$ | $X > 50$ | A |

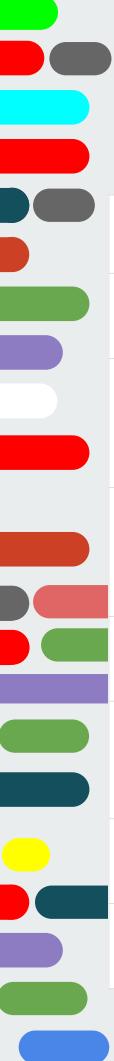
Write test cases for this program

ICDT 1201Y Computer Programming



Python Unittest

- a built-in testing framework that provides a set of tools for testing the code's functionality in a systematic way.
- create test cases, fixtures, and suites to verify if the code behaves as expected.
- unittest requires no additional installations
- Alternatives: Pytest & Nose



Assert Methods in Python Unittest

| Method | Description |
|--------------------------------------|---|
| <code>.assertEqual(a, b)</code> | Checks if a is equal to b, similar to the expression <code>a == b</code> . |
| <code>.assertTrue(x)</code> | Asserts that the boolean value of x is True, equivalent to <code>bool(x)</code> is True. |
| <code>.assertIsInstance(a, b)</code> | Asserts that a is an instance of class b, similar to the expression <code>isinstance(a, b)</code> . |
| <code>.assertIsNone(x)</code> | Ensures that x is None, similar to the expression <code>x is None</code> . |
| <code>.assertFalse(x)</code> | Asserts that the boolean value of x is False, similar to <code>bool(x)</code> is False. |
| <code>.assertIs(a, b)</code> | Verifies if a is identical to b, akin to the expression <code>a is b</code> . |
| <code>.assertIn(a, b)</code> | Checks if a is a member of b, akin to the expression <code>a in b</code> . |

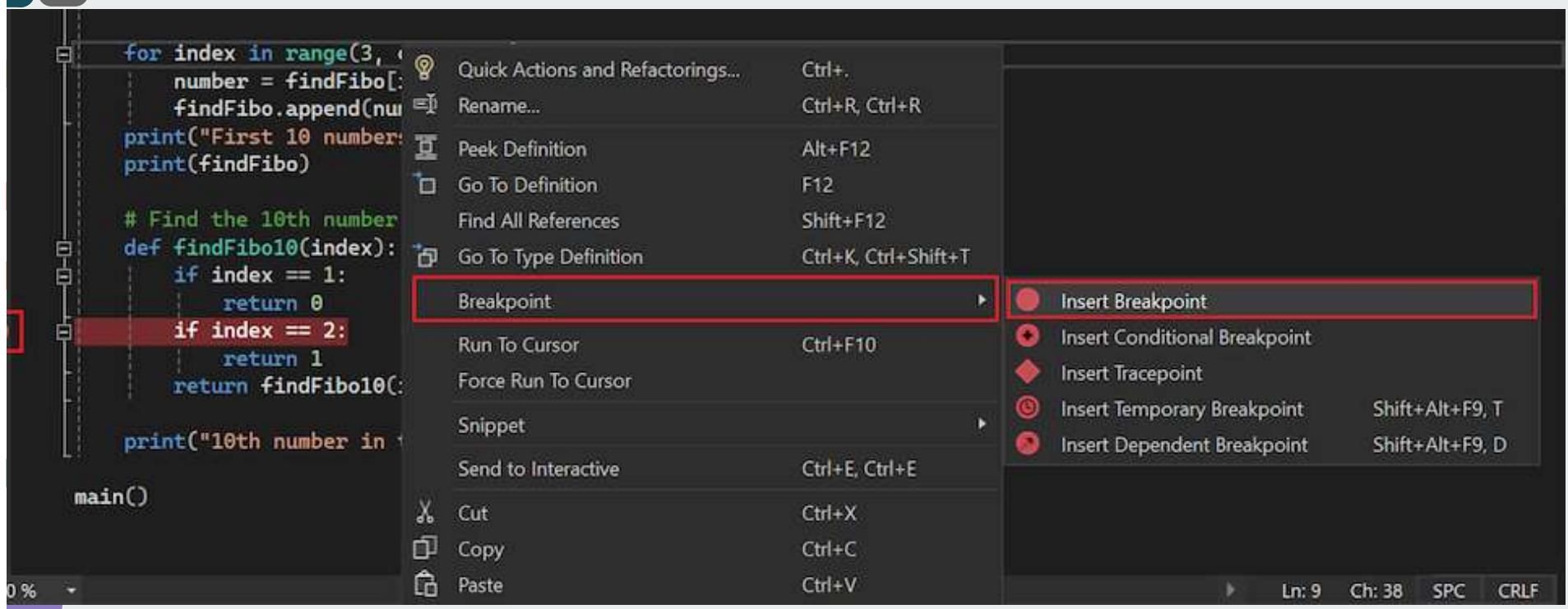


Debugging

- Debugger always starts with the active Python environment for the project.
- **Set Breakpoints:** Breakpoints stop execution of code at a marked point so you can inspect the program state.
- Breakpoints can be inserted or deleted
- **Set conditions and actions:** You can customize the conditions under which a breakpoint is triggered, such as breaking only when a variable is set to a certain value or value range.
- **Step through code:** When Visual Studio Code stops code execution at a breakpoint, there are several commands you can use to step through your code or run blocks of code before breaking again.



Breakpoints



A screenshot of a code editor showing a context menu for a breakpoint. The menu is open over the line of code:

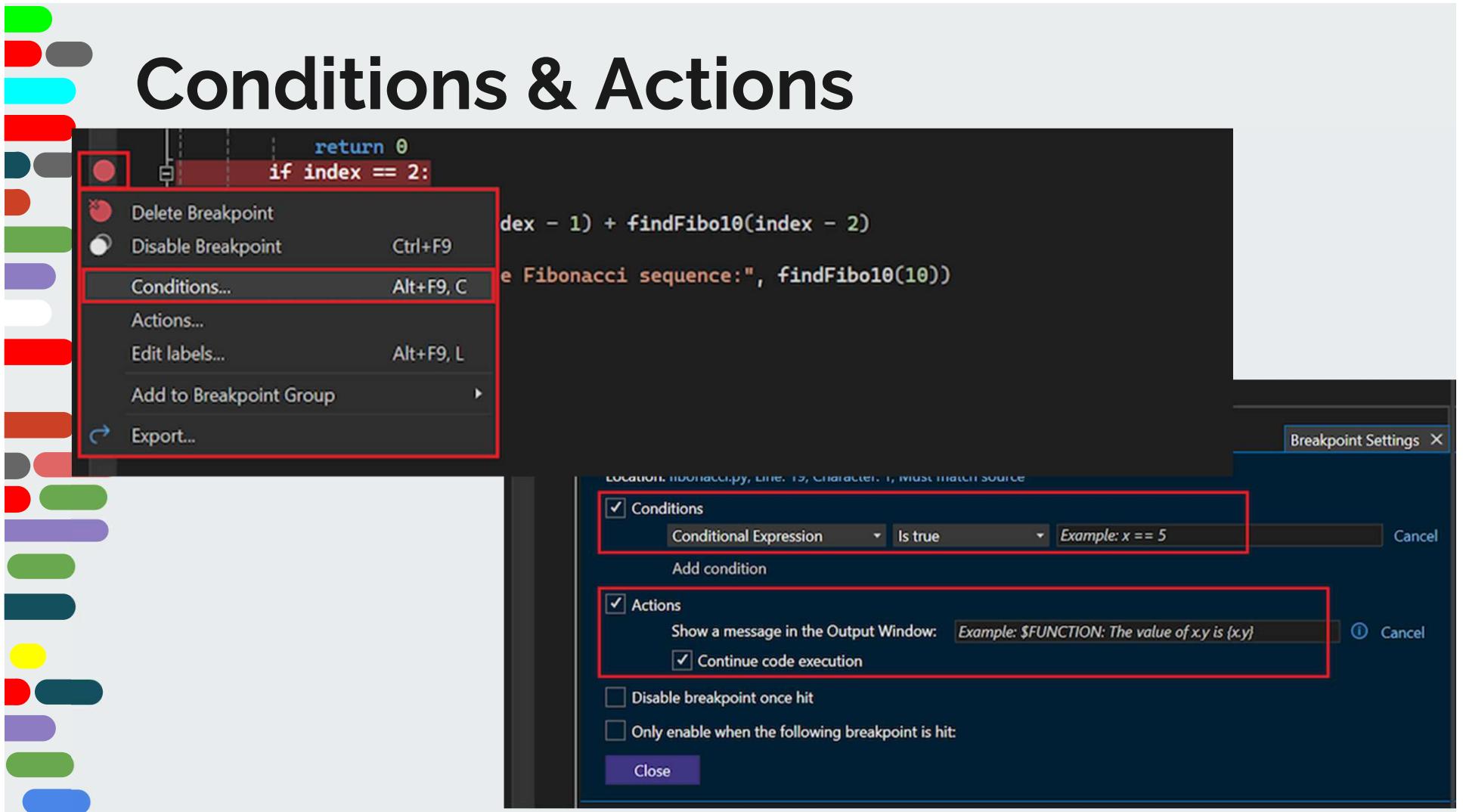
```
for index in range(3, 10):
    number = findFibo()
    findFibo.append(number)
print("First 10 numbers")
print(findFibo)

# Find the 10th number
def findFibo10(index):
    if index == 1:
        return 0
    if index == 2:
        return 1
    return findFibo10(index - 1)

print("10th number is " + str(findFibo10(10)))
```

The context menu is expanded from the line containing the second conditional statement. The "Breakpoint" submenu is highlighted with a red border, and the "Insert Breakpoint" option is also highlighted with a red border. Other options in the submenu include "Insert Conditional Breakpoint", "Insert Tracepoint", "Insert Temporary Breakpoint", and "Insert Dependent Breakpoint". The main menu items include "Quick Actions and Refactorings...", "Rename...", "Peek Definition", "Go To Definition", "Find All References", "Go To Type Definition", "Run To Cursor", "Force Run To Cursor", "Snippet", "Send to Interactive", "Cut", "Copy", and "Paste".

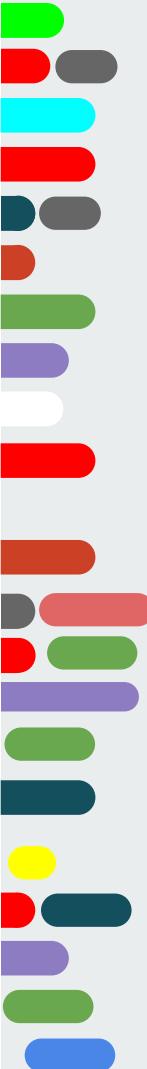
Conditions & Actions





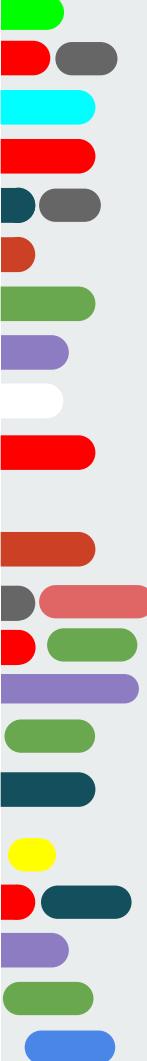
Debugging commands

| Command | Shortcut | Description |
|-----------|-------------------|--|
| Stop | Shift + F5 | Stop the debugging session. |
| Restart | Ctrl + Shift + F5 | Restart the current debugging session. |
| Continue | F5 | Run code until you reach the next breakpoint. |
| Step Into | F11 | Run the next statement and stop. If the next statement is a call to a function, the debugger stops at the first line of the called function. |
| Step Over | F10 | Run the next statement, including making a call to a function (running all its code) and applying any return value. This command allows you to easily skip functions that you don't need to debug. |
| Step Out | Shift+F11 | Run the code until the end of the current function, then step to the calling statement. This command is useful when you don't need to debug the remainder of the current function. |



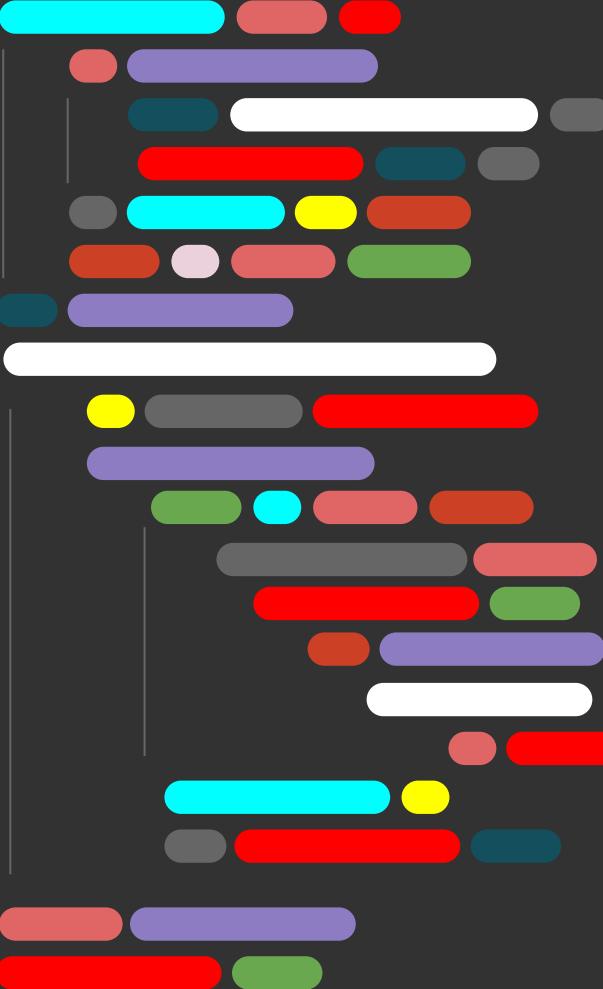
Summary

- In this lecture, students have been exposed to software testing and different types of testing
- Unit testing in Python
- Next lecture → Introduction to OOP



References

- <https://www.geeksforgeeks.org/unit-testing-python-unittest/>
- <https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing>
- <https://learn.microsoft.com/en-us/visualstudio/python/debugging-python-in-visual-studio?view=vs-2022>



Week 13
ICDT1201Y – Computer
Programming

Introduction to Object Oriented Programming



Learning Objectives

By the end of this lecture, students should be able to:

- Understand the Object-Oriented Programming (OOP) paradigm
- Explain key concepts: classes, objects, attributes, and methods
- Understand why OOP is useful
- Write their first basic Python class
- Implement constructors
- Implement methods
- Write a main program to create and modify objects



What is OOP?

