COMP07027 Introduction to Programming

School of Computing, Engineering and Physical Sciences Scotland Academy Wuxi

Dr Muhammad Aslam
Lecturer UWS Wuxi
Muhammad.Aslam@uws.ac.uk

Lecture 1

Agenda of the lecture 1

01 **Introduction to Module COMP07027**

02 **Introduction to the Computer programming**

03 **Basics of Programming**

1-Introduction to Module

Welcome to the Module

- This is a core module for the undergraduate programmes in Cyber Security, Web and Mobile Development.
- The module is designed for 16 weeks.
- This is an entry-level programming module and aims to introduce the skills required to write simple structured programs in a high-level language and assess these skills in practical situations.
- The lectures will include topics such as data and variables, input/output, data structures, iterations and structured code.
- We also focus on object-based programming, debugging and troubleshooting, programming efficiency
- Future modules in the technical computing programmes assume knowledge of basic programming principles and an ability to create simple structured programs.
- In lab sessions, we will install and execute PyCharm and Jupyter Notebook for writing simple programming codes of Python.

Weekly Timeline

	Week
Introduction to the Module and Basics	1 - 2
Data and Variables	2
Selection	3
Iteration 1	4
Iteration 2	5
Sample solution to number guessing game (with suggestions)	6
Functions	7
Menus and Methods	8
Files	9
Assessment 1	10
OOP 1	11
OOP 2 Methods	12
OOP3 Structures	13
Inheritance & Encapsulation OOP4	14
Testing & Error Handling	15
Assessment 2	16

Learning Outcomes

L1

Develop understanding of all the key concepts of programming (Python)

L2

Write small structured programs in a high level programming language

L3

Write small structured programs in a high level programming language.

L4

Create a simple console-based user-interface for a program and use this to create interactive software

Leaning and Teaching

Learning and Teaching

Lectures will be used for exposition of topics, provide context and suggest appropriate background material. Lab sessions, using pair programming, will provide practical experience in developing small software system

Learning Activities **During completion of this module, the learning activities undertaken to achieve the module learning outcomes are stated below:**	Categories	Student Learning Hours (Normally totalling 200 hours): (Note: Learning hours include both contact hours and hours spent on other learning activities)
Lecture/Core Content Delivery	Scheduled	24
Laboratory/Practical Demonstration/Workshop	Scheduled	32
Independent Study	Independent	144
		200 Hours Total

Indicative Resources

The following materials form essential underpinning for the module content and ultimately for the learning outcomes:
McMonnies, A. (2016) Introduction to Programming Module Notes (Python)

Recommended reading:

