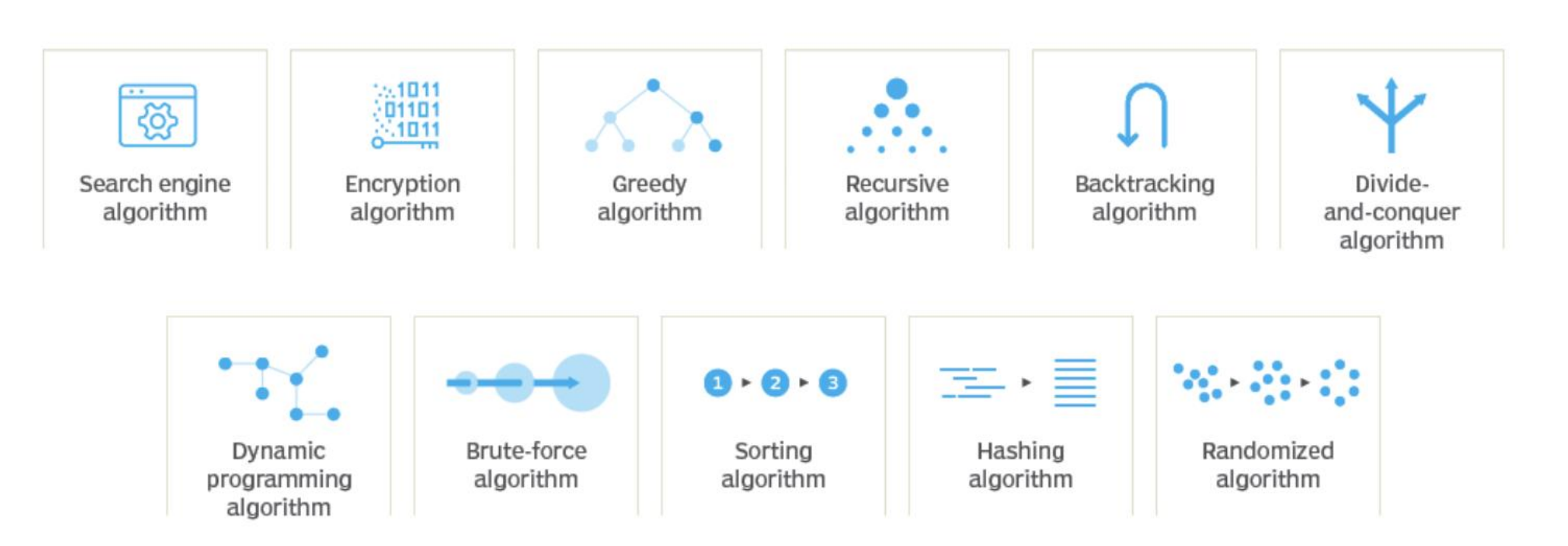
Topic Areas over the next 2 years

- Programming constructs – sequence, selection and iteration
- Working with stings and files
- Modular design – functions and procedures
- Data Structures –arrays, lists, tuples, queues, stacks, linked lists trees and graphs
- GUI Design
- Algorithms – Sorting & Searching algorithms
- Shortest path algorithms
- Recursive Algorithms
- Big O – Time and space efficiency
- Programming standards
- Object oriented programming

What is an algorithm?

An algorithm is a set of rules or a sequence of steps specifying how to solve a problem.



Learning Aims

- Define what is meant by pseudocode
- Learn how and when different data types are used
- Learn the arithmetic, relational and logical operations available in a typical programming language
- Become familiar with basic string-handling operations
- Distinguish between variables and constants
- Learn about how to use selection, sequence and iteration

Programming Constructs

- **Sequencing** Do one statement after the other

- **Iteration** Do a set of statements multiple times.

Iteration is either **'count controlled**' or **'condition controlled**', Repeat until

- **Branching/Selection** Do a set of instructions based on conditions e.g If ... Else. Switch Case statement

Key concepts to remember

Term	What it means	How to do it in Python

Iteration	A loop. There are two loops –
	
	Conditional loops or while loops (Repetition)
	
	for loops (a set number of iterations)
		`number = 0`
		`while number <= 5:`
		`  print("Hello")`
		`  number = number + 1`
		`print("Goodbye")`
		
		`for line in range(4):`
		`  print("hello")`

Variable	Something you can give a value to and then change it at other times in the program.	`name = "Rhiannon" # name is a variable` `number = 56 # number is a variable`

Selection	Where there is a choice point in the program design and an if statement is used to create more than one possible pathway.	`if answer == "Paris":`
		`  print("Correct")`
		`else:`
		`  print("Not correct")`

Input/Output	Getting input from the keyboard or outputting something to the screen.	`name = input("What is your name?")` `print(name)`
Assignment	Where a variable is given a value	`number = 56 # number is assigned the value 56` `name = "Rhiannon" # name is assigned the value Rhiannon`

Key concepts to remember

Selection	Controlling the flow of execution in programs using if and Switch/Case statements
Iteration	Conditional loops or while loops (Repetition) for loops (a set number of iterations)
Condition	Used to control the flow of execution in a program. A condition contains a logical expression. An expression that results in either True or False.
Relational operators	== (equal to) != (not equal to) > (greater than) < (less than) >= (greater than or equal to) <= (less than or equal to) Used in conditions, if x > 10:
Logical operators	AND → True if **both** conditions are True OR → True if **at least one** condition is True NOT → Reverses the Boolean value (True → False, False → True) Used in logic: if (x > 10) AND (y < 5):

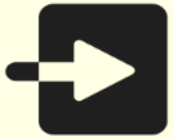
Data Types

- The basic data types which can be used in a high-level programming language are as follows:

Data Type	Description	Example
Boolean	True or false (or any data that only has two possible values)	**True** or **False**
Character	Any single character you see on the keyboard (and more)	'**A**', '**z**', '**8**','**?**'
String	A number of characters	"**Hello World**"
Integer	Whole numbers, positive and negative	**23**, **-45**, **0**
Real/ Float	A number which can contain a fractional part	**15.7**, **-19.25**, **8**

Arithmetic Operators

Operator	Meaning	Example Code	Result
+	Addition	5 + 3	8
-	Subtraction	10 - 4	6
*	Multiplication	7 * 2	14
/	Division (float result)	9 / 2	4.5
//	Floor Division	9 // 2	4
%	Modulus (remainder)	9 % 2	1
**	Exponent (power)	2 ** 3	8



Converting data types (Casting)

When asking for an input the value will always be a string:

```
num = input()
```

To do a calculation the num has to be converted to an integer:

```
calculation = int(num) * 2
```

Calculation will need to be converted back to a string if you print like this:

```
print("Answer" + str(calculation)
```

However not if you do this:

```
print("Answer", calculation)
```

The Round function

You can round this number using a function round.

```
billBetween3 = round(billBetween3,2) #round to 2 decimal places
```

This will return the value 6.67.

String-handling functions

Programming languages have a number of built-in string-handling methods or functions. Some of the common ones in a typical language are:

len(string)	Returns the length of a string
string.find(str)	Determines if str occurs in string. Returns index (the position of the first character in the string) if found, and -1 otherwise. In our pseudocode we will assume that string (1) is the first element of the string, though in Python, for example, the first element is string (0)
ord("a")	returns the integer value of a character (97 in this example)
chr(97)	returns the character represented by an integer ("a" in this example)

To concatenate or join two strings, use the + operator.

name = "Johnny" + "Bates"

Variables and constants

- **A variable is a name/identifier which represents a value in a program**
- ... points to a **memory location**
- ... and the value be changed (while the program is running)
- ….variable can have no value assigned at design time
- …..you can assign a value to a variable at **run-time**

- **A constant is also a name/identifier which represents a value in a program**
- ... points to a **memory location**
- ….the value of a constant cannot be changed once the program is running/can only be set at **design time**
- …..the value of a constant is set when the constant is declared

Variables are identifiers

Variables are **identifiers** (names) given to **memory locations** whose contents will change during **runtime**

Need to know Key Terms

Identifier: the name given to a variable: e.g. score, numberOfLives.

Declare: Some programming languages require variables to be declared first before they can be used. When declaring a variable you give it an **identifier** (and a **data type** for some languages) e.g.
int score

Initialise: To give/**assign** a variable its **first** value which can be changed later on in the program. e.g.
score = 0

Assign: To give or change the value of a variable, using the assignment operator (=) e.g.
numberOfLives = 3

Casting: to convert/change the data type of a variable e.g. in Python using functions such as int(), str(), float().