- OOP is a programming paradigm based on the concept of objects.
- Objects contain data (attributes) and code (methods).
- An object is like a real-world item (e.g., a car, a phone, a pen).
- Objects have **attributes /features**, i.e., details about the objects (e.g., colour, make, model of the car; price, make, size of the phone, etc).



Objects

- Objects are not limited to representing real-world physical items.
- Abstract concepts, processes, or even logical constructs can be modeled as objects.
- Examples of abstract concepts: bank account, event, etc.
- Examples of processes: orders, tasks, transactions
- Examples of logical constructs: data structures



Methods

- Objects can do things. These are known as behaviours or functions (or methods in terms of programming).
- For example, a car can start, move, brake, stop.
- A phone can make a call, send a message, take a photo, connect to the internet, etc.
- A pen can write, draw, check ink level, etc.



Classes

- Objects which have similar behaviours (functions) and features (attributes) are grouped into Classes.
- For example, in a parking lot, you can have different cars (a red car, a blue car, a white car). All of them belong to the same class (car in this case).
- In Python, the convention is to use singular names for classes (e.g. Car, Person, Book, etc).



Classes

- In terms of programming, classes can be considered as a factory from which objects will be created.
- A class is a blueprint (template) for creating and managing similar objects.
- In Python, the convention for naming classes is to use **PascalCase**, where the first letter of each word in the name is capitalised.
- For example a bank account class would be written as **BankAccount**.



Classes and Objects

Aspect

Class

A **class** is a blueprint or template that defines the attributes and behaviours (methods) for objects. One class can be used to create multiple objects.

A class is like a recipe for baking a cake.

Typically named in **PascalCase** (e.g., Car, BankAccount).

Definition

Object

An **object** is a specific instance of a class.

Example

An object is the actual cake baked using the recipe.

Naming Convention

Can have any valid variable name (e.g., my_car, user_account).



Multiple Choice Question

Question 1: What is a class in Python?

- A. A specific instance of a real-world entity.
- B. A template used to create objects.
- C. A built-in Python data type.
- D. A variable that stores object attributes.



Multiple Choice Question

Question 2: Which of the following correctly describes an object?

- A. A blueprint for creating classes.
- B. A logical concept with no real-world representation.
- C. A specific instance of a class.
- D. A method inside a class.



Multiple Choice Question

Question 3: What happens when you create an object from a class?

- A. The class gets executed and disappears.
- B. The object becomes a copy of the class.
- C. The object is created with attributes and methods defined by the class.
- D. The object stores only the methods from the class, not the attributes.



True/False Questions

1. A class is a blueprint for creating objects, but it does not hold specific attribute values for any object.
2. An object can exist without a class in Python.
3. All objects from the same class share the same methods.
4. All objects from the same class have the same attribute values.



Fill in the blank questions

1. A class acts as a _____ for creating objects.
2. The process of creating an object from a class is called_____.
3. Multiple objects created from the same class will share the same _____ but can have different attribute values.
4. To define a new class in Python, the keyword _____ is used.



How to define a class in Python?

Question 1

- Create a class named Rectangle.
- The class Rectangle has two attributes: width and height.
- The class Rectangle also has three methods: area(), perimeter() and display_info().
- The class must have a constructor and methods for setting and accessing the attributes.

Answer for Question 1

```
1  class Rectangle:
2      def __init__(self, width, height):
3          self._width = width
4          self._height = height
5
6      @property
7      def width(self):
8          return self._width
9
10     @width.setter
11     def width(self, value):
12         self._width = value
13
14     @property
15     def height(self):
16         return self._height
17
18     @height.setter
19     def height(self, value):
20         self._height = value
21
22     @property
23     def area(self):
24         return self._width * self._height
```



Main program to create an object

Question 1 (continues)

- Write a main program to create a rectangle object named **rect** with a width of 4 and a height of 5.
- Print the width, height and area of the rectangle.
- Update the width to 6 and height to 7.
- Print the new width, new height and new area of the rectangle.

The main() program

```
26  def main():
27      rect = Rectangle(4, 5)
28
29      print("Width:", rect.width)    # Access the width
30      print("Height:", rect.height) # Access the height
31      print("Area:", rect.area)    # Access the area
32
33      rect.width = 6              # Update the width
34      rect.height = 7             # Update the height
35
36      print("Updated Width:", rect.width)
37      print("Updated Height:", rect.height)
38      print("Updated Area:", rect.area)
39
40  main() # calls the main() program
```



Creating more objects

- Modify the main program to create another Rectangle object named **rect2** with a width of 10 and a height of 8.
- Print the width, height and area of the rectangle.

Answer

```
rect2 = Rectangle(10, 8)
```

```
print("Width:", rect2.width)
```

```
print("Height:", rect2.height)
```

```
print("Area:", rect2.area)
```



Modifying the class Rectangle to add a method

- Modify the class Rectangle to add a method to compute the perimeter of a rectangle.
- Modify the main program to display the perimeter of both rectangles (rect and rect2).

```
1 class Rectangle:
2     def __init__(self, width, height):
3         self._width = width
4         self._height = height
5
6     @property
7     def width(self):
8         return self._width
9
10    @width.setter
11    def width(self, value):
12        self._width = value
13
14    @property
15    def height(self):
16        return self._height
17
18    @height.setter
19    def height(self, value):
20        self._height = value
21
22    @property
23    def area(self):
24        return self._width * self._height
25
26    @property
27    def perimeter(self):
28        return 2*(self._width + self._height)
```

```
30 def main():
31     rect = Rectangle(4, 5)
32
33     print("Width:", rect.width) # Access the width
34     print("Height:", rect.height) # Access the height
35     print("Area:", rect.area) # Access the area
36
37     rect.width = 6 # Update the width
38     rect.height = 7 # Update the height
39
40     print("Updated Width:", rect.width)
41     print("Updated Height:", rect.height)
42     print("Updated Area:", rect.area)
43
44     rect2 = Rectangle(10, 8)
45
46     print("Width:", rect2.width)
47     print("Height:", rect2.height)
48     print("Area:", rect2.area)
49
50     print("Perimeter:", rect.perimeter)
51     print("Perimeter:", rect2.perimeter)
52
53 main() # calls the main() program
```



Multiple Choice Question

1. What is the primary purpose of using the `@property` decorator in Python?

- A. To access attributes like variables instead of calling methods.
- B. To make a variable hidden from the user.
- C. To automatically store data in the class.
- D. To create a global variable.



Multiple Choice Question

2. What is the main purpose of the `__init__` method in Python classes?
- A. To define the class attributes globally.
 - B. To initialise an object's attributes when it is created.
 - C. To create a private method in the class.
 - D. To execute code only once during the program's runtime.



True/False Questions

1. In Python, you must explicitly call the `__init__` method to initialise an object.
2. Attributes of a class can only be accessed using the `self` keyword inside methods.
3. The `@property` decorator in Python allows a method to be accessed like an attribute.
4. It is mandatory for every Python class to define the `__init__` method.
5. You can create a method in a Python class without using any decorators.



Fill in the blank questions

1. The `__init__` method in a class is called the _____.
2. The keyword _____ is used to refer to the current instance of a class.
3. The _____ decorator is used to define a method that can be accessed like an attribute.
4. Python, a class is defined using the keyword _____.

Class Exercise

Study the class `BankAccount` provided below and then answer the questions which follow.
Note that not all methods are shown in the screenshot below.

```
1  class BankAccount:
2      def __init__(self, account_holder, balance=0):
3          self._account_holder = account_holder
4          self._balance = balance
5
6      @property
7      def balance(self):
8          return self._balance
9
10     def deposit(self, amount):
11         self._balance = self._balance + amount
12
13     def withdraw(self, amount):
14         if amount > self._balance:
15             raise ValueError("Insufficient funds.")
16         self._balance -= amount
```



Question

- Write a main program to create a bank account instance/object named **ba1**. The parameters are Alia and 10000.
- Write instructions to deposit Rs5000 in the bank account and display the new balance.
- Write instructions to withdraw Rs3000 in the bank account and display the new balance.

Answer

```
def main():
    # Create an account for Alia with 10000
    account = BankAccount("Alia", 10000)

    # Deposit 5000
    account.deposit(5000)

    # Check current balance
    print(account.balance)

    # Withdraw 3000
    account.withdraw(3000)

    #print the balance
    print(account.balance)

main()
```



Benefits of OOP

Object-Oriented Programming (OOP) offers several benefits that make it a widely-used programming paradigm, especially for large and complex software systems.

- **Real-World mapping:** OOP concepts like classes, objects, and inheritance align naturally with real-world entities and relationships. This makes designing and understanding software systems more intuitive, especially for domains like game development, simulations, and GUIs.
- **Modularity:** Code is organised into small, self-contained units (classes) that represent specific concepts or entities. This makes programs easier to read, debug, and maintain since related functionalities are grouped together.



Benefits of OOP

- **Code Reusability:** OOP promotes the reuse of existing code through classes and inheritance. This reduces duplication and accelerates development by allowing developers to extend existing classes instead of writing new ones from scratch.
- **Scalability / Extensibility:** The modular structure of OOP makes it easier to add new features or modify existing ones without disrupting other parts of the system. This facilitates growth and evolution of software over time.



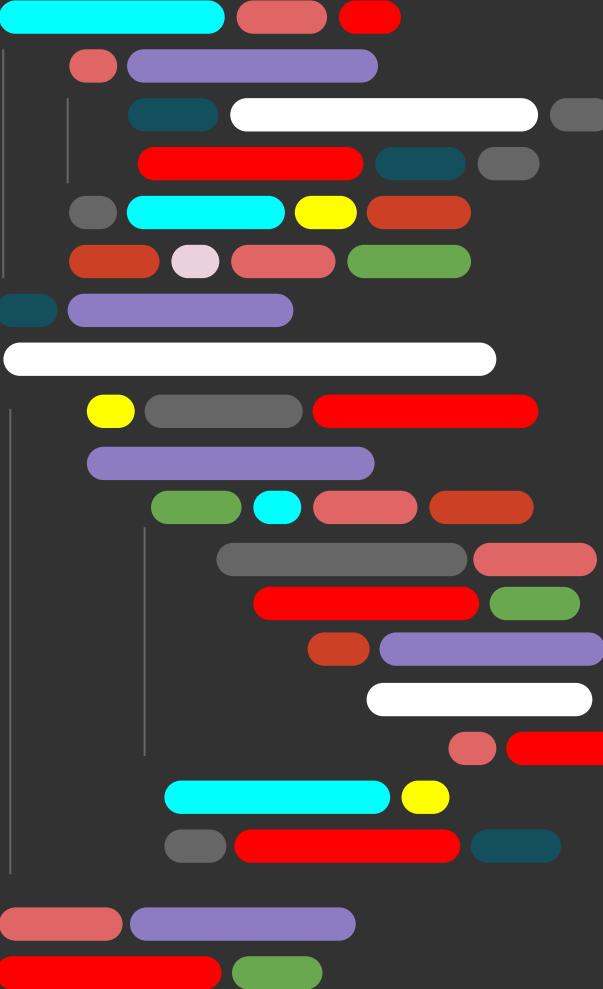
Benefits of OOP

- **Maintainability:** OOP structures code in a way that is easier to update and modify. This makes fixing bugs and implementing changes more efficient, reducing the long-term cost of maintaining the software.
- **Collaboration:** OOP allows different developers to work on different classes independently. This enhances teamwork and parallel development, as the modular nature minimises interference between teams.



What's Next

- Destructors
- Class variables and @classmethods
- Static methods
- Using a list to store a group of objects
- Process a list of objects



Week 14
ICDT1201Y – Computer
Programming

Object Oriented Programming



Learning Objectives

- Implement destructors
- Understand and use class variables
- Understand and use static methods
- Use a list to store a group of objects
- Process a list of objects



Destructors

- A **destructor** is a special method used to clean up memory resources or perform specific actions when an object is destroyed.
- Python has a built-in garbage collection system that automatically handles memory management, cleaning up objects when they are no longer in use.
- If your class manages resources that require explicit cleanup (like closing files, or releasing network connections), you may need a destructor.
- This can be implemented using the `__del__` method.

Example of a destructor

```
class Rectangle:  
    def __init__(self, width, height):  
        self._width = width  
        self._height = height  
  
    @property  
    def area(self):  
        return self._width * self._height  
  
    @property  
    def perimeter(self):  
        return 2*(self._width + self._height)  
  
    def __del__(self):  
        print("A rectangle object has just been deleted.")
```

*Not all methods have been shown in the class.

Sample main program

```
def main():
    rect = Rectangle(4, 5)

    print("Width:", rect.width)      # Access the width
    print("Height:", rect.height)    # Access the height
    print("Area:", rect.area)        # Access the area

    rect.width = 6                  # Update the width
    rect.height = 7                 # Update the height

    print("Updated Width:", rect.width)  # Displays the width
    print("Updated Height:", rect.height) # Displays the height
    print("Updated Area:", rect.area)    # Displays the area

    rect2 = Rectangle(10, 8)

    print("Width:", rect2.width)
    print("Height:", rect2.height)
    print("Area:", rect2.area)

    print("Perimeter:", rect.perimeter)
    print("Perimeter:", rect2.perimeter)
```

Write the output for the program

Width: 4

Height: 5

Area: 20

Updated Width: 6

Updated Height: 7

Updated Area: 42

Width: 10

Height: 8

Area: 80

Perimeter: 26

Perimeter: 36

A rectangle object has just been deleted.

A rectangle object has just been deleted.



Class variables

- A **class variable** is a variable that is shared among all instances of a class.
- It is defined at the class level, outside of any methods, and is accessible to all instances of the class.
- It can be accessed using the class name (e.g., `Rectangle.class_variable`) or an instance (e.g., `rect.class_variable`).
- It is often used to store information or properties that are common to all instances of a class (e.g., a counter to track the number of objects).