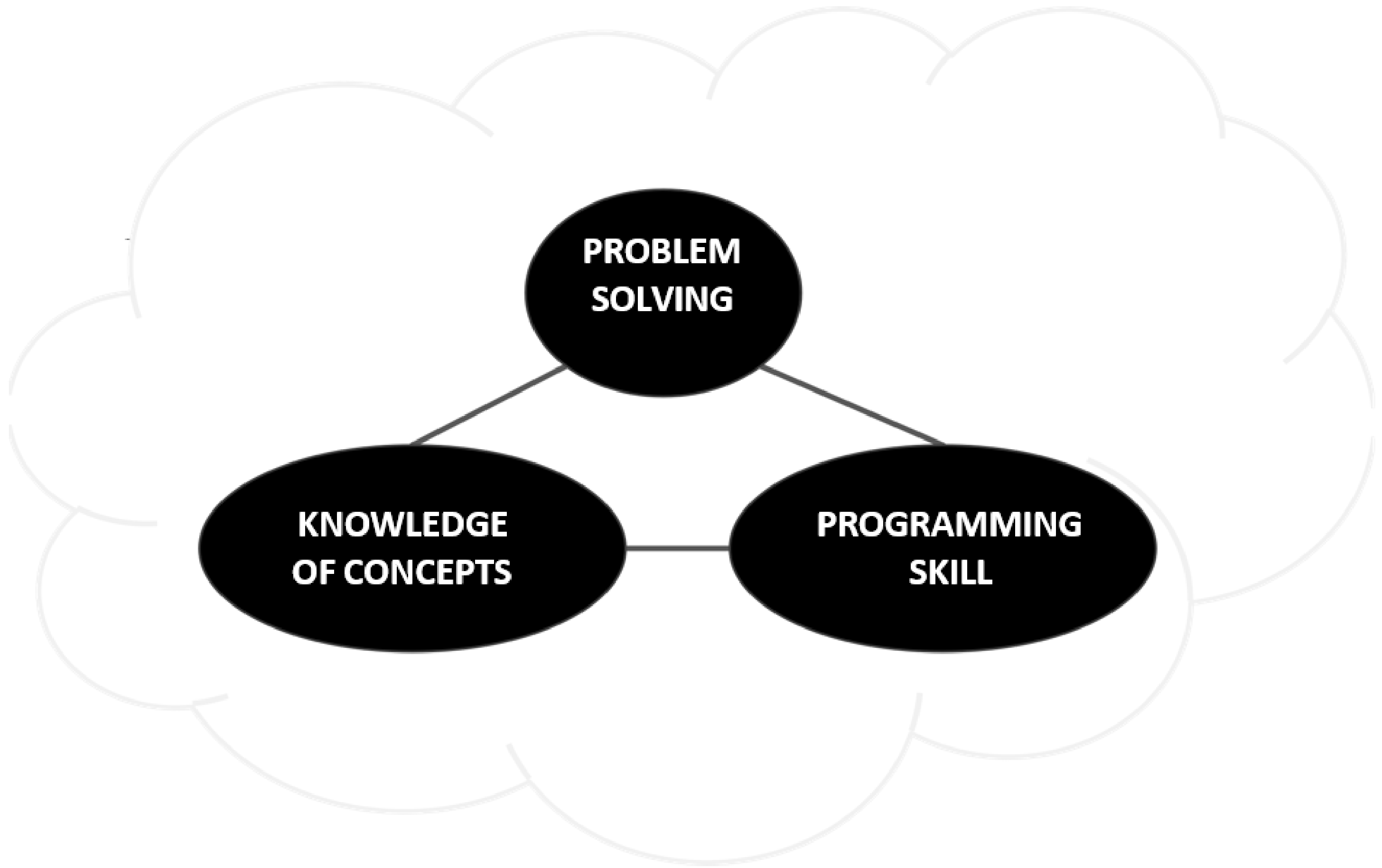
Mike Dawson (2010) Python Programming for the Absolute Beginner. 3rd Edition. Course Technology PTR

Shaw, Z.A. (2013) Learn Python the Hard Way: A Very Simple Introduction to the Terrifyingly Beautiful World of Computers and Code (Zed Shaw's Hard Way). 3rd Edition. Addison-Wesley

Learning Tips

- Practice, Practice, Practice…..
- can’t passively absorb programming as a skill.
- download the slides before lecture and follow along.
- Actively participate at course platform and complete the tasks in time.
- Don’t be afraid to try out Python commands.

Programming language Basic Goal



2-Introduction to Programming with Python

What does Computer do?

- Fundamentally:
  - performs **calculations**

    a billion calculations per second!
  - **remembers** results

    100s of gigabytes of storage!
- What kinds of calculations?
  - **built-in** to the language
  - ones that **you define** as the programmer
- computers only know what you tell them

Types of Knowledge

- **declarative knowledge** is **statements of fact**.
  - We arranged a lecture for Introduction to computer programming
- **imperative knowledge** is a **recipe** or “how-to”.

1) Students sign up for the class
2) Students complete the enrolment process
3) We arrange the zoom meting to deliver the first lecture
4) We start our lecture on scheduled time

COMPUTERS ARE MACHINES

- How to capture a recipe in a mechanical process
- **Fixed program** computer
  - Calculator