Increment: To increase the value a variable by a value *score += 1*

Decrement: To decrease the value a variable by a value
timer -= 1

Guidelines could include:

- Start all variable names with a lowercase letter
- Do not use underscores in the middle of variable names
- Use "camelCaps" to separate parts of a variable name – for example, timeInMinutes, maxTemperature
- Do not use overly long names but keep them meaningful – maxTemp is better than maximumTemperature if there is not likely to be any confusion over the meaning of max
- Use all uppercase letters for constants, which are then instantly identifiable
- When defining a class in object-oriented programming, start with an uppercase letter, with the rest of the class name lowercase

Following guidelines such as these will save a lot of time in looking through a program to see whether you called something best_score, Best_Score, bestScore or some other variation.

Selection

	Pseudocode	Python	

if…else	day = 2
	IF day = 1 THEN
	  print("Monday")
	ELSE
	  print("Not Monday")
	ENDIF
		day = 2
		if day = 1:
		  print("Monday")
		else:
		  print("Not Monday")
			Use **if…else** when there are exactly 2 possibilities.

if…elif…else	day = 2
	
	IF day = 1 THEN
	  print("Monday")
	ELSE IF day = 2 THEN
	  print("Tuesday")
	ELSE IF day = 3 THEN
	  print("Wednesday")
	ELSE
	  print("Invalid day")
	ENDIF
		day = 2
		
		if day = 1:
		  print("Monday")
		elif day = 2:
		  print("Tuesday")
		elif day = 3:
		  print("Wednesday")
		else:
		  print("Invalid day")
			Use **if…elif…else** when you need to check multiple conditions.

Switch case	day = 2
	
	SWITCH day
	  CASE 1:
	  OUTPUT "Monday"
	  CASE 2:
	  OUTPUT "Tuesday"
	  CASE 3:
	  OUTPUT "Wednesday"
	  DEFAULT:
	  OUTPUT "Invalid day"
	ENDSWITCH
		Not supported in python	Use **Switch…case** when there are many discrete choices → cleaner and easier to maintain.

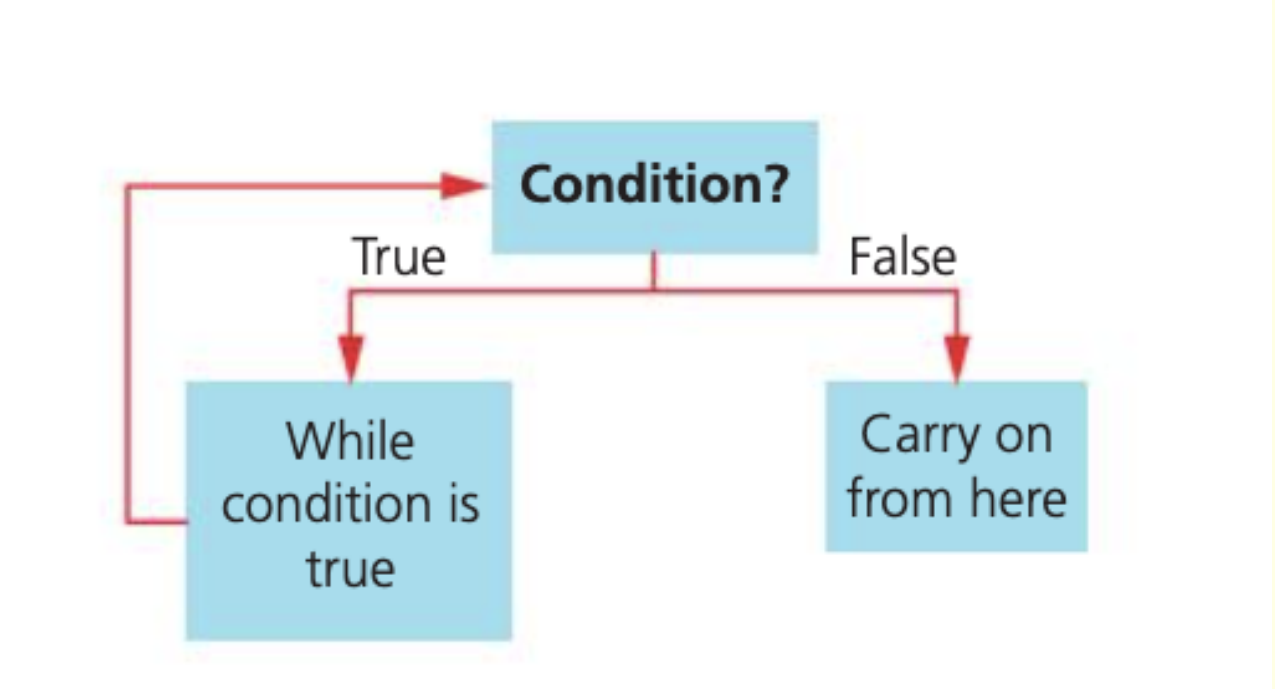
Operators

- The basic operators which can be used in a high-level programming language are as follows:

Operators	Description	Example
> >=	Greater than/ greater than and equal to	if i > 10
< <=	Less than/ less than and equal to	if i < 10
==	Equal to	flag == True
!=	Not equal to	flag != True
and	Both conditions have to be True	if i > 10 and flag == True
or	Either conditions have to be True	if i > 10 or flag == True
not	condition have to be False	if i > 10 not flag == True

Iteration

- Do a set of statements multiple times.
- Iteration is either **'count controlled**' or **'condition controlled**'.
- Count repeats the section n times,
- Condition waits until a condition has been met before stopping iteration



Iteration

Iteration in high level languages can be done in one of 3 ways:

for Loop
Execute a sequence of statements multiple times
while Loop
Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
do while Loop
Like a while statement, except that it tests the condition at the end of the loop body.

Pseudo coding Loops

Pseudo coding	Python
Count Controlled for i=0 to 7 print("Hello") next i Will print hello 8 times (0-7inclusive).	for i in range(8): print("Hello")
While (Condition Controlled) while answer!="computer" answer=input("What is the password?") endwhile	while answer!="computer": answer=input("What is the password?")
Do until (Condition Controlled) do answer=input("What is the password?") until answer=="computer"	Not supported in python

Flags and counters

- **Flag** → flexible stopping condition (e.g., user input, event).
- **Counter** → fixed number of iterations, controlled by incrementing a variable.

Pattern	Description	Example pseudocode	How it Works

While with a Flag	Uses a boolean variable (flag = True/False) to control when the loop should stop. Often used when the stopping condition is based on user input or an event.	# While loop with a flag
		flag = True
		
		while flag
		  # handle spaces & case
		  answer = input("Type 'exit' to stop: ").strip().lower()
		  if answer == 'exit' then
		  flag = False
		  print("Exiting the loop...")
		  else
		  print("You typed:", answer)
		  endif
		End while
		
		print("Loop finished!")
			1. Loop continues as long as flag is True.
			2. User types "exit"
			3. flag becomes False
			4. loop ends.

While with a Counter	Uses a numeric counter to repeat a fixed number of times. Often used instead of a for loop when manual control is needed.	#While loop with a counter
		count = 0
		limit = 5 # makes it easy to change how many times the loop runs
		
		while count < limit
		  print("Count is:", count)
		  count += 1
		End while
		print("Loop finished!")
			1. Loop runs while count < 5.
			2. Each iteration adds 1 to count.
			3. When count reaches 5, condition is False
			4. loop ends.

AND / OR

A tourist attraction has a daily charge for children of £5.00 on a weekday, or £7.50 on a weekend or bank

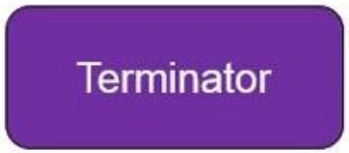
holiday. Adults are charged £8.00 on weekdays and £12.00 on weekends and bank holidays.

Write pseudocode to allow the user to calculate the charge for a visitor.

It is important to use brackets and to get them in the correct place to avoid any confusion over which operator is processed first. In standard Boolean logic the precedence rules make NOT highest, then AND, then OR.

```
day = input("Enter W for weekend, B for bank holiday or D for weekday: ")
visitor = input("Enter A for adult, C for child: ")
if ((day = "W") OR (day = "B")) AND (visitor = "A") then
      charge = 12.0
else if ((day = "W") OR (day = "B")) AND (visitor = "C") then
      charge = 7.5
else if (visitor = "A") then
       charge = 8.0
else
        charge = 5.0
endif
```

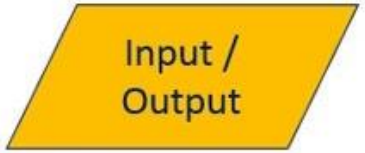
Flow chart Symbols



Terminator
Used to represent the Start and end of a program with the Keywords BEGIN and END.

Decision
Used to split the flowchart sequence into multiple paths in order to represent SELECTION and REPETITION.

Process
An instruction that is to be carried out by the program.



Input / Output
Used to represent data entry by a user or the display of data by the program.

Arrow
Indicates the flow of the algorithm pathways.

Subprogram
References another program within the program.