Class variables

```
class Rectangle:  
    # Class variable to count the number of Rectangle objects  
    object_count = 0  
  
    def __init__(self, width, height):  
        self._width = width  
        self._height = height  
        Rectangle.object_count += 1 # Increment object count  
  
    def __del__(self):  
        Rectangle.object_count -= 1 # Decrement object count  
        print("A rectangle object has just been deleted.")
```

Main program

```
def main():
    print(f"Current number of Rectangle objects: {Rectangle.object_count}")

    rect = Rectangle(4, 5)

    # Access the count directly
    print(f"Current number of Rectangle objects: {Rectangle.object_count}")
    print(f"Current number of Rectangle objects_rect: {rect.object_count}")

    rect2 = Rectangle(10, 8)

    # Access the count directly
    print(f"Current number of Rectangle objects: {Rectangle.object_count}")
    print(f"Current number of Rectangle objects_rect: {rect.object_count}")
    print(f"Current number of Rectangle objects_rect2: {rect2.object_count}")
```

Class Rectangle with all methods

```
class Rectangle:  
    # Class variable to count the number of Rectangle objects  
    object_count = 0  
  
    def __init__(self, width, height):  
        self._width = width  
        self._height = height  
        Rectangle.object_count += 1 # Increment object count  
  
    @property  
    def width(self):  
        return self._width  
  
    @width.setter  
    def width(self, value):  
        self._width = value  
  
    @property  
    def height(self):  
        return self._height  
  
    @height.setter  
    def height(self, value):  
        self._height = value  
  
    @property  
    def area(self):  
        return self._width * self._height  
  
    @property  
    def perimeter(self):  
        return 2 * (self._width + self._height)  
  
    def __del__(self):  
        Rectangle.object_count -= 1  
        print("A rectangle object has just been deleted.")
```

The Main Program

```
def main():

    print(f"Current number of Rectangle objects: {Rectangle.object_count}")

    rect = Rectangle(4, 5)

    print("Updated Width:", rect.width)
    print("Updated Height:", rect.height)
    print("Updated Area:", rect.area)
    print("Perimeter:", rect.perimeter)

    # Access the count directly
    print(f"Current number of Rectangle objects: {Rectangle.object_count}")
    print(f"Current number of Rectangle objects_rect: {rect.object_count}")

    rect2 = Rectangle(10, 8)

    print("Width:", rect2.width)
    print("Height:", rect2.height)
    print("Area:", rect2.area)
    print("Perimeter:", rect2.perimeter)

    # Access the count directly
    print(f"Current number of Rectangle objects: {Rectangle.object_count}")

    print(f"Current number of Rectangle objects_rect: {rect.object_count}")
    print(f"Current number of Rectangle objects_rect2: {rect2.object_count}")

    del rect
    del rect2
    print(f"Current number of Rectangle objects: {Rectangle.object_count}")

main()
```



Static Methods

- A static method is a method in a class that does not depend on instance-specific data or the class itself.
- Static methods are typically used for utility functions that are relevant to the class but do not need to access or modify its state.
- It is defined using the **@staticmethod** decorator.
- It can be called on the class directly:
Rectangle.method_name()
- It can also be called on an instance:
rect.method_name()

Example of a static method

```
class Rectangle:
    object_count = 0

    def __init__(self, width, height):
        if not self.is_valid_rectangle(width, height):
            print("Both the width and height must be greater than zero.")
        self._width = width
        self._height = height
        Rectangle.object_count += 1

    @width.setter
    def width(self, value):
        if not self.is_valid_rectangle(value, self._height):
            print("Width must be greater than zero.")
        self._width = value

    @height.setter
    def height(self, value):
        if not self.is_valid_rectangle(self._width, value):
            print("Height must be greater than zero.")
        self._height = value

    def __del__(self):
        Rectangle.object_count -= 1
        print("A rectangle object has just been deleted.")

    @staticmethod
    def is_valid_rectangle(w, h):
        return w >= 0 and h >= 0
```

Main program

```
def main():

    #valid rectangle
    rect = Rectangle(4, 5)
    print(f"Width: {rect.width}, Height: {rect.height}")

    #invalid rectangle
    rect2 = Rectangle(3, -5)

    #Check if dimensions are valid
    print(Rectangle.is_valid_rectangle(6, 7))
    print(Rectangle.is_valid_rectangle(-1, 2))
    print(Rectangle.is_valid_rectangle(6, -7))
    print(Rectangle.is_valid_rectangle(-1, -2))

main()
```



Class Exercise

Write a main program which checks the validity of the dimensions (width and height) using the `is_valid_rectangle()` method before creating a Rectangle object.

Assume that the values are already saved in the variables `w` and `h`.

An appropriate message must be displayed in both cases (successful creation or unsuccessful creation).

Answer

```
if Rectangle.is_valid_rectangle(w, h):  
    rect3 = Rectangle(w, h)  
    print("A rectangle has been created.")  
  
else:  
    print("The dimensions are invalid.")
```



Handling a set of objects

- Write a main program which ask the user for the number of rectangles that needs to be created. The dimensions of these rectangles are then entered inside a loop. The rectangles are created and saved to a list.
- Add another loop to display the area and perimeter of all the rectangles in the list.

Answer

```
def main():
    abcd = []
    num = int(input("How many rectangles would you like to create? "))

    for i in range(num):
        print("\nEnter dimensions for the rectangle:")
        width = float(input("Width: "))
        height = float(input("Height: "))
        rect = Rectangle(width, height)
        abcd.append(rect)

    for r in abcd:
        print(f"\nArea = {r.area}, Perimeter = {r.perimeter}")

main()
```



Multiple Choice Question

Which of the following is true about static methods in Python?

- A. Static methods are independent of instance and class attributes.
- B. Static methods can only be defined outside the class.
- C. Static methods are required to access instance attributes directly.
- D. Static methods are automatically executed when an object is created.



Multiple Choice Question

Which of the following is true about class variables in Python?

- A. Class variables are shared among all instances of the class.
- B. Class variables are unique to each instance of the class.
- C. Class variables can only be accessed through instance methods.
- D. Class variables are defined inside methods, not directly in the class.



Multiple Choice Question

What is the purpose of a **destructor** in Python?

- A. To initialise the attributes of a class.
- B. To delete the class definition from memory.
- C. To clean up resources when an object is no longer needed.
- D. To create multiple objects of a class.



Fill in the blank questions

1. A _____ method in Python does not require access to instance or class-specific data.
2. A _____ is shared among all instances of a class and is defined at the class level.
3. The _____ method is called when an object is being destroyed.
4. To store and manage a collection of objects, we commonly use a _____ in Python



Fill in the blank questions

5. A constructor in Python is defined using the _____ method.

6. A _____ in Python is a special function that is prefixed with the @ symbol.

7. The _____ method of a list adds an element (e.g. an object) to the end of the list.

8. A _____ in Python is a _____ for creating _____, which are _____ of that _____.

True/False Questions

1. A static method in Python can directly access instance variables.
2. Class variables are shared among all instances of a class.
3. The `__del__` method is called automatically when an object is created.
4. A list of objects in Python allows you to store and manipulate multiple instances of a class.



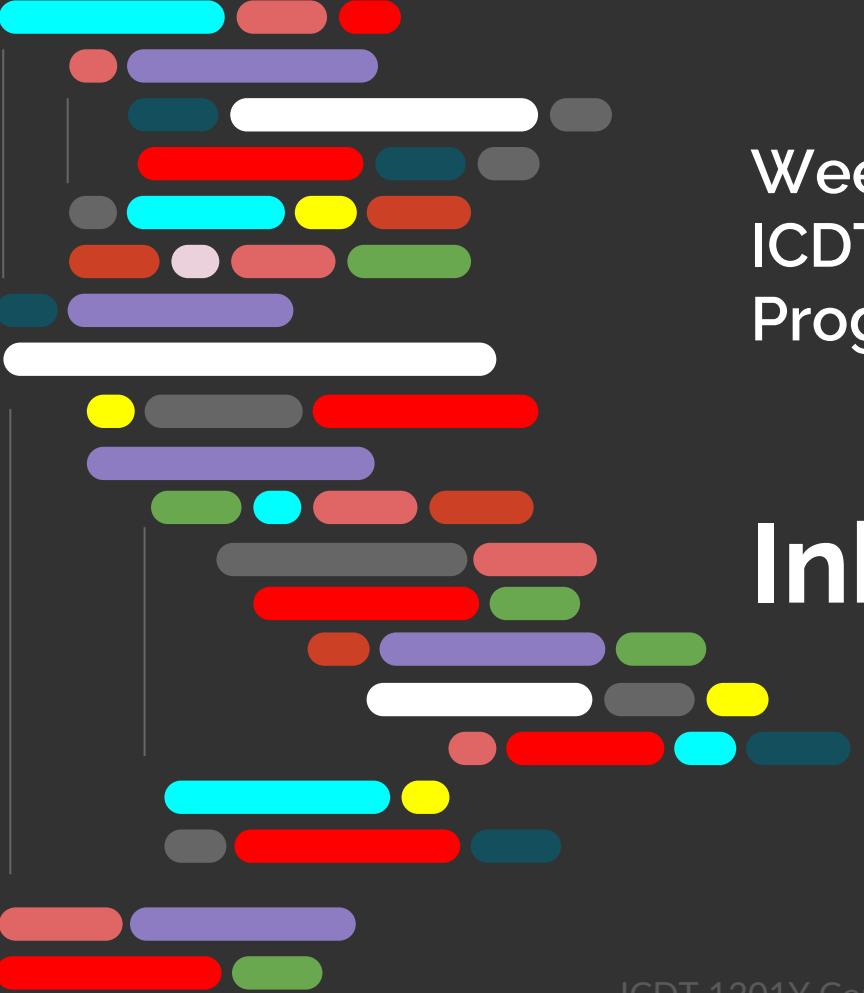
True/False Questions

5. A constructor in Python is defined using the `__create__` method.
6. A class variable in Python can be accessed using both the class name and an instance.
7. Constructors are used to define the initial state of an object when it is created.
8. The `append` method of a list replaces the last element with a new value.



What's Next

- Inheritance
- Parent and child classes
- Inheritance in Python
- Method overriding
- Method overloading



Week 15

ICDT1201Y – Computer Programming

Inheritance



Inheritance

- Inheritance is a fundamental concept in object-oriented programming that allows one class (called the child class, subclass or derived class) to acquire the properties and behaviours (attributes and methods) of another class (called the parent class, superclass or base class).
- This promotes code reuse and establishes a hierarchical relationship between classes.



Types of Inheritance

- **Single inheritance:** A child class inherits from one parent class.
- **Multilevel inheritance:** A class inherits from a child class, creating a chain of inheritance.
- **Multiple inheritance:** A child class inherits from more than one parent class.
- **Hierarchical inheritance:** Multiple child classes inherit from a single parent class.
- **Hybrid inheritance:** A combination of two or more types of inheritance.

Single inheritance

```
# Superclass
class Shape:
    def __init__(self, name):
        self._name = name

    def display_name(self):
        return f"This is a {self._name}."

# Child class
class Rectangle(Shape):
    def __init__(self, width, height):
        super().__init__("Rectangle")
        self._width = width
        self._height = height

    def area(self):
        return self._width * self._height

# Example usage
rect = Rectangle(5, 10)
print(rect.display_name())
print("Area:", rect.area())
```



Class Exercise

- Implement another child class Circle which inherits from the parent class Shape.
- The class Circle has only one additional attribute which is radius.
- Also, implement a method area() to compute the area of a circle.

Answer – Class

```
# Child class
class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self._radius = radius

    def area(self):
        return 3.14159*self._radius**2
```

Answer – Main program

```
rect = Rectangle(5, 10)  
print(rect.display_name())  
print("Area:", rect.area())
```

```
circle = Circle(7)  
print(circle.display_name())  
print("Circle Area:", circle.area())
```

Multilevel Inheritance

```
# Child class of Rectangle
class Square(Rectangle):
    def __init__(self, width):
        super().__init__(width, width)
        self._name = "Square"
```

Multiple Inheritance – Parent Classes

```
# Superclass 1: Shape
class Shape:
    def __init__(self, name):
        self._name = name
    def display_name(self):
        return f"This is a {self._name}."

# Superclass 2: Colour
class Colour:
    def __init__(self, colour):
        self._colour = colour
    def display_colour(self):
        return f"Its colour is {self._colour}."
```

Multiple Inheritance - Subclass

```
# Subclass: ColouredRectangle
class ColouredRectangle(Shape, Colour):
    def __init__(self, width, height, colour):
        Shape.__init__(self, "Rectangle")
        Colour.__init__(self, colour)
        self._width = width
        self._height = height
    def area(self):
        return self._width * self._height
```

Multiple Inheritance – Main program

```
def main():

    rect = ColouredRectangle(5, 10, "red")

    print(rect.display_name())
    print(rect.display_colour())
    print("Its area is: ", rect.area())

main()
```

The complete program – Multiple Inheritance

```
1 # Superclass 1: Shape
2 class Shape:
3     def __init__(self, name):
4         self._name = name
5
6     def display_name(self):
7         return f"This is a {self._name}."
8
9 # Superclass 2: Colour
10 class Colour:
11     def __init__(self, colour):
12         self._colour = colour
13
14     def display_colour(self):
15         return f"Its colour is {self._colour}."
16
17 # Subclass: ColouredRectangle
18 class ColouredRectangle(Shape, Colour):
19     def __init__(self, width, height, colour):
20         Shape.__init__(self, "Rectangle")
21         Colour.__init__(self, colour)
22         self._width = width
23         self._height = height
24
25     def area(self):
26         return self._width * self._height
27
28 rect = ColouredRectangle(5, 10, "red")
29
30 print(rect.display_name())
31 print(rect.display_colour())
32 print("Its area is:", rect.area())
```



Class Exercise on Inheritance

- A private hospital wishes to have a computerised system to manage staff information. The attributes for staff are name, address, gender and salary.
- **Staff** is a parent class from which **Doctor** and **Nurse** inherit.
- Doctors have three additional attributes which are specialisation, daily fee and number of days worked for each month.
- Doctors have a fixed basic salary and a daily fee paid for each day of work in a month.
- Nurses also have three additional attributes which are department, number of overtime hours worked per month and the overtime rate.
- Implement the class Staff, Doctor and Nurse.
- Implement a method to calculate the monthly income of a doctor.
- Implement a method to calculate the monthly income of a nurse.
- Implement a main program to test the above methods.