- **Stored program** computer
  - machine stores and executes instructions

Computing System Key Components

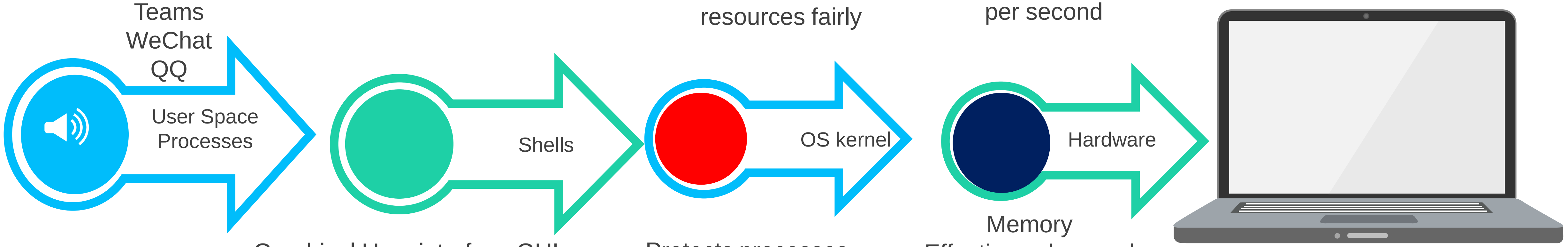
These are the Apps that you love to use in daily life on Computing devices
Web browser
Skype
Zoom
Teams
WeChat
QQ

These are programs to access the control over these Apps, and perform start, stop, pause

Shells interact with OS kernel to get computer system resources

Kernel can share the resources fairly

Hardware
Central Control Processor (Brain of the system)
Incredibly fast to process billions of instructions per second



Graphical User interface GUI,

Command Line interfaces CLI

These are controlling commands to use Apps and interact these commands with Computer Operating System

Protects processes from each other

Operating System utilize the kernel to provide effective access to resources

You can use multiple Apps at the same time

Memory
Effective ephemeral memory

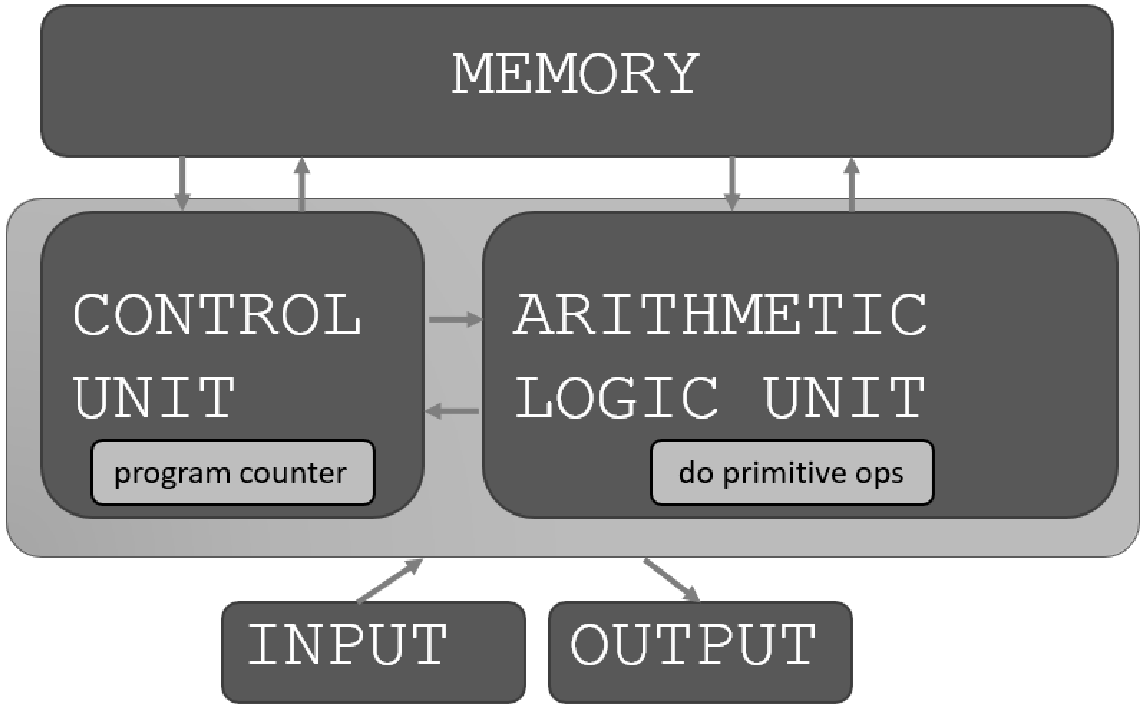
Memory has improved over the past decades