4. A fast food restaurant offers half-price meals if the customer is a student or has a discount card. The offer is not valid on Saturdays.

A computer system is used to identify whether the customer can have a half-price meal.

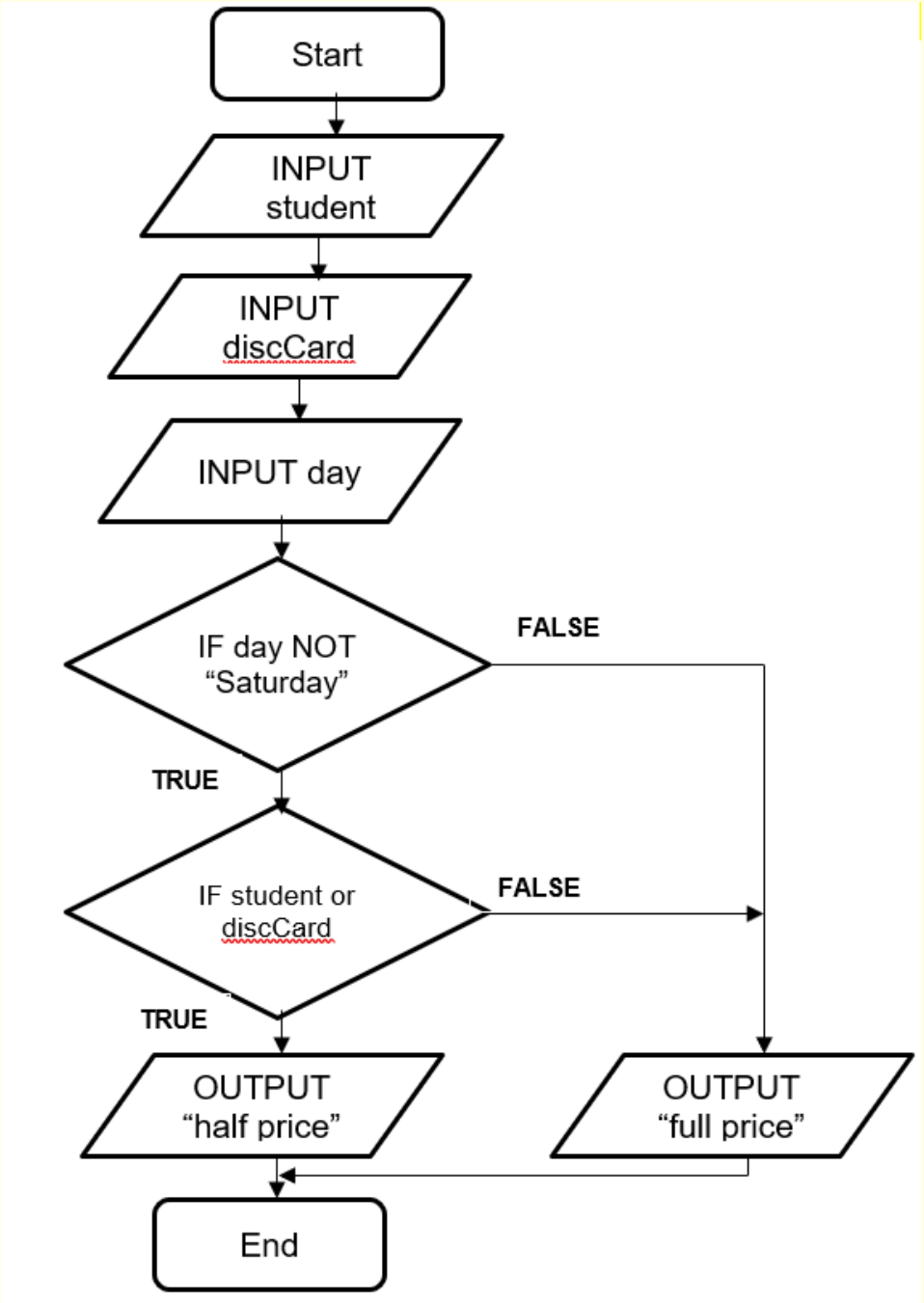
The three inputs to the computer system:

A Is a student
B Has a discount card
C The current day is Saturday

The restaurant needs an algorithm designing to help employees work out if a customer can have a half price meal or not. It should:

- input required data
- decide if the customer is entitled to a discount
- output the result of the calculation.

Design the algorithm using a flowchart. [5]



Learning Aims

- Be familiar with subroutines (functions and procedures), their uses and advantages
- Use subroutines that return values to the calling routine
- Describe the use of parameters to pass data to subroutines by value and by reference
- Contrast the use of local and global variables

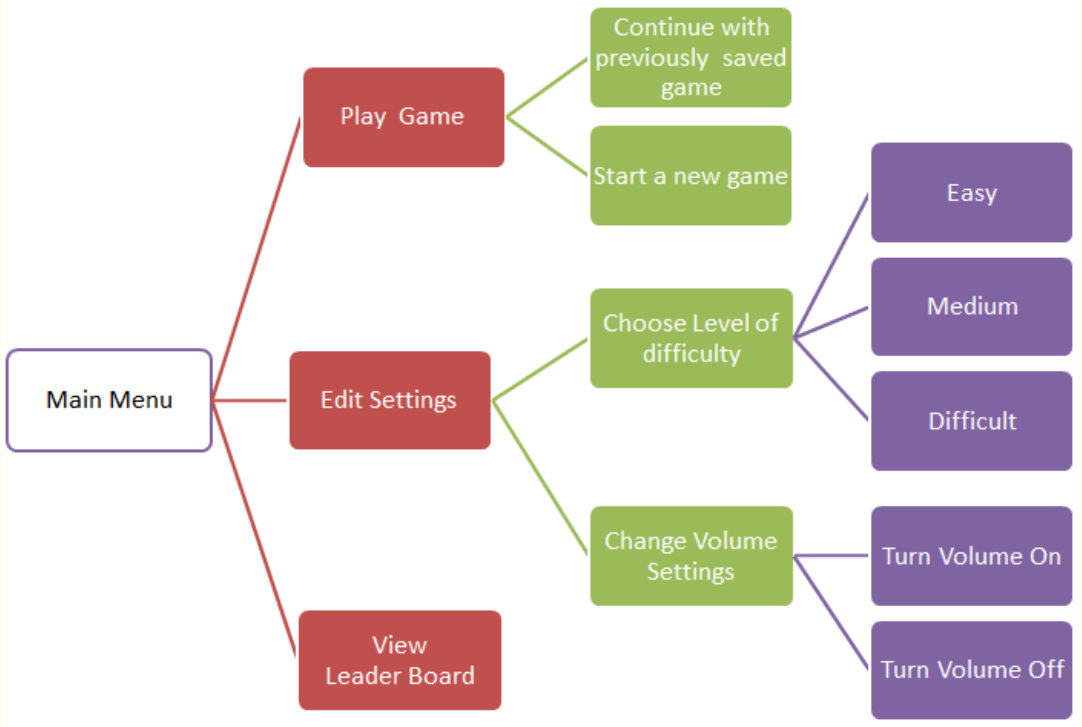
Key terms

Blocks of Code	**Passing** Data
Modularity Function Procedure Subroutine	Argument Parameter

Passing data to subroutine
Calling a subroutine

Why is it good to develop code in a modular way?

- Modularity is used to break a program down into manageable parts that can be self-contained and tested independently.
- It allows a team of programmers to split tasks and focus on specific parts of the program.
- Each module carries out a single, specific task.
- Used to reuse code
- To save you rewriting lots of code again and again you might use a sub routine from an existing library
- It allows faster program development as it allows programmers to re-use previously created code, so we don't have to repeat it.



Structuring code

```
# Program make a simple calculator

# This function adds two numbers
def add(x, y):
    return x + y

# This function subtracts two numbe
def subtract(x, y):
    return x - y

# This function multiplies two numb
def multiply(x, y):
    return x * y

# This function divides two numbers
def divide(x, y):
    return x / y

print("Select operation.")
print("1.Add")
print("2.Subtract")
print("3.Multiply")
print("4.Divide")

# Take input from the user
choice = input("Enter choice(1/2/3/
```

Procedures vs Functions

Procedure

- Performs a set task
- Takes in zero, one or more parameters

Function

- Performs a set task
- Takes in zero, one or more parameters
- **Returns a value**

Procedure Syntax

```
procedure procedure_name(parameters):
    instructions
end procedure
```

Python example

```
def result(name, age, gender):
    print ("My name is" +name)
    print ("My age is" + str(age))
    print("My gender is" + gender)

result("Your name", 13, "gender")
```

Parameters

Arguments

Parameter is variable in the declaration of procedure/function.

Argument is the actual value of this variable that gets passed to function.