Staff

```
class Staff:

    def __init__(self, nam, add, gen, sal):
        self._name = nam
        self._address = add
        self._gender = gen
        self._salary = sal

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        self._name = value

    @property
    def address(self):
        return self._address

    @address.setter
    def address(self, value):
        self._address = value

    @property
    def gender(self):
        return self._gender

    @gender.setter
    def gender(self, value):
        self._gender = value
```



Staff

```
class Staff:  
  
    @property  
    def salary(self):  
        return self._salary  
  
    @salary.setter  
    def salary(self, value):  
        self._salary = value  
  
    def monthly_income(self):  
        return self.salary  
  
    def display_data(self):  
        print(f"Name: {self.name}")  
        print(f"Address: {self.address}")  
        print(f"Gender: {self.gender}")  
        print(f"Salary: {self.salary}")
```

Doctor

```
class Doctor(Staff):
    def __init__(self, n, a, g, s, special, fee, number_of_days):
        super().__init__(n, a, g, s)
        self._specialisation = special
        self._daily_fee = fee
        self._num_days = number_of_days

    @property
    def specialisation(self):
        return self._specialization

    @specialisation.setter
    def specialisation(self, value):
        self._specialisation = value

    @property
    def daily_fee(self):
        return self._daily_fee

    @daily_fee.setter
    def daily_fee(self, value):
        self._daily_fee = value
```

Doctor

```
class Doctor(Staff):  
    @property  
    def num_days(self):  
        return self._num_days  
  
    @num_days.setter  
    def num_days(self, num_days):  
        self._num_days = num_days  
  
    def monthly_income(self):  
        return self.salary + self.daily_fee * self._num_days  
  
    def display_data(self):  
        super().display_data()  
        print(f"Specialisation: {self.specialisation}")  
        print(f"Daily Fee: {self.daily_fee}")  
        print(f"Number of days works : {self.num_days}")  
        print(f"Monthly income : {self.monthly_income()}")
```



Nurse

```
class Nurse(Staff):  
  
    def __init__(self, na, ad, ge, sa, dept, num_hours, rate):  
        super().__init__(na, ad, ge, sa)  
        self._department = dept  
        self._overtime_hours = num_hours  
        self._overtime_rate = rate  
  
    @property  
    def department(self):  
        return self._department  
  
    @department.setter  
    def department(self, dept):  
        self._department = dept  
  
    @property  
    def overtime_hours(self):  
        return self._overtime_hours  
  
    @overtime_hours.setter  
    def overtime_hours(self, hours):  
        self._overtime_hours = hours
```



```
class Nurse(Staff):
    @property
        def overtime_rate(self):
            return self._overtime_rate

    @overtime_rate.setter
        def overtime_rate(self, overtimerate):
            self._overtime_rate = overtimerate

    def monthly_income(self):
        return self.salary + self.overtime_hours * self.overtime_rate

    def display_data(self):
        super().display_details()
        print(f"Department: {self.department}")
        print(f"Overtime Hours: {self.overtime_hours}")
        print(f"Overtime Rate: {self.overtime_rate}")
        print(f"Monthly Income: {self.monthly_income()}")
```

Nurse

Main program

```
if __name__ == "__main__":
    staff1 = Staff("Sam", "Port Louis", "Male", 30000)
    staff1.display_data()
    print(f"Monthly Income: {staff1.monthly_income()}")
    doctor1 = Doctor("Dr Sam", "Maurice", "Male", 50000, "Cardiology", 2000, 20)
    doctor1.display_data()
    nurse1 = Nurse("Alia", "Reduit", "Female", 25000, "Pediatrics", 15, 200)
    nurse1.display_data()
```



Multiple Choice Question

What is the purpose of inheritance in Python?

- A. To create private attributes in a class
- B. To create reusable classes that derive properties and behaviors from a base class
- C. To prevent access to certain methods in a class
- D. To create functions within a class

Multiple Choice Question

Given the following Python code, what will be the output?

```
class Parent:
```

```
    def greet(self):  
        return "Hello from Parent"
```

```
class Child(Parent):
```

```
    def greet(self):  
        return super().greet() + " and Child"
```

```
parent_obj = Parent()
```

```
child_obj = Child()
```

```
print(parent_obj.greet())
```



Multiple Choice Question

- A. Hello from Parent
- B. Hello from Child
- C. Hello from Parent and Child
- D. Error: Parent class method cannot be called



Multiple Choice Question

What is method overriding in Python?

- A. Defining multiple methods with the same name but different arguments in the same class.
- B. Redefining a method in the child class that already exists in the parent class.
- C. Using the super() keyword to call a method from the parent class.
- D. Defining methods without implementation in a class.



Fill in the blank questions

1. In OOP, the base class is also known as the _____ class.
2. The keyword _____ is used to access the methods of a parent class in a child class.
3. Inheritance allows a class to acquire the _____ and _____ of another class.
4. In Python, if a method is not overridden in the child class, the _____ class method is executed.



Fill in the blank questions

5. _____ refers to defining multiple methods with the same name but with different argument signatures in a class.
6. Python supports multiple _____, allowing a class to inherit from more than one parent class.
7. The constructor of the _____ class is executed before the constructor of the _____ class.
8. When an object is destroyed, the destructor of the _____ class is executed first.



True/False Questions

1. A Python class can inherit from multiple parent classes.
2. The super() function is used to access methods and attributes of the child class from the parent class.
3. Method overriding occurs when a method in a parent class is replaced by a method with the same name and signature in the child class.
4. If a child class does not define a constructor, the parent class constructor is automatically invoked.



True/False Questions

5. A class in Python can have only one constructor.
6. An instance of a class is created using the class keyword.
7. Class variables are shared among all instances of a class.
8. Polymorphism allows methods with the same name to behave differently in different classes.



Week 15
ICDT1201Y – Computer
Programming

Naming Conventions & Coding Ethics

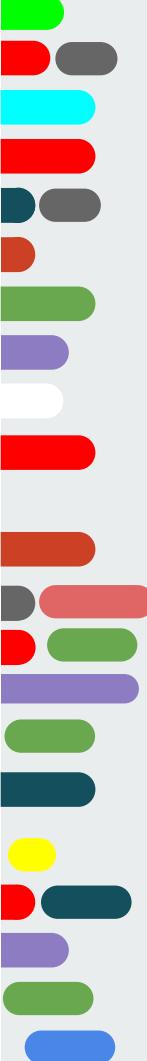
ICDT 1201Y Computer Programming



Learning Objectives

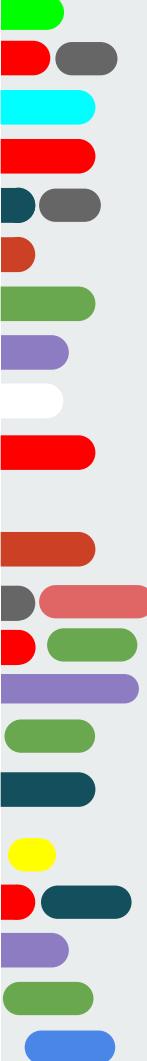
By the end of this lecture, students should be able to:

- Understand the importance of program documentation
- Understand the importance of naming conventions in coding
- Apply naming conventions in Python coding
- Understand the importance of coding ethics
- Know how to apply coding ethics



Agenda

- Software Documentation
- Naming Conventions
- Coding Ethics
- <https://policies.python.org/python.org/code-of-conduct/>
- Plagiarism
- LLM & coding



Software Documentation

- Information, available in writing, about a program/software
- How the software works
- Different types to suit different stakeholders – programmers, end user, technical operators



Software Documentation

- Code documentation - inline comments and external documents that describe the purpose & function of the code.
- Requirement Documentation – how software shall perform, environment setup
- Architectural Documentation – concerns design & system components
- Technical Documentation – technical aspects like APIs, algorithms
- End-user Documentation – meant for end users



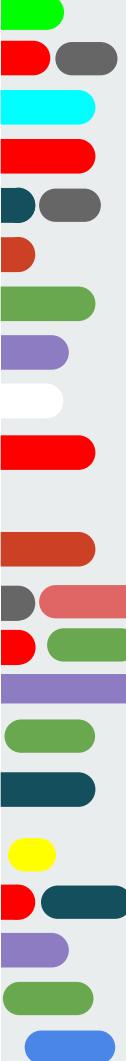
Naming Conventions

- Python known for simplicity & readability
- Strong emphasis on writing clean and maintainable code.
- Python naming conventions:
 - Modules
 - Variables
 - Classes
 - Exceptions



Naming Conventions

- Modules/functions - lowercase letters and underscores (e.g: `add_numbers`)
- Variables:
 - Global variables - uppercase with underscores separating words (e.g `PI_VALUE`)
 - Local variables – same as modules (e.g `counter_val`)
- Classes - CapWords (or CamelCase) (e.g `BankAccount`)
- Exceptions – end with Error (e.g `CustomError`)



Importance of Naming Conventions

- Enhance code readability, making it easier for developers to understand the purpose and functionality of variables, functions, classes, etc
- Consistent naming conventions contribute to code maintainability. When developers follow a standardized naming pattern, it becomes more straightforward for others to update, debug, or extend the code.
- Naming conventions are especially important in collaborative coding environments. When multiple developers work on a project, adhering to a common naming style ensures a cohesive and unified codebase.
- Well-chosen names can help prevent errors. A descriptive name that accurately reflects the purpose of a variable or function reduces the likelihood of misunderstandings or unintentional misuse.



Class Exercise

- List some challenges associated with programs that do not follow a naming conventions?



Coding Ethics

- Programs have very specific characteristics that can make it very harmful if not created or used properly.
 - Programs are everywhere and allows communication of all types of information.
 - Software controls hardware.
 - Software handles financial information (virtual money).



Coding Ethics

- Developers are expected to follow ethical guidelines and apply them when writing programming code.
- Developers should use good practices and have good attitudes while doing programming. This ensures correct programs that are meant to solve problem and not create problems.



Programming Ethical Guidelines

Code of Ethics and Professional Conduct (ACM):

- Contribute to society and human well-being.
- Avoid harm to others.
- Be honest and trustworthy.
- Give proper credit for intellectual property.
- Respect the privacy of others.
- Honor confidentiality.



Python Software Foundation Code of Conduct

- Being open
- Focusing on what is best for the community.
- Acknowledging time and effort.
- Being respectful of differing viewpoints and experiences.
- Showing empathy towards other community members.
- Being considerate.
- Being respectful.
- Gracefully accepting constructive criticism.
- Using welcoming and inclusive language.



Class Exercise

- Name some Inappropriate Behaviours in a coding community
- Further reading: Check the following link -
Inappropriate behaviours (Python Community):
<https://policies.python.org/python.org/code-of-conduct/>



Plagiarism

- Plagiarism is the act of presenting someone else's work as yours, it is also known as intellectual theft.
- Two types:
 - Intentional Plagiarism
 - Accidental Plagiarism

What's the difference between the two?



Plagiarism in coding

- Just like any other type of Plagiarism
- Use of somebody else's code and claiming it's your own



Plagiarism in coding

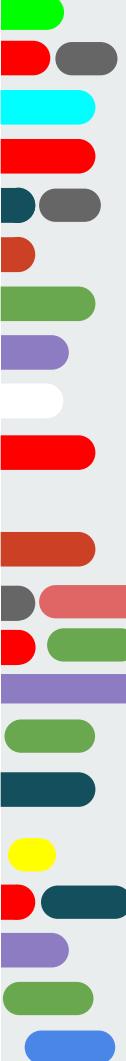
- **Direct Copy-Pasting** – Taking code from online sources, books, or another developer without giving credit.
- **Slight Modifications** – Changing variable names or formatting but keeping the logic and structure the same.
- **Unauthorized Code Reuse** – Using code from open-source projects or other developers without following licensing terms.
- **Automated Code Generation Without Credit** – Using AI-generated or publicly available code without proper acknowledgment.
- **Collusion in Assignments** – Submitting another person's work as your own in academic or professional settings.

Source: <https://chatgpt.com/c/679f94c0-78e0-8001-8b79-c8ddc4717621>



Class Exercise

- Analyse the previous slide
- Would you define this slide as having plagiarised content?



Avoiding plagiarism in coding

- Best Way – Write your own codes
- When using external resources, always cite them properly and clearly.
- Give Credit – Use comments or documentation to acknowledge sources.
- Follow Open-Source Licenses – Respect licensing rules when using external code.
- Understand and Rewrite Code – Learn the logic and write it in your own way.
- Use Plagiarism Detection Tools – Tools like MOSS, Codequiry, or JPlag can help detect similarities.

Source: <https://chatgpt.com/c/679f94c0-78e0-8001-8b79-c8ddc4717621>



LLM and coding

- There are LLMs intended for code generation, and programming-specific LLMs can help developers with a multitude of programming tasks
 - OpenAI Codex
 - GPT-4
 - GitHub Copilot
 - Llama 3
 - Claude 3 Opus



Why Are LLMs Important for developers?

- Increased Efficiency
- Improved Accuracy
- Enhanced Learning and Documentation
- Better Collaboration
- Learning Opportunities
- Consistent Coding Standards



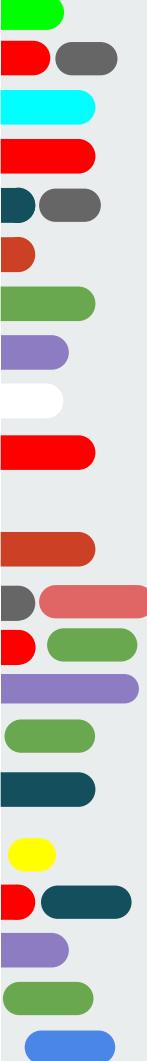
Cons of Using LLMs and AI Tools

- Quality of Output
- Accuracy??
- Dependency on AI → reduces problem solving skills
- Security Concerns
- Intellectual Property Issue
- Integration Challenges



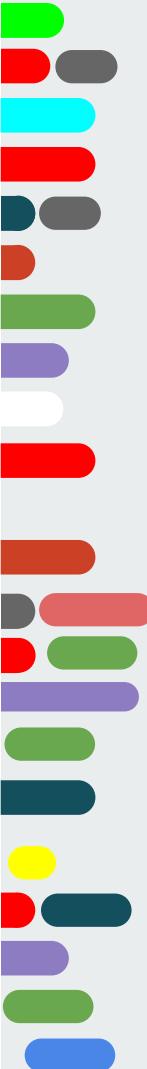
Reflection

- As a student, should you AI tools during coding?