Storage
Long term, slower and persistent
Hard-drive

Computer System has Input and output hardware

Basic Machine Architecture

- Input/ Output: User interface
- Memory: contains data and stored programs with sequence of instructions
- ALU: responsible for primitive operations like addition, subtraction, division, multiplication
- Control Uni: Program counter and



Creating recipes of programming language

- A programming language provides a set of primitive **operations**
- **expressions** are complex but legal combinations of primitives in a programming language
- expressions and computations have **values** and meanings in a programming language

Programming language

- A program is simply a list of instructions, written in a (very precise) language that the computer can understand, and is designed to carry out some function.

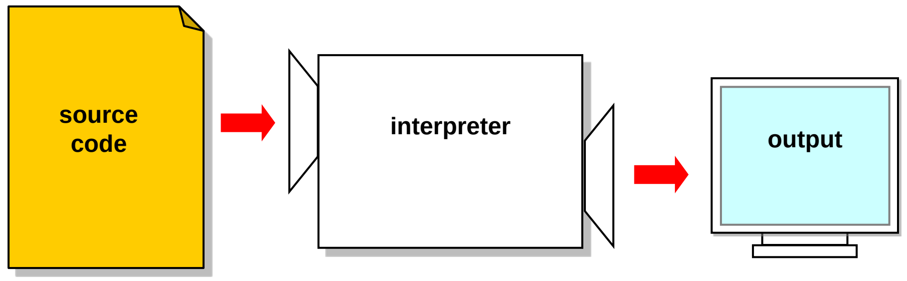
- Most programs fit the same basic model …….

High-Level Language

- Python is a high level language.
- This means that code is written so that we can (sort of) understand it.
- It is also much easier for us to write.
- However, before the computer can understand it, the code needs to be converted to a low-level language (sometimes called machine or assembly language).
- This is the job of ***interpreters*** and ***compilers***.

Interpreted

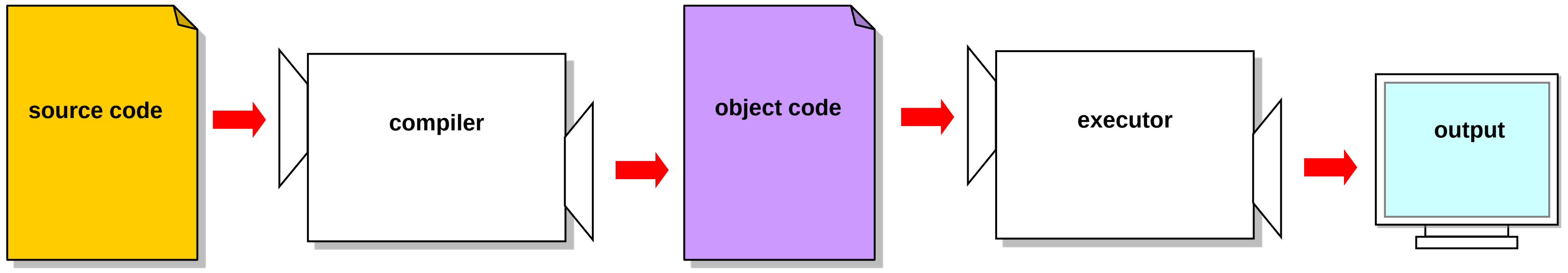
- An interpreter reads a high-level program and executes it a wee bit at a time.
- It reads a line, executes it, then reads the next line and executes it, then reads the next line etc etc till the end.