Function Syntax

function function_name(parameters):

    instructions

    return variable/ list

end function

The return statement is used to specify the output.

main.py

```
def age_check(age):

  if age < 13:
    check True
  else:
    check = False
  return check

your_age = int(input())
# call function and store returned value in a variable
security_check = age_check(age)
# if security check is True
if security_check:
  print("Access Denied")

else:
  print("Welcome")

```

The returned value will need to be stored in a variable

Advantages of using Subroutines

- **Breaking down** or **decomposing** a complex programming task into **smaller sub-tasks** and **writing** each of these as subroutines, makes the **problem** easier to solve.
- **Subroutines** can be used several times within a program.
- It saves the programmer time as it **reduces the amount of code** that needs to be written or amended by allowing you to reuse code **without** having to write it again.
- If you are working as part of a team you can divide a large program into smaller sections and allow individuals to simultaneously work on those sections.
- It makes the code easier to read if you use sensible subroutine labels as the headings tell the reader what that section of code is doing.
- By **reducing** the amount of **repeating tasks** you also **reduce the risk** of introducing errors in a program.
- **Easy to maintain** as each subroutine can be tested separately.

Flowcharting a subprogram

Figure 1.7.3 The symbol for a subprogram in a flowchart

Flowchart

In a flowchart, a subprogram is represented by the symbol shown in Figure 1.7.3.

Fully documenting a complex solution using flowcharts can be cumbersome, and it's quite easy to get lost. Figure 1.7.4 shows what part of the noughts and crosses game might look like using subprograms.

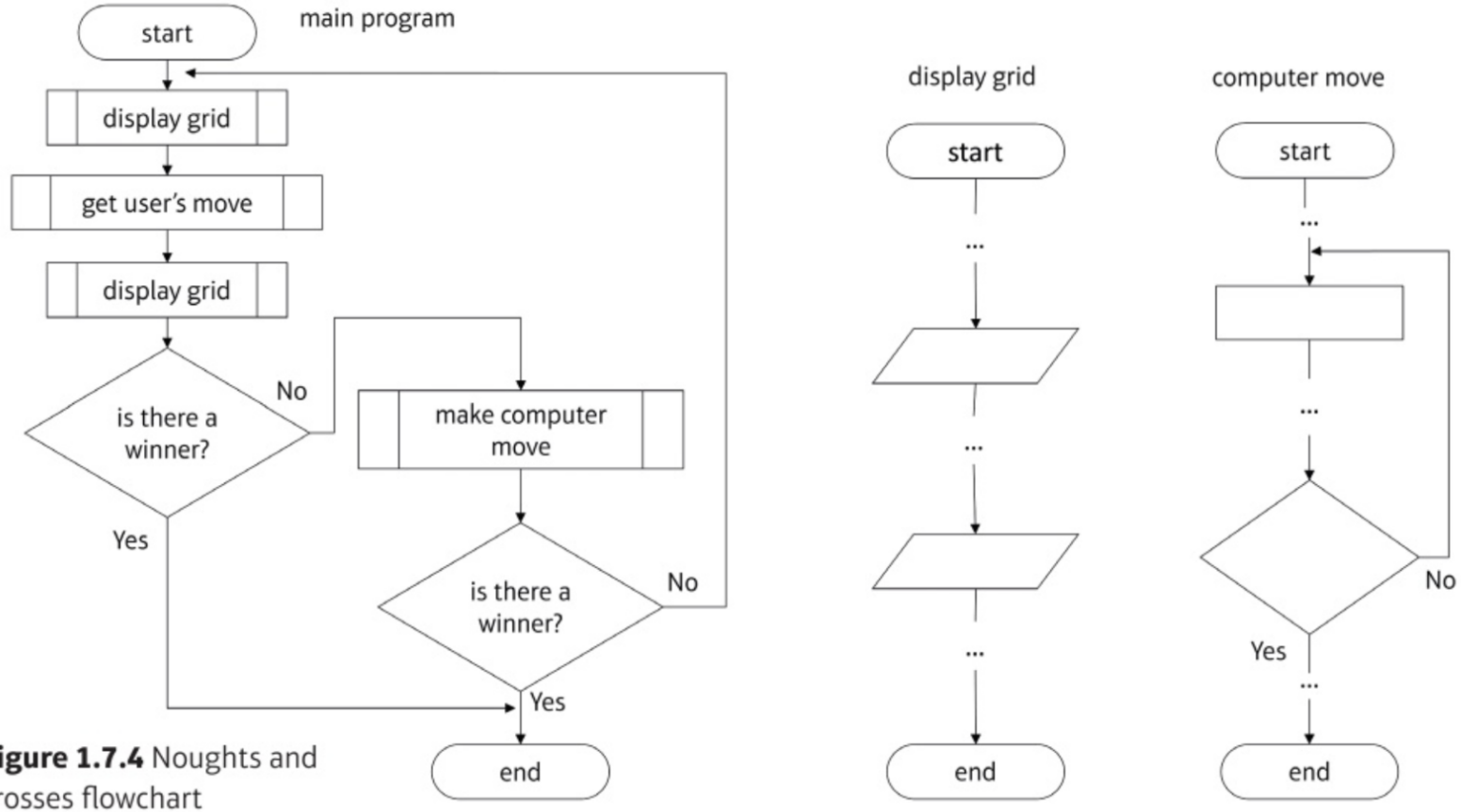
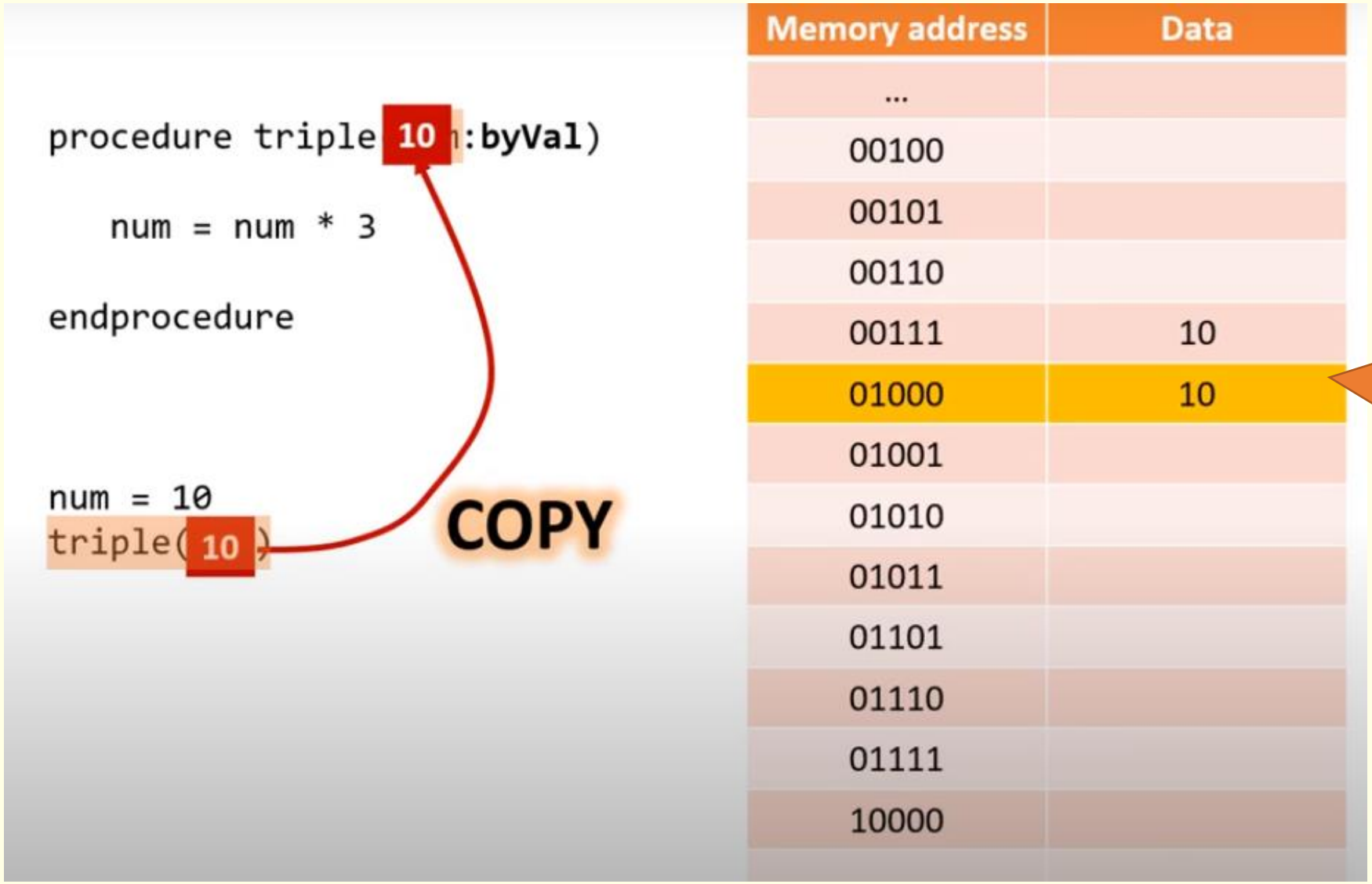


Figure 1.7.4 Noughts and crosses flowchart

Types of Parameters:

- When parameters/arguments are passed to a procedure/function they can be passed in two ways:
- Passing by **value -** a **copy** of the original data is passed to the function and any changes made are lost as soon as the function is no longer in use.
- Passing by **reference -** the function receives a pointer to the actual **memory address** where the data is stored. This means that the function works directly with the original data and if it changes, it stays changed.