Self Learning

- Use an AI tool to generate the Python codes for Labsheet 11
- Analyse the code and compare with the codes that you wrote
- What can you say in terms of the accuracy, validity of the code generated?



Summary

- Need for program documentation
- Need & application of naming conventions
- Need for coding ethics
- Writing codes using ethical
- Next lecture → Introduction to Data Structures



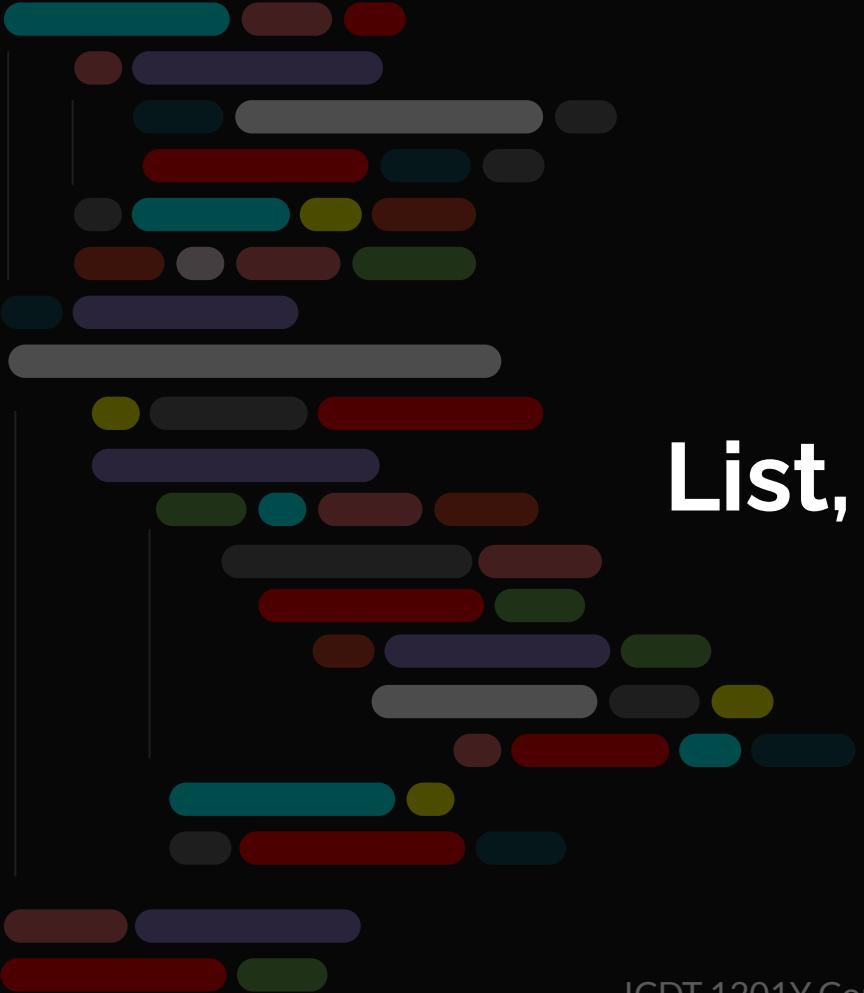
References

- <https://www.geeksforgeeks.org/python-naming-conventions/>
- <https://policies.python.org/python.org/code-of-conduct/>
- <https://crossplag.com/what-is-plagiarism-in-programming/>
- Lecture notes of SIS1040Y from FoICDT
- <https://chatgpt.com>
- Source:<https://autogpt.net/top-langs-for-coding-all-developers-should-know-about/#t-1738937707903>
- Source:<https://www.linkedin.com/pulse/pros-cons-using-langs-ai-tools-code-generation-refactoring-shatru-naik-emvcc/>
- <https://www.geeksforgeeks.org/overview-software-documentation/>



Data Structures

Week 16 -20



List, tuples and sets



Learning Outcome and Assessment Criteria

Introduce the concepts of advanced data structures

- Understand the difference between commonly-used data structures (List, Tuples, Sets, Dictionaries, Stack and Queues)
- Write medium-sized programs that use different data structures to manipulate data

Demonstrate the principles of Sorting and Searching algorithms

- Introduce the concepts of searching and sorting
- Use in-built searching algorithms
- Use in-built sorting algorithms
- Write medium-sized programs that use basic searching and sorting algorithms

Data Structures Part 1

- Data Structures (List, Tuples, Sets, Dictionaries, Stack and Queues)
- Lists ('Advanced' Operations of List (e.g Searching, Sorting, Insertions and Removal)

Data Structures Part 2

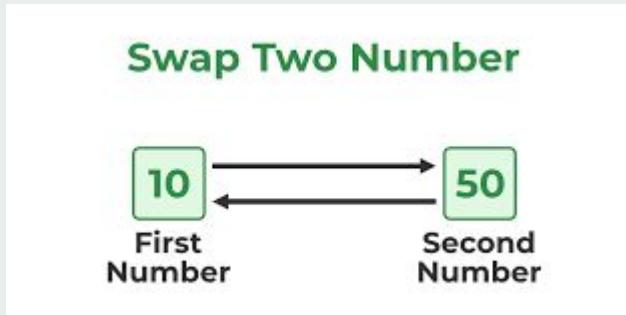
- Stacks and Queues

Data Structures Part 3

- Dictionaries
- Problem Solving using ADT

Swapping Values - a little warm-up!

Assuming $a=10$ and $b=50$, how would you swap the values of x and y ?



Swapping

There are 2 approaches:

1. Use a third variable:

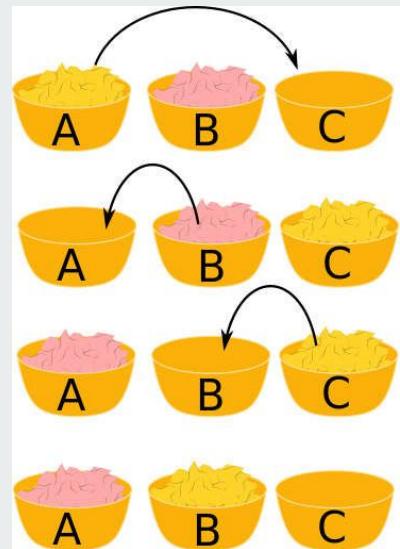
c=a

a=b

b=c

2. Python Simultaneous Assignment

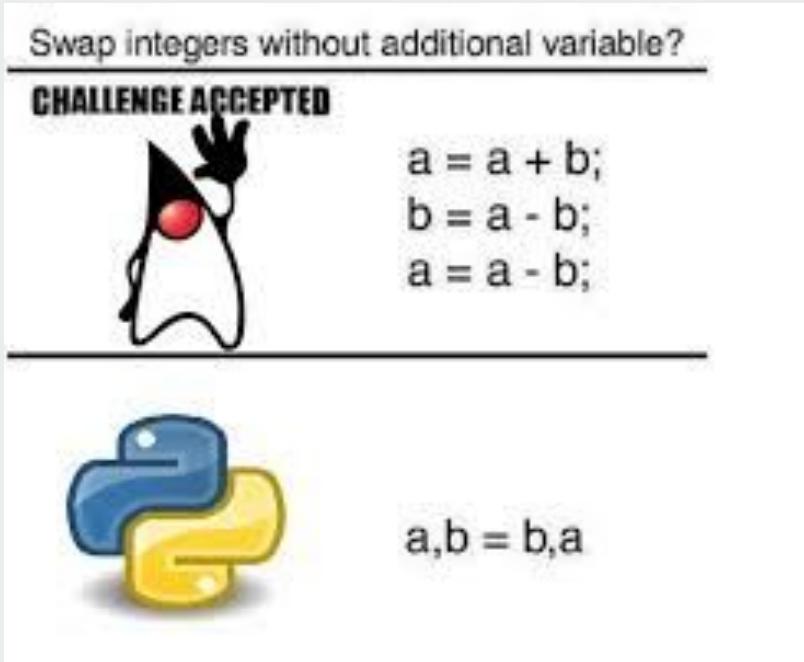
a , b = b , a



Source:

<https://www.i-programmer.info/babbages-bag/498-the-magic-swap.html>

OR!!!



Source:

<https://www.quora.com/How-can-you-explain-logic-Swapping-of-two-numbers-with-out-using-third-variable-to-a-layman-type-programmer>



Data Structure

A data structure in the context of programming is a way of organising, storing, retrieving and processing data. A data structure to which you have been exposed to is the list.

There are operations which can be carried out on data structures.

Most of the time these operations are already written and placed in modules/libraries which can be used by programmers



Types of Data Structures

It is essential that programmers know the specificities of different data structures.

Knowing the differences between the different data structures allows you to select the best one to solve a problem.

Being able to select the most appropriate Data Structure increase the robustness of your code and decreases the programming time and its complexity.



Types of Data Structures

- List
- Tuples
- Sets
- Dictionary
- Stacks
- Queues
- Trees
- Graphs



Data Structures

1. Lists

Definition:

A collection of ordered elements that can hold items of different types, including duplicates.

Characteristics:

Dynamic size (can grow and shrink).

Supports various operations like insertion, deletion, and indexing.

Common Use Cases: Storing a sequence of items, such as user inputs or records.

2. Tuples

Definition:

An immutable collection of ordered elements.

Characteristics:

Once created, the elements cannot be changed.

Can hold different types of data.

Common Use Cases: Returning multiple values from a function, fixed records.



Data Structures

3. Sets

Definition:

An unordered collection of unique elements.

Characteristics:

No duplicate entries allowed.

Common operations include union, intersection, and difference.

Common Use Cases: Removing duplicates from data, membership testing, and mathematical set operations.

4. Dictionaries (Hash Maps)

Definition: A collection of key-value pairs where each key is unique.

Characteristics:

Fast look-up, insertion, and deletion based on keys.

Keys must be immutable types.

Common Use Cases: Storing associative arrays, configurations, and mappings between objects.



Data Structures

5. Stacks

Definition:

A collection of elements that follows the Last In First Out (LIFO) principle.

Characteristics:

Elements can be added or removed only from the top.

Typical operations include push (add), pop (remove), and peek (view the top element).

Common Use Cases: Function call management, undo operations in applications, and expression evaluation.

6. Queues

Definition:

A collection of elements that follows the First In First Out (FIFO) principle.

Characteristics:

Elements are added at the back and removed from the front.

Common operations include enqueue (add) and dequeue (remove).

Common Use Cases: Task scheduling, handling requests in web servers, and breadth-first traversal in graphs.



Data Structures

7. Trees

Definition:

A hierarchical data structure with nodes connected by edges, having a root node and child nodes.

Characteristics:

Each node can have multiple children, but only one parent (except the root).

Variants include binary trees, binary search trees, AVL trees, and heaps.

Common Use Cases: Hierarchical data representation, search algorithms, and organizing sorted data.

8. Graphs

Definition:

A collection of nodes (or vertices) connected by edges.

Characteristics:

Can be directed or undirected.

Can include cycles.

Represented using adjacency lists or adjacency matrices.

Common Use Cases: Network routing, social networks, and modeling relationships between entities

List

List Methods - called with <listName>.method

| <u>Method</u> | <u>Description</u> |
|---------------|---|
| append() | Adds an element at the end of the list |
| clear() | Removes all the elements from the list |
| copy() | Returns a copy of the list |
| count() | Returns the number of elements with the specified value |
| extend() | Add the elements of a list, to the end of the current list |
| index() | Returns the index of the first element with the specified value |
| insert() | Adds an element at the specified position |
| pop() | Removes the element at the position passed as argument to it, if no argument is specified, the last value is removed |
| remove() | Removes the first item with the specified value |
| reverse() | Reverses the order of the list |
| sort() | Sorts the list |



Activity: The Island-Capital Problem

Write a program which allows the user to enter the name of an island nation of the Indian Ocean and prints the capital of that island nation.

| Island Nation | Capital City |
|---------------|---------------|
| Maldives | Malé |
| Sri Lanka | Colombo |
| Mauritius | Port Louis |
| Seychelles | Victoria |
| Madagascar | Antananarivo |
| Reunion | Saint-Denis |
| Comoros | Moroni |
| Djibouti | Djibouti City |
| Rodrigues | Port-Mathurin |



Think!!!



Use Two Lists - one for Countries and another for Capitals

listIslands=[“Maldives”, “Sri-Lanka”, “Mauritius”, “Seychelles”, “Madagascar”, “Reunion”, “Comoros”, “Djiibouti”, “Rodrigues”]

| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
|----------|-----------|-----------|------------|------------|---------|---------|-----------|-----------|
| Maldives | Sri-Lanka | Mauritius | Seychelles | Madagascar | Reunion | Comoros | Djiibouti | Rodrigues |

listCapitals=[“Malé”, “Colombo”, “Port Louis”, “Victoria”, “Antananarivo”, “Saint-Denis”, “Moroni”, “Djibouti City”, “Port-Mathurin”]

| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
|------|---------|------------|----------|--------------|-------------|--------|---------------|---------------|
| Malé | Colombo | Port Louis | Victoria | Antananarivo | Saint-Denis | Moroni | Djibouti City | Port-Mathurin |

Logic:

Step 1: Allow user to enter name of Country, C

Step 2: Look for index, i , of Country ,C, in the list called listIslands, if C is not found, display an error & exit.