- IronPython
- Jython
- Cpython
- IronPython

Compiled

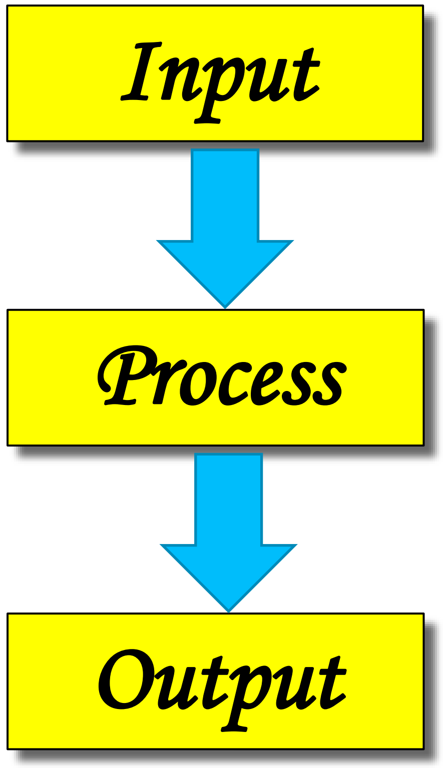
- A compiler reads the whole program (the **source** code), and translates it into **object** code (or **executable** code).
- When compiled, the program can be executed repeatedly without being compiled again.



Basic difference between interpreter and compiler

Interpreter translates just one statement of the program at a time into machine code.	Compiler scans the entire program and translates the whole of it into machine code at once.
An interpreter takes very less time to analyze the source code. However, the overall time to execute the process is much slower.	A compiler takes a lot of time to analyze the source code. However, the overall time taken to execute the process is much faster.
An interpreter does not generate an intermediary code. Hence, an interpreter is highly efficient in terms of its memory.	A compiler always generates an intermediary object code. It will need further linking. Hence more memory is needed.
Keeps translating the program continuously till the first error is confronted. If any error is spotted, it stops working and hence debugging becomes easy.	A compiler generates the error message only after it scans the complete program and hence debugging is relatively harder while working with a compiler.
Interpreters are used by programming languages like Ruby and Python for example.	Compliers are used by programming languages like C and C++ for example.

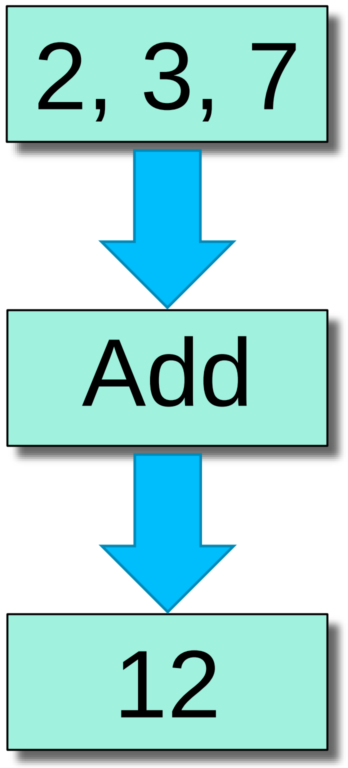
Programming language



Input often comes from the keyboard or mouse, but can also come from another device or a file.

The process is the operation(s) that the program has been created to carry out.

The Output is the result of performing the *Process* on the *Input(s).*



Aspects of Language language

- **primitive constructs**
  - English: words
  - programming language: numbers, strings, simple operators

- **syntax**
  - English: `"cat dog boy"` → not syntactically valid

    `"cat hugs boy"` → syntactically valid
  - programming language: `"hi"5` → not syntactically valid

    `3.2*5` → syntactically valid

- **Syntax Errors**
- Syntax is the set of rules that govern how language is used and understood.
- It’s a bit like spelling and grammar in English.
- However, where we can often understand a sentence in English that is not perfectly grammatical, or contains spelling mistakes, Python is not so forgiving.
- **Python demands perfection**!