Example: Passing by Value



Example: Passing by Reference

```
procedure triple(num:byRef)

    num = num * 3

endprocedure



num = 10
triple(num)
```

Memory address	Data
...	
00100	
00101	
00110	
00111	30
01000	
01001	
01010	00111
01011	
01101	
01110	
01111	
10000	
...	

Overwriting the original data

Exam Top Tips

From the OCR specification:

"Unless stated, values passed to subroutines can be assumed to be passed by value. If this is relevant to the question, byVal and byRef will be used.

In the case below, x is passed by value and y is passed by reference."

```
procedure foobar(x:byVal, y:byRef)
    ...
    ...
endprocedure
```

Variable Scope – When declaring a variable or constant the programmer needs to aware of its scope – global or local

	Local variable	Global variable
Accessibility	Single subroutine only	Entire program
Created	Inside a subroutine	Outside of a subroutine Typically at the start of a program
Destroyed	When the subroutine exits	When the program ends

- A **local** variable is typically:
  - Declared inside a subroutine.
  - Only accessible by that subroutine.
  - Created when the subroutine is called.
  - Destroyed when the subroutine ends.
- A **global** variable is typically:
  - Declared at the top of a program, outside of any subroutine.
  - Accessible throughout the program.
  - Created when the program starts.
  - Destroyed when the program ends.

Global & Local Variables in Python

```
#local variable
def foo():
        y = "local"
        print(y)

foo()
```

```
#assigning and accessing a global variable
x = "global“
print(x)
def foo():

        global x
        y = "local"
        x = x * 2
        print(x)
        print(y)
foo()
print(x)
print(y)
```

Good Practice – Avoid global variables

- Good practice is to avoid the use of global variables in favour of local variables.
- Excessive, unnecessary use of global variables can make programs hard to test, debug and maintain.
- Global variables are more likely to be accidental changed, especially in a larger program when hundreds or thousands of variables are in use.
- A global variable places greater demand on system resources - it needs to be retained in memory (RAM) throughout the entire execution of a program whereas the values of local variables will be held in RAM only during the execution of this sub-program.
- Local variables should be used where possible for this reason and also as they use less memory as they are created and destroyed within their subroutine.

A better approach is **Parameter Passing:**

```
maximum = 100.00

def postage(cost, max):

   print("Free postage if you spend more than "
+str(maximum))
   if cost < max:
        cost = cost + 3.50
  return ( "£"+ " " + str(cost))


totalAmount = float(input("Enter Total"))
print(postage (totalAmount, maximum))
```

Worked Examples

Describe the use of local variables.

- Defined within one module… For module you can use procedure / function / sub routine / block of code
- … accessible only in that module / Any mention of scope
- Can be used as parameters
- Data is lost at end of module
- Same variable name can be used in other modules without overwriting values/causing errors

State two features of global variables that distinguish them from local variables.

- Defined at start of program
- Exists throughout program / in all modules
- Allows data to be shared by modules

Learning Objectives

- Be familiar with the concept of a data structure
- Be familiar with arrays of up to 3 dimensions,
- Difference between an array and list
- Tuples
- Records

Arrays, Lists and Tuples

0	1	2	3	4	5
blue	green	purple	cyan	magenta	violet

Allows multiple items of data to be stored under one identifier

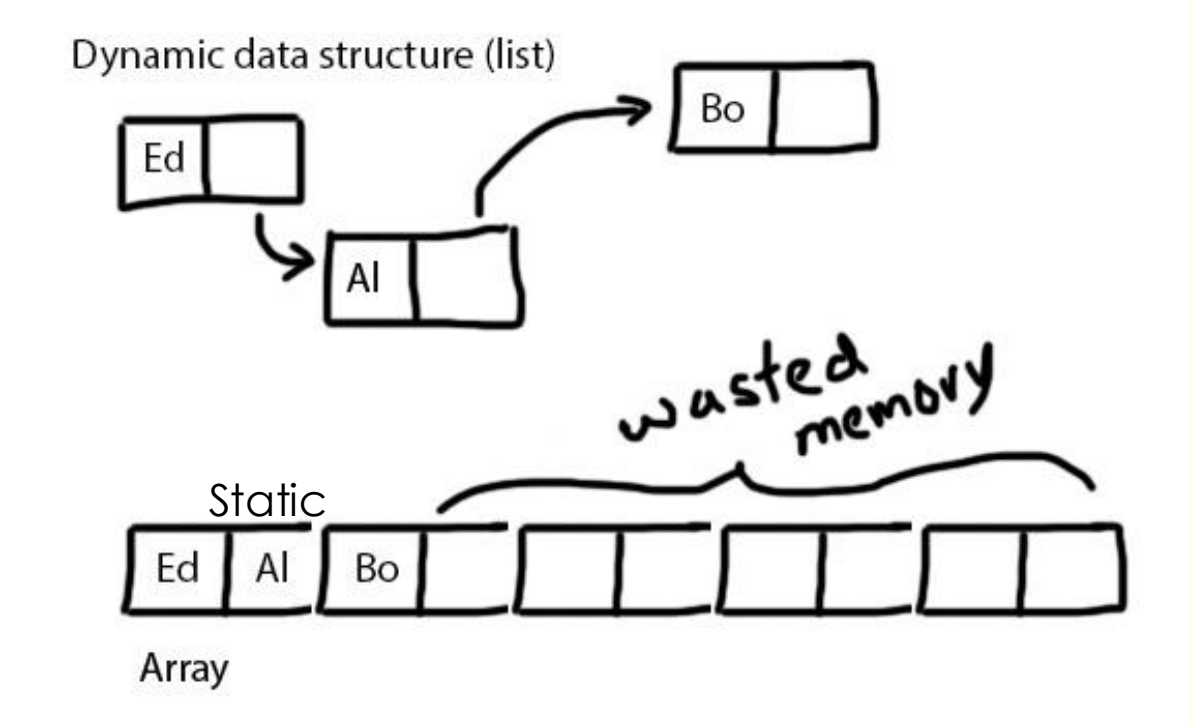
Can store a table structure

Reduces need for multiple variables

Array … must be of the **same data type** and fixed size (**static**)

List … A **list** is a sequence of values, can be **different data types**. It is **mutable** – so you can add, remove and change items in it. (**dynamic**)

Tuple…is **immutable**, so once it's set up, it can't be changed.



1-dimensional arrays

An array is defined as a finite, ordered set of elements of the same type, such as integer, real or char.

Finite means that there is a specific number of elements in the array.

Ordered implies that there is a first, second, third etc. element of the array.
For example, (assuming the first element of the array is myArray[0]:

```
myArray = [51, 72, 35, 37, 0, 3]
x = myArray[2] #assigns 35 to x
```

Worked Example

Every year the RSPB organises a Big Garden Birdwatch to involve the public in counting the number of birds of different types that they see in their gardens on a particular weekend. During 30-31 January 2016, more than 8 million birds were counted and reported.

The scientists add all the sightings together, and once the data has been analysed, they can discover trends and understand how different birds and other wildlife are faring.

An array of strings could be used to hold the names of the birds, and an array of integers to hold the results as they come in. As a simple example we will hold the names of 8 birds in an array:

```
birdName = ["robin", "blackbird", "pigeon", "magpie", "bluetit","thrush", "wren", "starling"]
```

We can reference each element of the array using an index. For example:

```
birdName[2] = "pigeon" #the index here is 2
```

Most languages have a function which will return the length of an array, so that

```
numSpecies = len(birdName)
```

will assign 8 to numSpecies.

Worked Example

- To find at which position of the array a particular bird is, we could use the following pseudocode algorithm:

```
bird = input("Enter bird name: ")
birdFound = False
numSpecies = len(birdName)
for count = 0 to numSpecies - 1
   if bird == birdName[count] then
      birdIndex = count
      birdFound = True
   endif
next count
if birdFound == False then
   print("Bird species not in array")
else
   print("Bird found at",birdIndex)
endif
```

We need a second array of integers to accumulate the totals of each bird species observed.
We can initialise each element to zero.

birdCount = [0,0,0,0,0,0,0,0]

To add 5 to the blackbird count (the second element in the list) we can write a statement

birdCount[1] = birdCount[1] + 5

Working with Files

- Reading, Writing, and Handling Data Persistently

Why Use Files?