Step 3: Display element at position i from the list called listCapitals

Step 4: Exit

2 Solutions!!!

```
listIslands=["Maldives", "Sri-Lanka", "Mauritius", "Seychelles", "Madagascar", "Reunion", "Comoros", "Djibouti", "Rodrigues"]
listCapitals=["Malé", "Colombo", "Port Louis", "Victoria", "Antananarivo", "Saint-Denis", "Moroni", "Djibouti City",
"Port-Mathurin"]
```

```
C=input("Enter Name of Country")
i=listIslands.index(C) #Note: Index returns none if element is not found in list
if i!=None:
    print(listCapitals[i])
else:
    print(C+" is not in the Indian Ocean")
```

```
listIslands=["Maldives", "Sri-Lanka", "Mauritius", "Seychelles", "Madagascar", "Reunion", "Comoros", "Djibouti", "Rodrigues"]
listCapitals=["Malé", "Colombo", "Port Louis", "Victoria", "Antananarivo", "Saint-Denis", "Moroni", "Djibouti City",
"Port-Mathurin"]
```

```
C=input("Enter Name of Island")
try:
    i=listIslands.index(C)
    print(listCapitals[i])
except:
    print(C+" is not in the Indian Ocean")
```

Tuples

Tuples

Tuples are similar to lists, except that the tuple is immutable.

A data structure is said to be immutable when its contents cannot be changed after it has been initialised.

Tuple Methods

| Method | Description |
|---------|---|
| count() | Returns the number of times a specified value occurs in a tuple |
| index() | Searches the tuple for a specified value and returns the position of where it was found |

Tuple Example

#Creating a Tuple

It is possible to create a tuple by giving it a name and listing its items as follows:

```
myTuple=("Red", "Blue", "Yellow", "Green")
```

Note:

The parentheses '(' and ')' when declaring the tuple.

It is also possible to create a tuple from a list.

```
<tupleName>=tuple(<list>)
```

E.g

```
Colours=["Red", "Blue", "Yellow", "Green"]  
myTuple=tuple(Colours)
```

#Printing a Tuple

Assuming myTuple=("Red", "Blue", "Yellow", "Green")

```
for i in myTuple:  
    print(i)
```

Or

```
for i in range(len(myTuple)):  
    print(myTuple[i])
```

Note:

Tuple elements are indexable just like elements in a list



Converting a Tuple to a List

It is possible to convert a tuple to a list.

```
myTuple=("Red", "Blue", "Yellow", "Green")
```

```
myList=list(myTuple)
```

```
for i in range(len(myList)):  
    print(myList[i])
```

Converting a List to a Tuple

It is possible to convert a list to a tuple.

```
myList=["Red", "Blue", "Yellow", "Green"]
```

```
myTuple=tuple(myList)
```

```
for i in range(len(myList)):  
    print(myTuple[i])
```



Adding/removing elements from a tuple

If it is a matter of life and death whereby you need to add/remove an element from a tuple, you can adopt the following logic.

- 1) Convert the tuple to a list,
- 2) Edit the list
- 3) Convert the list back to a tuple.

Sets



Sets

A set is an unchangeable and unordered collection of items.

Unchangeable means that the individual elements cannot be altered, but it is possible to add or remove elements in/from the set

A set cannot contain duplicates

Set Methods

| Method | Shortcut | Description |
|-------------------------------|----------|--|
| add() | | Adds an element to the set |
| clear() | | Removes all the elements from the set |
| copy() | | Returns a copy of the set |
| difference() | | Returns a set containing the difference between two or more sets |
| difference_update() | -= | Removes the items in this set that are also included in another, specified set |
| discard() | | Remove the specified item |
| intersection() | & | Returns a set, that is the intersection of two other sets |
| intersection_update() | &= | Removes the items in this set that are not present in other, specified set(s) |
| isdisjoint() | | Returns whether two sets have a intersection or not |
| issubset() | <= | Returns whether another set contains this set or not |
| | < | Returns whether all items in this set is present in other, specified set(s) |
| issuperset() | >= | Returns whether this set contains another set or not |
| | > | Returns whether all items in other, specified set(s) is present in this set |
| pop() | | Removes and returns a random element from the set |
| remove() | | Removes the specified element |
| symmetric_difference() | ^ | Returns a set with the symmetric differences of two sets |
| symmetric_difference_update() | ^= | Inserts the symmetric differences from this set and another |
| union() | | Return a set containing the union of sets |
| update() | = | Update the set with the union of this set and others |

Note: How to check **set equality** in Python (check whether two sets are equal or not)? You can use the `==` operator to test if two sets are equal in Python. The sets are considered equal if they have the same elements, regardless of the order in which the elements appear. Python Set equality comparison considers if the two sets share the exact same elements, regardless of order.

Dealing with Sets

#Creating a set

#Creates an empty Set

mySet=set({})

#Creates a set with elements e_1, e_2, \dots, e_n

mySet={ e_1, e_2, \dots, e_n }

#Adding an element ,e, to a set

mySet.add(e)

#Printing a set

print(mySet)

for item in mySet:

 print(item)

#Removing an element from a set

#Using Remove

<setName>.remove(<item>)

Note: if the item is not present in the set, the remove method raises an error (exception) which should be handled

#Using Discard

<setName>.discard(<item>)

Note: if the item is not present in the set, the discard method does not raise any error

#Union

<set3>=<set1>.union(<set2>)

Note: Union returns a set

#Intersection

<set3>=<set1>.intersection(<set2>)

Note: Intersection returns a set

#Update

<Set1>.update(<Set2>) # adds all the elements of Set2 in Set1

Note: The update() method inserts all items from one set into another.



Jolly Jumper

A sequence of n numbers is called Jolly Jumper if the absolute values of the differences between the successive elements take on all possible values from 1 through $n-1$. The definition implies that any sequence of a single integer is a jolly jumper.

Given a list of n integers, determine whether it is a Jolly Jumper Sequence

Think!!!

Examples:

Input: 1 4 2 3

Output: True

This sequence 1 4 2 3 is Jolly Jumper because
the absolute differences are 3, 2, and 1.

Input: 1 4 2 -1 6

Output: False

The absolute differences are 3, 2, 3, 7.

This does not contain all the values from 1
through n-1. So, this sequence is not Jolly.

Input: 11 7 4 2 1 6

Output: True





Jolly Jumper

If the list has n element, then the set containing the absolute differences between each consecutive elements from the list should be of length $n-1$ and contain all the values from 1 to $n-1$

Solution

Sequence=[3,3,2,1]

```
setJolly=set([])      #Set Containing values from 1 to n-1 inclusive
for i in range(1,len(Sequence)):
    setJolly.add(i)
```

```
setDiff=set([])        #Set Containing Absolute Differences
for i in range(len(Sequence)-1):
    setDiff.add(abs(Sequence[i]-Sequence[i+1]))
```

```
if(setDiff==setJolly):
    print("Jolly Sequence Sequence")
else:
    print("Not a Jolly Jumper Sequence")
```

Dictionaries



Dictionaries

Consider a conventional dictionary, for every word(key), there is a meaning(value) associated with it.

Dictionaries are used to store data values in '*key:value*' pairs.

A dictionary is a collection which is ordered, changeable and do not allow duplicates.

Dictionary Methods

| Method | Description |
|--------------|--|
| clear() | Removes all the elements from the dictionary |
| copy() | Returns a copy of the dictionary |
| fromkeys() | Returns a dictionary with the specified keys and value |
| get() | Returns the value of the specified key |
| items() | Returns a list containing a tuple for each key value pair |
| keys() | Returns a list containing the dictionary's keys |
| pop() | Removes the element with the specified key |
| popitem() | Removes the last inserted key-value pair |
| update() | Updates the dictionary with the specified key-value pairs |
| values() | Returns a list of all the values in the dictionary |
| setdefault() | Returns the value of the specified key. If the key does not exist: insert the key, with the specified value |

Dictionary Examples (Creation)

Consider Module Code and Module Name.

Each Module Code has a module name associated with it.

#Creating and empty Dictionary

```
myModule={}
myModules.update({"ICDT1016Y":"Communication and Business Skills for IT"})
myModules.update({"ICDT1201Y":"Computer Programming",})
myModules.update({"ICT1019Y":"Computer Organisation and Architecture"})
myModules.update({"ICT1208Y":"Software Engineering Principles"})
myModules.update({"ICT1043Y":"Computational Mathematics"})
myModules.update({"ICDT1202Y":"Database Systems"})
```

OR

```
myModules={"ICDT1016Y":"Communication and Business Skills for IT", "ICDT1201Y":"Computer Programming",
"ICT1019Y":"Computer Organisation and Architecture", "ICT1208Y":"Software Engineering Principles",
"ICT1043Y":"Computational Mathematics", "ICDT1202Y":"Database Systems"}
```

OR

```
myModules=dict(ICDT1016Y="Communication and Business Skills for IT",
ICDT1201Y="Computer Programming", ICT1019Y="Computer Organisation and Architecture",
ICT1208Y="Software Engineering Principles", ICT1043Y="Computational Mathematics", ICDT1202Y="Database Systems")
```

Some Dictionary Methods

To retrieve the value associated with a key `<dictionaryName>.get(<key>, <Value if absent>)`

E.g. `print(myModules.get("ICDT1201Y", "Not Found"))`

Note: if second parameter is not supplied, None is returned if key is not present

To retrieve the list of Keys: `<dictionaryName>.keys()`

E.g. `print(myModules.keys())`

To return list of values: `<dictionaryName>.values()`

E.g. `print(myModules.values())`

To update the dictionary `<dictionaryName>.update({<key>: <value>})`

E.g. `myModules.update({"ICDT1000": "Practical"})`

To remove a key:value pair `<dictionaryName>.pop(<key>)`

E.g. `myModules.pop("ICDT1016Y")`

Note: It throws a KeyError if the key is not present

Island-Capital Problem!

Could we solve the Island-Capital Problem using a Dictionary in 3 lines of code???



Island-Capital Problem!

Yes!! The key could be the name of the island and the value would be the name of the capital !!!

```
myDictionary={  
    "Maldives": "Malé",  
    "Sri-Lanka": "Colombo",  
    "Mauritius": "Port Louis",  
    "Seychelles": "Victoria",  
    "Madagascar": "Antananarivo",  
    "Reunion": "Saint-Denis",  
    "Comoros": "Moroni",  
    "Djibouti": "Djibouti City",  
    "Rodrigues": "Port-Mathurin"  
}
```



Island-Capital Using a Dictionary!

```
myDictionary={  
    "Maldives": "Malé", "Sri-Lanka": "Colombo", "Mauritius": "Port  
Louis", "Seychelles": "Victoria",  
    "Madagascar": "Antananarivo", "Reunion": "Saint-Denis",  
    "Comoros": "Moroni", "Djibouti": "Djibouti City",  
    "Rodrigues": "Port-Mathurin"}
```

```
C=input("Enter Name of Country")  
print(myDictionary.get(C, " is not in the Indian Ocean"))
```

Island-Capital Problem!

Could we solve the Island-Capital Problem using a Dictionary in 2 lines of code??



Island-Capital Problem!

Yes!!



```
myDictionary={  
    "Maldives":"Malé", "Sri-Lanka":"Colombo", "Mauritius":"Port Louis",  
    "Seychelles":"Victoria", "Madagascar":"Antananarivo", "Reunion":"Saint-Denis",  
    "Comoros":"Moroni", "Djibouti": "Djibouti City", "Rodrigues": "Port-Mathurin"}
```

```
print(myDictionary.get(input("Enter Name of Country"), " is not in the Indian  
Ocean"))
```

Assuming lists are given - use zip

```
listIslands=["Maldives", "Sri-Lanka", "Mauritius", "Seychelles", "Madagascar",  
"Reunion", "Comoros", "Djibouti", "Rodrigues"]
```

```
listCapitals=["Malé", "Colombo", "Port Louis", "Victoria", "Antananarivo",  
"Saint-Denis", "Moroni", "Djibouti City", "Port-Mathurin"]
```

```
myDictionary={}
```

```
for i,c in zip(listIslands, listCapitals):  
    myDictionary.update({i:c})
```

```
C=input("Enter Name of Country")  
print(myDictionary.get(C, " is not in the Indian Ocean"))
```



Traversing / Iterating in a Dictionary

For key, value in <dictionaryName>.items():

```
    print(key, " - > ", value)
```

for i,c in listIsland.items():

```
    print("Island is ",i,"and capital is ",c)
```



Links

Python Documentation on W3 Schools

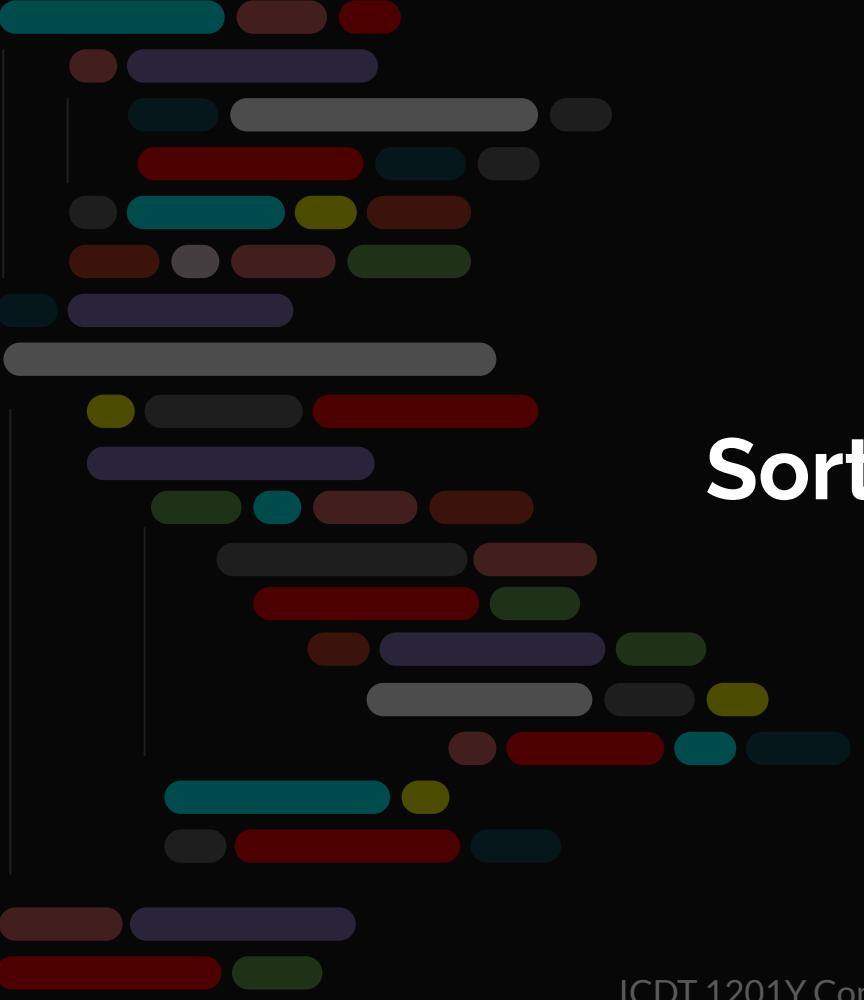
<https://www.w3schools.com/python/>

List, Tuples and Sets:

<https://youtu.be/W8KRzm-HUcc>

Dictionaries:

<https://youtu.be/daefaLgNkw0>



Sorting & Searching Algorithms



Algorithms

In mathematics and computer science, an algorithm (/ˈælgərɪðəm/) is a finite sequence of mathematically rigorous instructions, typically used to solve a class of specific problems or to perform a computation.