Aspects of Language language

- **static semantics** is which syntactically valid strings have meaning
  - English: `"I are hungry"` → syntactically valid
    but static semantic error
  - programming language: `3.2*5` → syntactically valid
    `3+"hi"` → static semantic error
- **semantics** is the meaning associated with a syntactically correct string of symbols with no static semantic errors
  - English: can have many meanings `"Flying planes can be dangerous"`
  - programming languages: have only one meaning but may not be what programmer intended

Major possible errors

- **syntactic errors**
  - common and easily caught
- **static semantic errors**
  - some languages check for these before running program
  - can cause unpredictable behavior
- no semantic errors but **different meaning than what programmer intended**
  - program crashes, stops running
  - program runs forever
  - program gives an answer but different than expected

- **Semantic Errors**
- Semantic errors can be really tricky to spot.
- The program will run properly,
- and it won’t generate any error messages.
- **It just gives you the wrong answer!!**

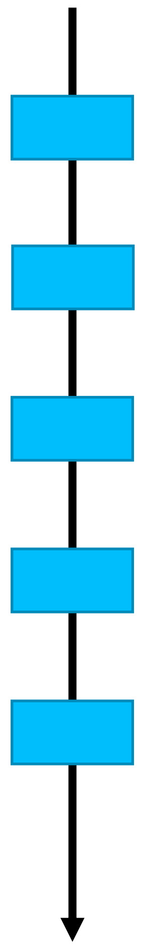
- We’re going to work out a strategy for finding this type of error – more later.

Python program

- 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
- Almost every program uses one, or more likely a combination, of only three constructs!!!
- **Sequence**, **Selection** and **Iteration**.

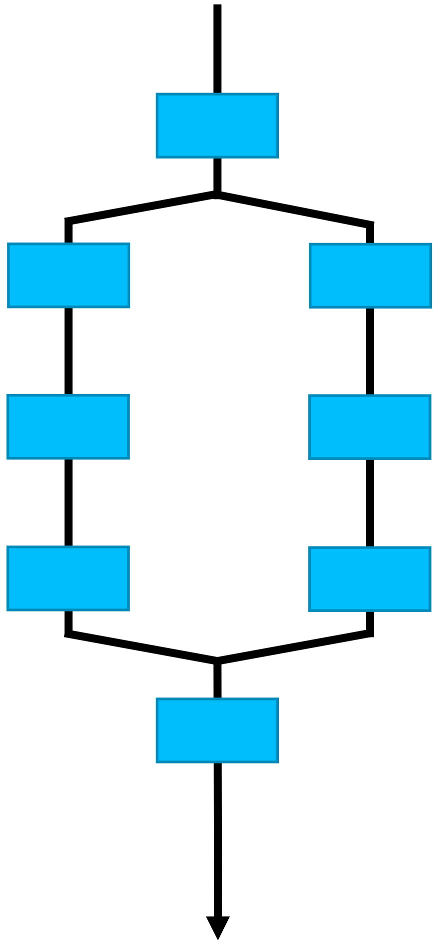
- **<u>Sequence</u>**

- This is the simplest construct.
- It means carrying out a set
- of instructions in a fixed order,
- and that order never changes.



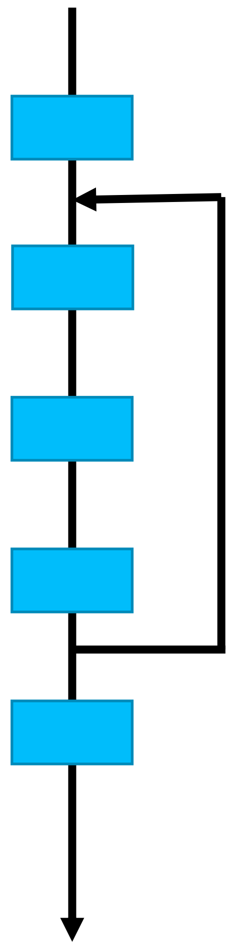
- **<u>Selection</u>**

- Selection involves choosing which
- one set instructions to perform, and
- ignoring the other.
- More on this in the coming weeks.



- **Iteration**

- Iteration is sometimes called looping
- and, as the name suggests, involves
- performing some of the instructions
- more than once.
- More of this later too.



Programming spectrum

- Represent knowledge with data structures
- Iteration and recursion as computational metaphors
- Abstraction of procedures and data types
- Organize and modularize systems using object classes and methods
- Different classes of algorithms, searching and sorting
- Programming complexity and Debugging Problem solving

Creating Programs

- Most programs involve using all of these structures, often imbedded within each other.
- Programs can sometimes get a bit complicated, and, because of that, errors creep in, and it can often take time to find and fix these errors.
- It will help with the process of fixing errors (***debugging***) if we understand what types of errors can arise.