- Data in variables is **temporary** (lost when the program ends).
- Files allow data to be **stored permanently** on secondary storage.
- Files are essential for:
  - Saving user settings
  - Logging information
  - Storing large datasets

RAM = short-term memory

Files = long-term memory.

Types of Files

- **Text files (.txt):** store human-readable data (strings, numbers, etc.).
- **Binary files:** store data in computer-readable form (images, executables, etc.).
- CSV files (.csv): **Comma-Separated Values**
- Json files(.json): Stores data in **key-value pairs** (similar to Python dictionaries).

- **OCR focus** → mainly text file handling in Python.

Basic File Operations

- **Open** a file
- **Read** from a file
- **Write** to a file
- **Close** a file

Python uses the open() function with different modes:

- "r" → read
- "w" → write (overwrites)
- "a" → append

Reading from a File

```
file = open("data.txt", "r")
for line in file:
    print(line.strip())
file.close()
```

strip() removes the newline character \n.

Writing to a File

```
file = open("output.txt", "w")
file.write("Hello, world!\n")
file.write("This is stored permanently.")
file.close()
```

"w" overwrites the file.

Appending to a File

```
file = open("output.txt", "a")
file.write("\nAdding another line.")
file.close()
```

New data is added to the end of the file.

Using with (Best Practice)

Automatic closing of files.

```
with open("data.txt", "r") as file:
    for line in file:
        print(line)
```

Cleaner, safer, avoids forgetting close().

Common OCR Exam Points

- Understanding **modes** (“r”, “w”, “a”).
- Using iteration (for line in file).
- Knowing why files are needed (persistent storage).
- Writing pseudocode vs Python.

CSV Files

- **CSV = Comma-Separated Values**
- Stores data in a **table-like structure** (rows & columns).
- Each line = a record (row).
- Each value separated by commas (or sometimes tabs).

students.csv

```
Name,Score
Alice,85
Bob,73
Charlie,91
```

Working with CSV in Python

Output is a list per row: ['Alice', '85'].

```
import csv

with open("students.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Writing to CSV:

```
import csv

with open("students.csv", "a",
newline="") as file:
    writer = csv.writer(file)
    writer.writerow(["David", 77])
```

Json files

- **JSON = JavaScript Object Notation**
- Stores data in **key-value pairs** (similar to Python dictionaries).
- Human-readable and widely used in web APIs.

student.json

```
{
  "name": "Alice",
  "score": 85,
  "passed": true
}
```

JSON is stored as **dictionaries** in Python.

What is a Dictionary?

- A **data structure** in Python.
- Stores data as **key–value pairs**.
- Unlike lists (which are ordered by index), dictionaries are accessed by **keys**.

💡 Think of a real dictionary: you look up a *word* (key) to find its *definition* (value).

Creating a dictionary

```
# Empty dictionary
student = {}

# Dictionary with data
student = {
    "name": "Alice",
    "age": 17,
    "score": 92
}
```

Here:
"name", "age", "score" = keys
"Alice", 17, 92 = values

Accessing Values

Use the **key** in square brackets, not an index.

```
print(student["name"])   # Alice
print(student["score"])  # 92
```

Updating and Adding Data

Now student has a new key "grade".

```
student["age"] = 18          # update value
student["grade"] = "A"       # add new key-value pair
```

Iterating Through a Dictionary

```
for key, value in student.items():
    print(key, "→", value)
```

Working with JSON in Python

```
import json

# Reading
with open("student.json", "r") as file:
    data = json.load(file)
print(data["name"], data["score"])

# Writing
student = {"name": "Bob", "score": 73, "passed": False}
with open("student.json", "w") as file:
    json.dump(student, file, indent=4)
```

Summary

- Files allow data to be **stored beyond runtime**.
- Two main operations: **read** and **write**.
- Python file modes: "r", "w", "a".
- Best practice: use with open(...) as ....

Worked Example

The following algorithm enables a member of the Birdwatch team to enter results as they come in from members of the public.

```
birdName = ["robin", "blackbird", "pigeon", "magpie", "bluetit",
                                    "thrush", "wren", "starling"]
birdCount = [0,0,0,0,0,0,0,0]
bird = input("Please input name of bird (x to end): ")
while bird != "x"
   birdFound = False
   for count = 0 to 7
      if bird == birdName[count] then
         birdFound = True
         birdsObserved = input("number observed: ")
         birdCount[count] = birdCount[count] + birdsObserved
      endif
   next count
   if birdFound == False then
      print("Bird species not in array")
   endif
   bird = input("Please input name of bird (x to end): ")
endwhile
#now print out the totals for each bird
for count = 0 to 7
   print(birdName[count], birdCount[count])
next count
```

Useful List Functions

- **len(my_list)** length of list
- **my_list.append(item)** adds an item to the end of the list
- **my_list.extend(list)** adds a list to the end of a list
- **my_list.insert(index, item)** adds an item to the list at the specified index
- **my_list.remove(item)** removes the first occurrence of the item from the list
- **del my_list[index]** delete from a specific index
- **my_list.pop([index])** removes and returns an item from the list (if index is omitted it removes the last item)
- **my_list.index(item)** returns a number indicating the position of the item in the list
- **my_list.count(item)** counts how many times the item appears in the list
- **my_list.sort()** sorts list items into order
- **my_list.reverse()** reverses the list

2-dimensional arrays

An array can have two or more dimensions. A two-dimensional array can be visualised as a table, rather like a spreadsheet.

Imagine a 2-dimensional array called numbers, with 3 rows and 4 columns.

Elements in the array can be referred to by their row and column number, so that numbers[1,3] = 8 in the example below.

	Column 0	Column 1	Column 2	Column 3
Row 0	1	2	3	4
Row 1	5	6	7	8
Row 2	9	10	11	12

What is the value of numbers[2,1]?

Worked Example

Write a pseudocode algorithm for a module which prints out the quarterly sales figures (given in integers) for each of 3 sales staff named Anna, Bob and Carol, together with their total annual sales. Assume that the sales figures are already in the 2-dimensional array quarterSales. The staff names are held in a 1-dimensional array staff.

```
staff = ["Anna","Bob","Carol"]
quarterSales = [[100,110,120,110],
                [350,355,360,360],
                [200,210,220,220]]
for s = 0 to 2
   annualSales = 0
   #output staff name
   (insert statement here)

   for q = 0 to 3
      print("Quarter ", q, quarterSales[s,q])
      annualSales = annualSales + quarterSales[s,q]
   next q
   print("Annual sales: ", annualSales)
next s
```

What statement needs to be inserted after the comment #output staff name in order to output the staff name?

print (staff[s])

Arrays of three dimensions

- Arrays may have more than two dimensions. An n-dimensional array is a set of elements of the same type, indexed by n integers.
- In a 3-dimensional array x, a particular element may be referred to as x[4,5,2], for example.
- The first element would be referred to as x[0,0,0].

Tuples

- A tuple is an ordered set of values, which could be elements of any type such as strings, integers or real numbers, or even graphic images, sound files or arrays.
- Unlike arrays, the elements do not all have to be of the same type. However, a tuple, like a string, is immutable, which means that its elements cannot be changed, and you cannot dynamically add elements to or delete elements from a tuple.

In Python a tuple is written in parentheses, for example:

```
pupil = ("John", 78, "a")
```

You can refer to individual elements of a tuple, for example:

```
name = pupil[0]
```

but the following statement is invalid:

```
pupil[0] = "Mary"
```

Example: Write a pseudocode algorithm to perform the following task:

Create a function that accepts a 1D array of integers and returns the index of the largest number in the array.

Example Solution for the Algorithm

```
FUNCTION findLargestIndex(arr)
    maxIndex = 0
    maxValue = arr[0]
    FOR i = 1 TO len(arr) - 1
        IF arr[i] > maxValue then
            maxValue = arr[i]
    return maxValue
End function
```

A 2-dimensional (2D) array, data, holds numeric data that Karl has entered. The declaration for the array is:
array data[16,11]
The array data, has 16 'rows' and 11 'columns'.

	0	1	2	3	...	10
0	1	5	7	12	...	36
1	3	4	15	16	...	48
2	0	0	1	3	...	10
3	12	16	18	23	...	100
...	...	...	...	...	...	...
15	6	10	15	25	...	96

The data in each 'row' is in ascending numerical order.

Karl needs to analyse the data.

Karl needs to find the mean average of each 'column' of the array. The mean is calculated by adding together the numbers in the column, and dividing by the quantity of numbers in the column.

For example, in Fig. 1.3 the first 'column' mean would be:
(1+3+0+12)/4 = 4

1	5	7	12
3	4	15	16
0	0	1	3
12	16	18	23

Write an algorithm to output the mean value of each 'column' in the array data. [5]

•	**Looping through each column [1]**
•	Looping through each row [1]
•	...Adding to a total [1]
•	...Calculating average correctly [1]
•	Outputting average [1]