The word Algorithm is named after Musa al-Kwarizmi was a renowned Persian mathematician and astronomer who lived in the 9th century. Born in Baghdad, he made significant contributions to the fields of algebra, geometry, and trigonometry.

Source:
Wikipedia

<https://en.wikipedia.org/wiki/Algorithm>



Sorting & Searching

Sorting means to arrange elements in a collection in a certain order (ascending or descending)

Searching means to looking for an element a.k.a the key in a collection of items.

Different Sorting/Searching Algorithms have different efficiencies. The efficiency of an algorithms is usually measured in terms of how the speed and space requirements increase when the size of input increases.



Common Sorting & Searching Algorithms

Sorting

Selection Sort

BubbleSort

Insertion Sort

QuickSort

MergeSort

RadixSort

HeapSort

.

.

.

Searching

Linear/Sequential Search

BinarySearch

.

.

.

In-built sorting and searching

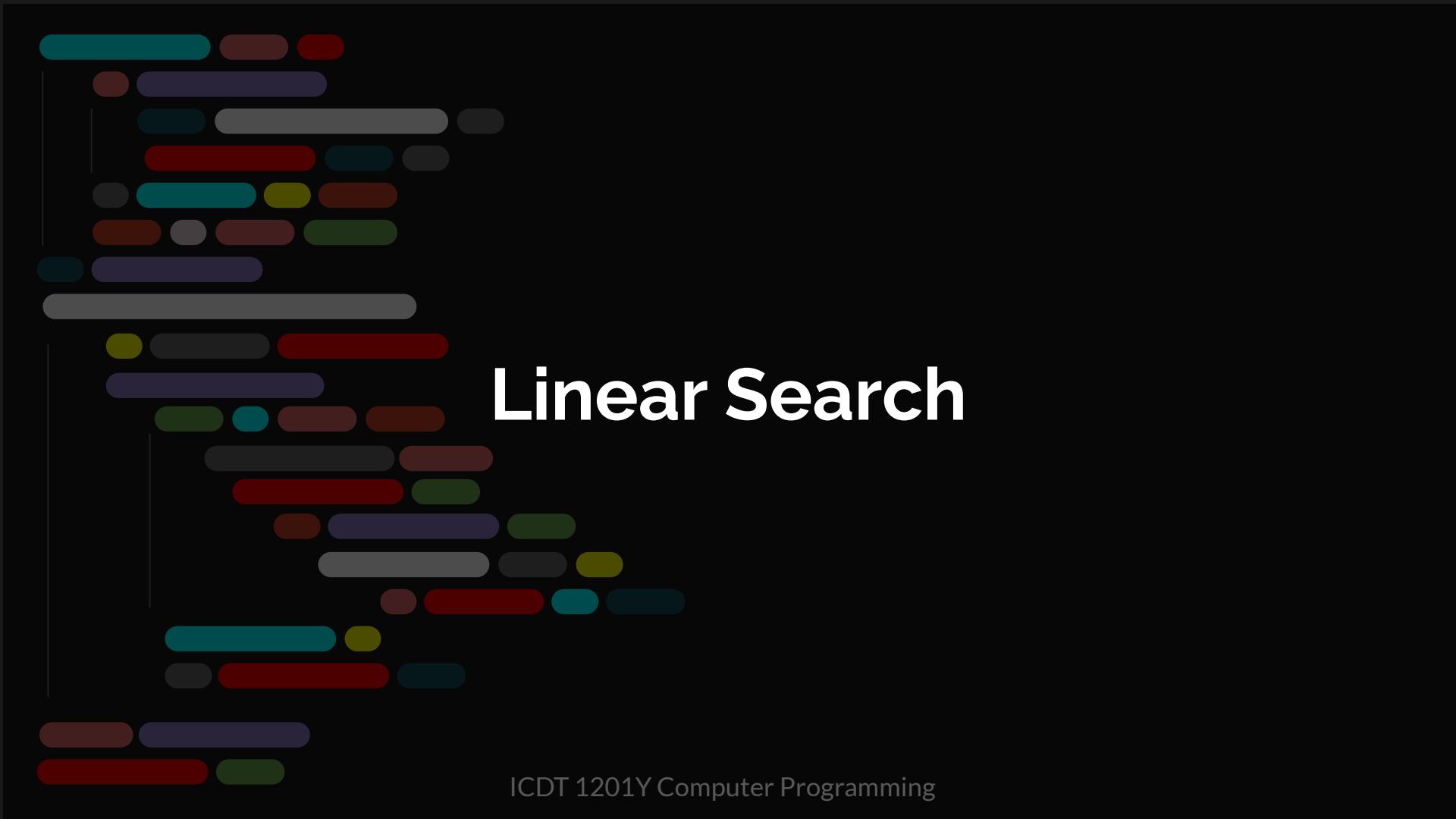
sorted(<ListName>),
returns sorted copy of
the list, original list is
unaltered

<ListName>.sort(), sorts
the list

<ListName>.index(key)
<ListName>.count(key)

<StringName>.find()
<StringName>.count()
<StringName>.index()
<StringName>.rfind()
<StringName>.rindex()

Note: Index() raise a ValueError
if the element if not found



Linear Search



Linear Search

Given a list of elements, Linear Search attempts to look for a value by comparing it to all the elements in the list. Linear Search is not an efficient Searching algorithm. If the list is sorted, a more efficient Searching Algorithm is Binary Search.

Link to Video:
Jenny's Lecture

<https://youtu.be/C46QfTjVCNU>

Linear Search

The algorithm for Linear Search is as follows:

Step 1: Assign the value 0 to a variable, i

Step 2: while $i < \text{length of the list}$ Repeat Step4 3 & 4

Step 3: If element @ position $i == \text{key}$,
 return i

Step 4: Increment i

Step 5: Return -1

Returning Position

```
#Indexing
```

```
def LinearSearch(k,L):  
    i=0  
    for i in range(len(L)):  
        if(L[i]==k):  
            return i  
    return -1
```

```
#Iterate
```

```
def LinearSearch(k,L):  
    i=0  
    for val in L:  
        if val==k:  
            return i  
    i+=1  
    return -1
```

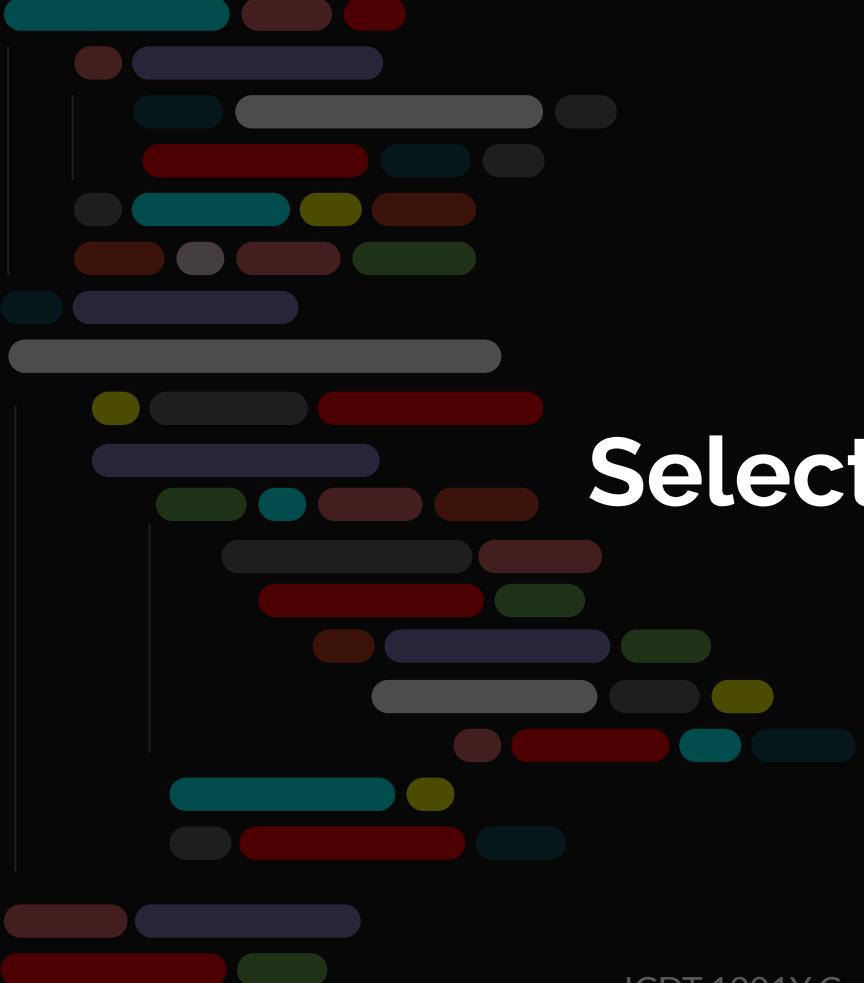
Returning Boolean

#Indexing

```
def LinearSearch(k,L):
    i=0
    for i in range(len(L)):
        if(L[i]==k):
            return True
    return False
```

#Iterate

```
def LinearSearch(k,L):
    for val in L:
        if val==k:
            return True
    return False
```



Selection Sort

Selection Sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundaries by one element to the right.

Source:

https://www.tutorialspoint.com/data_structures_algorithms/selection_sort_algorithm.htm

https://www.w3schools.com/dsa/dsa_algo_selectionsort.php

Link to Video:

Selection Sort | GeeksforGeeks <https://youtu.be/xWBP4lzkoyM>



Selection Sort

Given a list of elements, start with $i=0$
while $i < \text{length of list}$

Start from i , look for index of smallest element,
Store its index in variable j ,

Swap elements at position i with element at position j

Increment i

Selection Sort

i=0

while i<length of list

Loop into array Starting at i,
Look for smallest element,
Store its index in variable j,

Swap elements in slots i and j

Increment i

Link: https://www.w3schools.com/dsa/dsa_algo_selectionsort.php

```
def SelectionSort(L):
```

```
    i=0
```

```
    while i<len(L)-1:
```

```
        smallest=i
```

```
        j=i+1
```

```
        while(j<len(L)):
```

```
            if L[j]<L[smallest]:
```

```
                smallest=j
```

```
            j+=1
```

```
            L[i],L[smallest]=L[smallest]
```

```
,L[i]#swap
```

```
j+=1
```

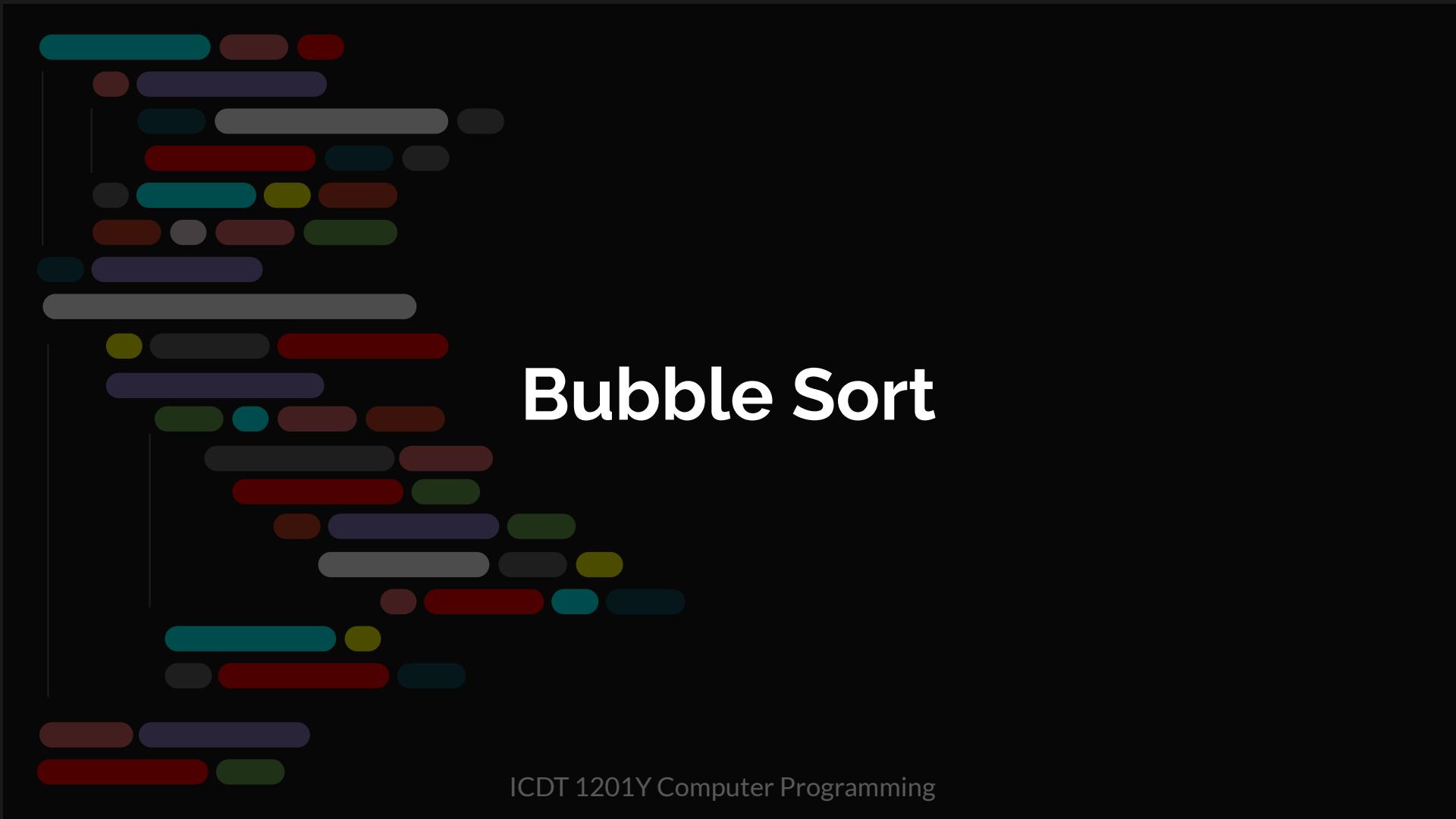
```
return L
```

Selection Sort

```
def SelectionSort(L):
    i=0
    while i<len(L)-1:
        smallest=i
        j=i+1
        while(j<len(L)):
            if L[j]<L[smallest]:
                smallest=j
            j+=1
        L[i],L[smallest]=L[smallest],L[i] #Swapping values at indices i and smallest
        i+=1
    return L
```