3-Basic building blocks of Programming

Objects

- programs manipulate **data objects**
- objects have a **type** that defines the kinds of things programs can do to them
  - Ana is a human so she can walk, speak English, etc.
  - Chewbacca is a wookie so he can walk, “mwaaarhrhh”, etc.
- objects are
  - scalar (cannot be subdivided)
  - non-scalar (have internal structure that can be accessed)

Scaler Objects

- `int` – represent **integers**, ex. `5`
- `float` – represent **real numbers**, ex. `3.27`
- `bool` – represent **Boolean** values `True` and `False`
- `NoneType` – **special** and has one value, `None`
- can use `type()` to see the type of an object

```
 >>>
type(5)
 Int
 >>> type(3.0)
 float
```

Convert type

- can **convert object of one type to another**
- `float(3)` converts integer `3` to float `3.0`
- `int(3.9)` truncates float `3.9` to integer `3`

Printing to console

- to show output from code to a user, use `print` command

```
In [11]: 3+2
Out[11]: 5

In [12]: print(3+2)
5
```

Expressions

- **combine objects and operators** to form expressions
- an expression has a **value**, which has a type
- syntax for a simple expression

```
<object> <operator> <object>
```

OPERATORS ON ints and floats

- `i+j` → the **sum**
- `i-j` → the **difference**
- `i*j` → the **product**
- `i/j` → **division**

For +, -, *if both are ints, result is int
if either or both are floats, result is float

For / result is float

- `i%j` → the **remainder** when `i` is divided by `j`
- `i**j` → `i` to the **power** of `j`

SIMPLE OPERATIONS

- parentheses used to tell Python to do these operations first
- **operator precedence** without parentheses
  - **
  - *
  - /
  - + and – executed left to right, as appear in expression

BINDING VARIABLES AND VALUES

- equal sign is an **assignment** of a value to a variable name

$$\overset{\text{variable}}{\boxed{\texttt{pi}}} \texttt{ = } \overset{\text{value}}{\boxed{\texttt{3.14159}}}$$

```
pi_approx = 22/7
```

- value stored in computer memory
- an assignment binds name to value
- retrieve value associated with name or variable by invoking the name, by typing `pi`

ABSTRACTING EXPRESSIONS

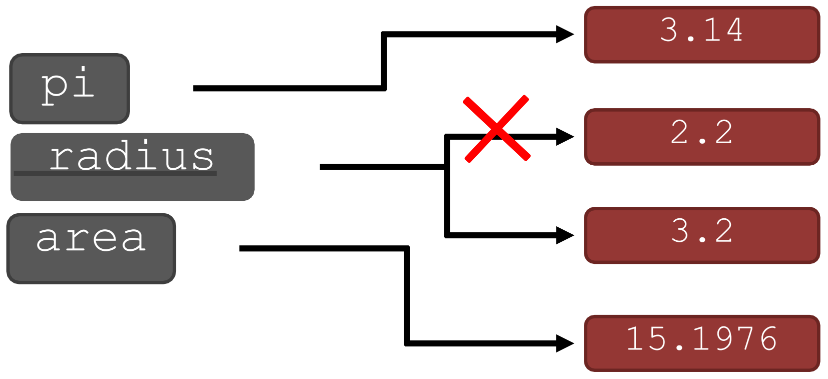
- why **give names** to values of expressions?
- to **reuse names** instead of values
- easier to change code later

```
pi = 3.14159
radius = 2.2
area = pi*(radius**2)
```

CHANGING BINDINGS

- can **re-bind** variable names using new assignment statements
- previous value may still stored in memory but lost the handle for it
- value for area does not change until you tell the computer to do the calculation again

```
pi = 3.14
radius = 2.2
area = pi*(radius**2)
radius = radius+1
```



Before the next class:

1. Install PyCharm at home.
2. Create a project called Labs containing all the programs from the lecture slides
3. Create a project called Lab1 containing all the exercises in Lab Sheet 1.

Thank You