```
for column in range(11)
    total = 0
    for row in range(16)
        total = total + data[row,col]
    print(total/16)
```

Learning Aims

- Be able to write programs that manipulate strings including length, position, substrings and case conversion.
- How to format a string.
- Design algorithms and write code to validate input
- Use a range of validation checks in programs
- Use built-in subprograms to manipulate variables of data type string

Concatenate	When two or more strings are joined together.
String	A value that is text. This can include numbers but they will be read as text.
String handling	Performing operations on string.
Element	A character in a string, or an item in a sequence.
ASCII	Acronym for American Standard Code for Information Interchange. It is used to represent characters with a numerical value.
Substring	Part of a string.

String Manipulation

A string is anything enclosed in quote marks. “”

Programming languages have built-in functions to manipulate strings.

The functions below are given in a Python format.

Each programming language will have its own particular functions and syntax.

Useful stings functions

To get the length of a string:

len(stringname)

To count the specific number of characters in a a string:

string.count(“letter”)

Position

Individual letters are obtained from a string by using their index.

string[index]

0	1	2	3	4
H	e	l	l	o

For example:

```
message = "Hello"
print(message[1])
```

This will output the letter ‘e’.

Python string indexes start at position 0, not 1.

This means that stringname[0] contains the first character of a string called stringname.

Substrings

A substring is part of a string. In Python, extracting a substring is carried out using string slicing.

To find a substring in Python use the syntax:

```
string[start:end]
```

For example:

```
message = "Hello everyone"
substring = message[4:8]
print(substring)
```

The output will be "o ev".

This is because position 4 in the string contains "o" and positions 5, 6 and 7 contain " ev".

Remember space is included as a position in the string.

End and start position

message = "Hello everyone"

The end position isn't included in Python.

To get the first characters from a string use:

```
string[:end]
```

For example, message[:8] will result in "Hello ev".

To get the last characters from a string use:

```
string[start:]
```

For example, message[start:]

message[11:] will result in the substring "one".

Substring position

It is possible to find the position of a substring by using string.find(substring)

The position will be returned. If the substring isn't in the string then -1 will be returned.

For example:

```
message = "Hello everyone"
position = message.find("very")
print(position)
```

This will return the position 7 as the 'v' in "very" is in index 7 in the string.

Summary

In the statements below, the result of each operation is shown in a comment on the right.

```
myName = "John Robinson”
firstname = myName[0:4]                          # "John"
surname = myName[5:]                             # "Robinson"
numChars = len(myName)                           # 13
lastTwoChars = myName[len(myName)-3:]            # "on"
lastTwoChars = myName[-2:]                       # "on"
positionOfRob = myName.find("Rob")               # 5
```

Case conversion

To convert cases:

```
string.upper()
string.lower()
```

For example:

```
message = "Hello everyone"
print(message.lower())
print(message.upper())
```

This will return:

```
hello everyone
HELLO EVERYONE
```

ASCII characters

Every character is represented in binary, for example using the ASCII representation. ASCII, “A” is

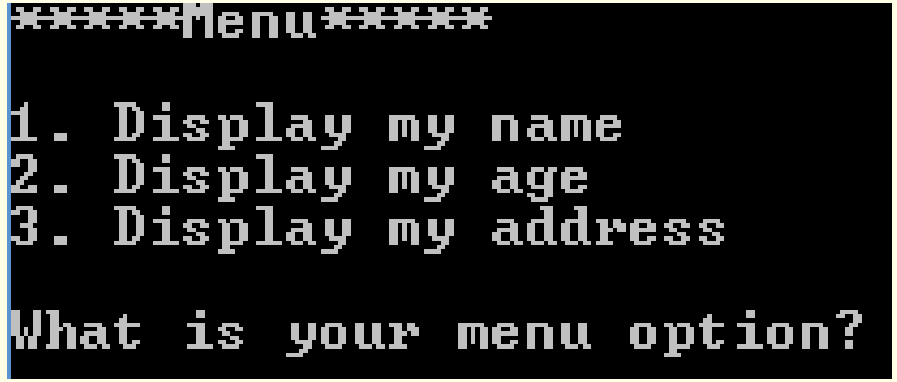
represented by the same binary code as the decimal number 65, “B” is 66, “C” is 67,… “Z” is 90.

The functions ord and chr convert characters between ASCII and decimal.

```
num = ord("A") # makes num = 65
letter = chr(66) # makes letter = "B"
```

Menus

- The user input can be validated using a range check to ensure that the option selected is allowed.




```
1  # -----------------------------------------------------------
2  # Global variables
3  # -----------------------------------------------------------
4  validChoice = False            # Assume everything is invalid
5  userChoice = 0                 # Set to an invalid value
6
7  # -----------------------------------------------------------
8  # Subprograms
9  # -----------------------------------------------------------
10 def showMenu():
11     print ("Option 1: Find the highest value")
12     print ("Option 2: Find the lowest value")
13     print ("Option 3: Calculate the average")
14
15 # -----------------------------------------------------------
16 # Main program
17 # -----------------------------------------------------------
18
19 while (not validChoice):
20     showMenu()
21     userChoice = int (input ("Enter an option: "))
22     if (userChoice >= 1) and (userChoice <= 3):
23         validChoice = True
24     else:
25         print ("Invalid option, try again ")
26
27 print ("You entered option " + str(userChoice))
```

Validating input data

Input validation is when a system will check that the input meets certain criteria, to ensure that the data is in an acceptable form.

- **Range check**: a number or date is within a sensible/allowed range
- **Type check**: data is the right type such as an integer, a letter or text
- **Length check**: text entered is not too long or too short – for example, a password is greater than 8 characters, a product description is no longer than 25 characters
- **Presence check**: checks that some data has been entered, i.e. that the field has not been left blank
- **Pattern check**: checks that the pattern of, for example, a postcode or email address is appropriate

Other useful string functions

The following string functions are also available in Python. You should be aware of how these work as they are included in the Edexcel PLS

String function	Description	Example

`string.isalpha()`	Checks if the string contains alphabetic letters. If it does then it returns True.	`message = "Southampton"`
		`print(message.isalpha())`
		`Output: True`

`string.isalnum()`	Checks if the string contains alphanumeric letters and numbers A-Z and 0-9. Returns True if that's all it contains.	`message = "Southampton23"`
		`print(message.isalnum())`
		`Output: True`

`string.isdigit()`	Returns true if the string contains just digits 0-9.	`message = "498"`
		`print(message.isdigit())`
		`Output: True`

`string.replace(string1, string2)`	Replaces all occurrences of string1 with string2.	`message = "Southampton"`
		`print(message.replace("South", "North"))`
		`Output: Northampton`

`string.split(character)`	Splits a string on each occurrence of the character. Each substring is stored in a list.	`message = "Lemon,Apple,Peach"`
		`fruitList = message.split(",")`
		`print(fruitList)`
		`Output: ["Lemon", "Apple", "Peach"]`

`string.strip(character)`	Remove all occurrences of the character from the string.	`message = "...Beach.."`
		`print(message.strip("."))`
		`Output: Beach`

`string.isupper()`	If all the characters in the string are in uppercase then it returns True.	`message = "HELLO"`
		`print(message.isupper())`
		`Output: True`

`string.islower()`	If all the characters in the string are in lowercase then it returns False.	`message = "hello"`
		`print(message.islower())`
		`Output: True`

`string.index(substring)`	This works in the same way as `string.find(substring)` which was discussed earlier. The difference is that it raises an exception (rather than returning -1) if the substring isn't found. Use this only if you are using exceptions.	

Validation Routines and While Loops

When we attempt to introduce input validations to our program, we will often make use of the while loop construct.

Here is an example of a simple validation routine that ensures that the program only accepts the inputs YES and NO, rejecting all other inputs:

```
Do you wish to carry on? Type YES or NO
yessss
Wrong word entered, please type YES or NO
yes
Wrong word entered, please type YES or NO
YES
You want to carry on!
```

Validation Routines and While Loops

A while loop to repeatedly ask for an input until the user enters YES or NO. This is the validation routine.

An input statement to record the user's answer to the question

If-Else statements to act on the users input

```
answer = input("Do you wish to carry on? Type YES or NO")


while (answer != "YES") and (answer != "NO"):
    answer = input("Wrong word entered, please type YES or NO")


if answer == "YES":
    print("You want to carry on!")
else:
    print("You don't want to carry on!")
```

Range check

- The following algorithm asks the user to enter an integer between 17 and 30, and validates it by performing a range check. The number is then multiplied by 3 and the result printed out.