#Assuming smallest is at index i
#Start from slot just after i
#Looking for index of smallest value in remaining
#Comparing every value at index j with smallest
#if value at j<smallest, make j =index of smallest

Link: https://www.w3schools.com/dsa/dsa_algo_selectionsort.php



Bubble Sort



Bubble Sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. After each pass i , the largest value will be pushed to the position $\text{length}-i-1$

Link:

https://www.w3schools.com/dsa/dsa_algo_bubblesort.php

https://www.w3schools.com/dsa/dsa_algo_bubblesort.php

Link to Video:

Bubble Sort Algorithm | GeeksforGeeks

<https://youtu.be/nmhjrl-aW5o>



Bubble Sort

Starting from $i=0$, to $i < \text{length}-1$

Start with $j=0$ to $j < \text{length}-i-1$

if $\text{Array}[j] > \text{Array}[j+1]$

Swap elements at position j and $j+1$

Increment j

Increment i

Bubble Sort

How it works:

1. Loop in the list starting from $i = 0$ to $\text{length}-1$
2. Loop starting from $j=0$ to $j < \text{length}-i-1$
3. compare the value v_j with the next value v_{j+1} . If the v_j is larger, swap the values at slots v_j and v_{j+1}
4. Go through the array as many times as there are values in the array.

```
def BubbleSort(L):
    for i in range( len(L)-1):
        for j in range(len(L)-i-1):
            if L[j]> L[j+1]:
                L[j+1],L[j]=L[j],L[j+1]
    return L
```

Link:

https://www.w3schools.com/dsa/dsa_algo_bubblesort.php

Bubble Sort

```
def BubbleSort(L):
    for i in range( len(L)-1): #stop at len-1 since j will add 1
        for j in range(len(L)-i-1):# stop at len-1 since j will add 1
            if L[j]>L[j+1]: # larger value is sent towards right
                L[j+1],L[j]=L[j],L[j+1] #swap
    return L
```

Optimising Bubble Sort

If in the inner loop, no swap was performed, this means that the array is already sorted!

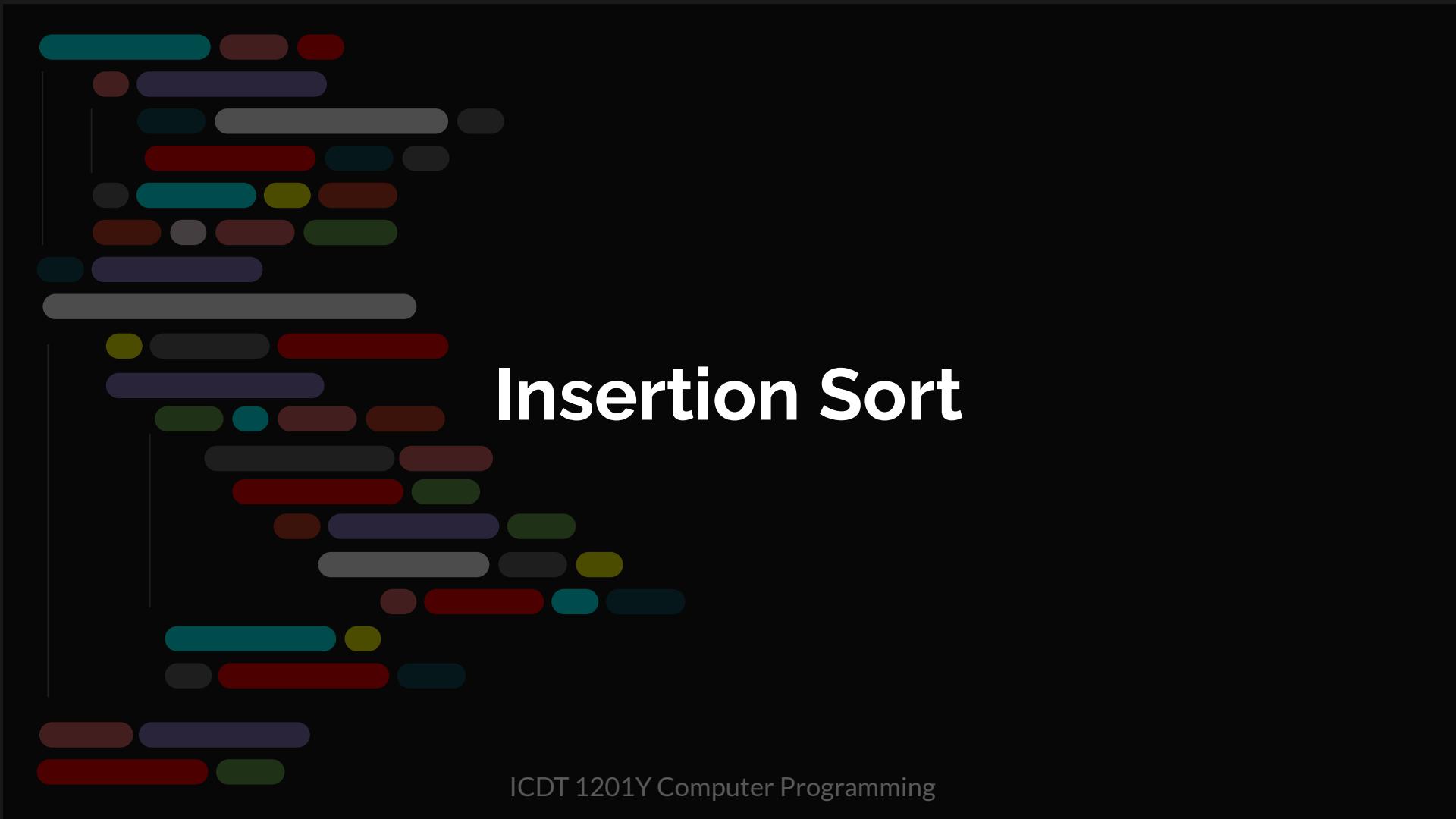
```
def BubbleSort(L):
    for i in range( len(L)-1):
        swap=False
        for j in range(len(L)-i-1):
            if L[j]>L[j+1]:
                L[j+1],L[j]=L[j],L[j+1]
                swap=True
        if swap==False:
            break
    return L
```

#stop at len-1 since j will add 1

stop at len-1 since j will add 1

larger value is sent towards right

#swap



Insertion Sort



Insertion Sort

Insertion sort is a very simple method to sort numbers in an ascending or descending order.

It can be compared with the technique how cards are sorted at the time of playing a game.

Source:

[https://www.tutorialspoint.com/data structures algorithms/insertion sort algorithm.htm](https://www.tutorialspoint.com/data_structures_algorithms	insertion_sort_algorithm.htm)

[https://www.w3schools.com/dsa/dsa algo insertionsort.php](https://www.w3schools.com/dsa/dsa_algo_insertionsort.php)

Link to Video:

Selection Sort | GeeksforGeeks

<https://youtu.be/OGzPmgsI-pQ>

Insertion Sort

Step 1 – loop i from 1 to length of list

Step 2 – insertion point is position 1, element popped = element i

Step 3 – loop j starting from position (i-1), until 0 **inclusive**

Step 4 – if element at position j > the popped element,

Step 5 – insertion point=j (move towards the left)

Step 6 – #else break as we found the insertion point

Step 7 – insert the popped item at the insertion point.

Source: https://www.tutorialspoint.com/data_structures_algorithms/insertion_sort_algorithm.htm

Insertion Sort

How it works:

1. **Step 1** – loop i from 1 to length of list
2. **Step 2** – insertion point is position i,
element popped = element i
3. **Step 3** – loop j starting from position
(i-1), until 0 **inclusive**
4. **Step 4** – if element at position j>
the popped element,
5. **Step 5** – insertion point=j
(move towards the left)
6. **Step 6** – else break as we found the
insertion point
7. **Step 7** – insert the popped item at the
insertion point.

```
def InsertionSort(L):  
  
    for i in range(1,len(L)):  
        insertPoint=i  
        item=L.pop(i)  
  
        for j in range(i-1,-1,-1):  
            if L[j]>item:  
                insertPoint=j  
            else:  
                break  
        L.insert(insertPoint,item)  
    return L
```

Insertion Sort

```
def InsertionSort(L):\n    for i in range(1,len(L)):\n        insertPoint=i\n        item=L.pop(i)\n\n        for j in range(i-1,-1,-1):\n            if L[j]>item:\n                insertPoint=j\n            else:\n                break\n        L.insert(insertPoint,item)\n\n    return L
```

#loop i from 1 to length of list

#insertion point is position i,
#element popped = element i

#loop j starting from position (i-1), until 0 **inclusive**

#if element at position j> the popped element,
#insertion point=j (move towards the left)

#else break as we found the insertion point

#insert the popped item at the insertion point.



Reversing a list

Reversing a list

Sorting usually arranges the list in ascending order, if you need the list in Descending order, you may reverse the array after sorting it.

5 Methods for Reversing:

- 1) <ListName>.reverse() - Reverses the list
- 2) reversed(<listName>) - returns a reversed copy of, Original List Unchanged
- 3) <ListName>[::-1]returns a reversed copy of, Original List Unchanged
- 4) In Place Swapping
- 5) Traverse and insert at beginning of another list

```
L=[1,2,3,4,5,9]
print("Swap Symmetry")
l=len(L)
mid=int(l/2)
for i in range(mid):
    L[i], L[l-i-1]=L[l-i-1],L[i]
print(L)
```

```
L=[1,2,3,4,5,9]
reversedL= []
for i in L:
    reversedL.insert(len(L)-1,i)
print(reversedL)
```

Stack and Queues

Stacks and Queues

Stack



Queue





Stacks and Queues

- A stack is a collection
- It is a data structure that contains multiple elements
- A stack is “Last In, First out” or LIFO data structure
- The last item added is the first one to be removed

- A queue is a collection
- It is a data structure that contains multiple elements
- A queue is “First In, First out” or FIFO data structure
- Simplest queuing policy



Stacks

The stack can be visualised as a vertical pile.

Items are pushed to the stack at the top

Items are popped from the stack from the top.

We can check if the stack is empty prior to popping

All these operations can be implemented using list methods.

A push would be an append, a pop would first need to check if the stack is empty (check if length ==0), then remove the last element using pop() or pop(len(<listName>)-1)



Methods for Stack class

1. Constructor
2. IsEmpty() - Returns boolean
3. Push(<Element>) - accepts an element as argument
4. Pop() - Removes the element at the top if the stack is not empty, returns none otherwise
5. Peek() - Displays the element at top of the stack
6. Clear() - clears the stack

Visualisation

```
class Stack:  
    def __init__(self):  
        self.listStack=[]  
  
    def isEmpty(self):  
        return len(self.listStack)==0  
  
    def push(self, data):  
        self.listStack.append(data)  
        self.printStack()  
  
    def pop(self):  
        if not self.isEmpty():  
            self.listStack.pop()  
        else:  
            return None  
        self.printStack()
```

```
def printStack(self):  
    print("Top\n\n")  
    for i in self.listStack:  
        print(str(i))  
    print("\n\nEnd")  
  
def peek(self):  
    lastIndex=len(self.listStack())-1  
    return self.listStack[lastIndex]  
  
def clear(self):  
    self.listStack.clear()  
    self.printStack()
```



Queue

The Queue can be visualised as a linear collection of items.
Items are added to the end of the queue - add/enqueue
Items are removed from the front of the queue - remove/dequeue
We need to check if the queue is empty prior to removing

All these operations can be implemented using list methods.

An add/enqueue would be an append().
A remove/dequeue would first need to check if the queue is not empty (check if length $<> 0$), then remove the front element using pop(0)



Method for Queue Class

1. Constructor
2. IsEmpty() - Returns boolean
3. enQueue(<Element>) - accepts an element as argument and adds at the rear of the queue
4. deQueue() - Returns the first element if the queue is not empty, returns none otherwise
5. Peek() - Displays the element at front of the queue
6. Clear() - clears the queue

Visualisation

```
class Queue:  
    def __init__(self):  
        self.listQueue=[]  
  
    def isEmpty(self):  
        return len(self.listQueue)==0  
  
    def enQueue(self, data):  
        self.listQueue.append(data)  
        self.printQueue()  
  
    def deQueue(self):  
        p=None  
        if not self.isEmpty():  
            p=self.listQueue.pop(0)  
        self.printQueue()  
        return p
```

```
def peek(self):  
    return self.listQueue[0]  
  
def printQueue(self):  
    print("Front> \t", end="")  
    for i in self.listQueue:  
        print("\t-\t"+str(i), end="")  
    print("\t< End")  
  
def clear(self):  
    self.listQueue.clear()  
    self.printStack()
```