```
num = int(input("Enter number between 17 and 30:"))
while num < 17 or num > 30:
        num = int(input("Invalid number - please re-enter: "))
        print(num * 3)
```

Type check

The following program asks the user to enter a number. It performs a type check. If the user has entered text rather than a number, they will be asked to enter the number again.

```
num = input("Please enter an integer: ")
while not num.isdigit():
    num = input("You must enter an integer, try again: ")
    num = int(num)
print(num)
```

Other useful functions for checking the type of data entered include:

- isalpha() Returns True if all the characters are alphabetic (A-Z)
- isalnum() Returns True if all the characters are alphabetic (A-Z) or digits (0-9).

Length check

The following program asks the user to enter their name, and performs a length check.

The name must be between 2 and 20 characters.

```
name = input("Please enter name: ")
while len(name) < 2 or len(name) > 20:
        name = input("Must be between 2 and 20 characters – please re-enter: ")
print(name)
```

Presence check

- This simply ensures that a value has been presented to the program, preventing the from leaving an input blank.
- This algoirthm asks the user to input their name and uses a presence check to ensure the they have entered a value.
- This example illustrates how to do a presence check on user input from the keyboard.

```
1  # ------------------------------------------------------------
2  # Global variables
3  # ------------------------------------------------------------
4  userName = ""               # Initialise to empty string
5
6  # ------------------------------------------------------------
7  # Main program
8  # ------------------------------------------------------------
9  userName = input ("Enter your name: ")
10 while (userName == ""):
11     userName = input ("Enter your name: ")
12 print ("Your name is: ", userName)
```

Pattern check

A pattern check (also known as a format check) allows the checking of an email address, for example.

Look at the following program which makes use of the split function to check that an @ symbol is in the address.

```
validEmail = False
email = ""
while not validEmail:
email = input("Enter email address: ")
emailParts = email.split("@")
if len(emailParts) == 2:
    validEmail = True
else:
     print("That isn't a valid email address")
```

The program splits the email address entered with each occurrence of the @ symbol. This should result in a list with two items, the left hand side and right hand side of the email address.

One problem with this program is that the left hand side and right hand side of the @ symbol could be empty, and this would still result in the program choosing it as a valid email.
Adapt the program so that it checks to see there are characters before and after the @ symbol.

```
validEmail = False
email = ""
while not validEmail:
    email = input("Enter email address: ")
    emailParts = email.split("@")
    if (len(emailParts) == 2 and len(emailParts[0]) > 0
                             and len(emailParts[1]) > 0):
        validEmail = True
    else:
        print("That isn't a valid email address")
```

Validating strings

Function	Description
`len(<string>)`	Returns the length of <string>
`<str>.upper()`	Returns the original string in upper case
`<str>.lower()`	Returns the original string in lower case
`<str>.isalnum()`	Returns True, if all characters are alphabetic (a-z, A-Z,) and digits (0-9)
`<str>.isalpha()`	Returns True, if all characters are alphabetic (a-z, A-Z)
`<str>.isdigit()`	Returns True, if all characters are digits (0-9)

Patterns with only letters

- A program may be written to accept a person's first name. The programmer makes a set of rules for validating first name. The rules are:
- must have at least 1 letter and no more than 20 letters
- must only have letters, i.e. no digits or symbols
- This example shows how to implement the two rules described above.

Using isalpha() to check is all the input only have letters

```
# -----------------------------------------------------------
# Constants
# -----------------------------------------------------------
MIN_NAME = 1                # Constants for name length
MAX_NAME = 20

# -----------------------------------------------------------
# Global variables
# -----------------------------------------------------------
firstName = ""              # User types
valid = False               # Assume everything is invalid
length = 0                  # Invalid length
newName = ""                # Invalid capitalised name

# -----------------------------------------------------------
# Main program
# -----------------------------------------------------------
while (not valid):
    firstName = input ("Enter your first name: ")
    length = len(firstName)
    if (length >= MIN_NAME) and (length <= MAX_NAME):
        if (firstName.isalpha()):
            newName = firstName
            print ("Your entry is valid.\nYour name is: ", newName)
            valid = True
        else:
            print ("Your entry is not valid")
    else:
        print ("Your entry is not the right size")
```

Patterns with both letters and digits

- Identification numbers, stock identification and postcodes are all items that may be combination of both letters and digits. These can be validated by the built-in subprograms.

- This example validates the strings representing form names. Examples of valid form names are "7AXB", "8PBD","9ARL". From this we create these rules:

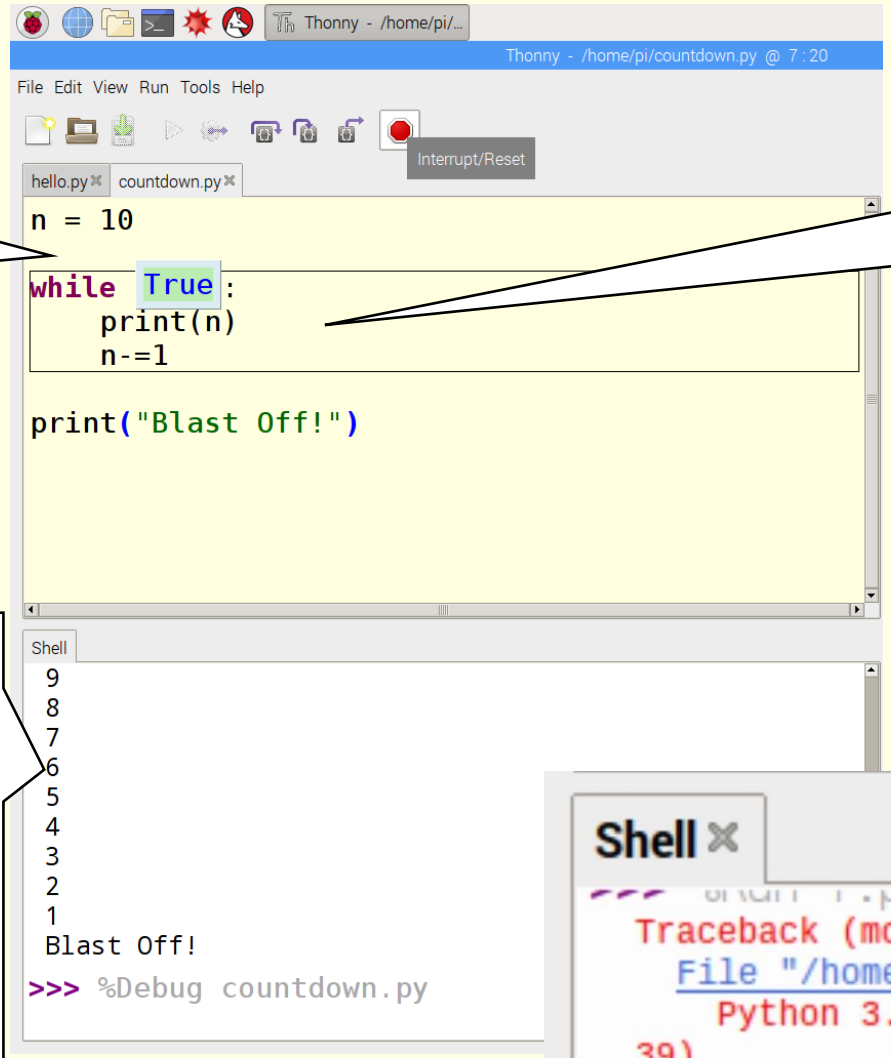
  - must start with the digits 7,8,9
  - must be followed by three letters, but we can adjust the case
  - must be exactly four characters long

```
# ---------------------------------------------------------
# Constants
# ---------------------------------------------------------
FORM_LEN = 4            # Constants for name length

# ---------------------------------------------------------
# Global variables
# ---------------------------------------------------------
formName = ""               # User types
valid = False               # Assume everything is invalid

# ---------------------------------------------------------
# Main program
# ---------------------------------------------------------
while (not valid):
    formName = input ("Enter a form name: ")
    if (len(formName) == FORM_LEN):        # Length is good
        if (formName.isalnum()):   # Letters and digits only
            # Check first character is 7, 8, or 9
            if ((formName[0] == "7") or
                (formName[0] == "8") or
                (formName[0] == "9")):
                # Good first char, check remaining three
                if (formName[1].isalpha() and
                    formName[2].isalpha() and
                    formName[3].isalpha()):
                    valid = True
                    print ("Your entry is valid.", formName)
                else:
                    print ("Last three characters must be letters")
            else:
                print ("First character must be 7, 8, or 9")
        else:
            print ("Your entry is not valid")
    else:
        print ("Form name is not correct length")
```

Learning Objectives

- Be familiar with the use of an IDE to develop and debug a program
- Understand the purpose of testing and devise a test plan
- Discuss the purpose of programming standards

Syntax Highlighting **Editor** **Run-time environment** **Debugger**

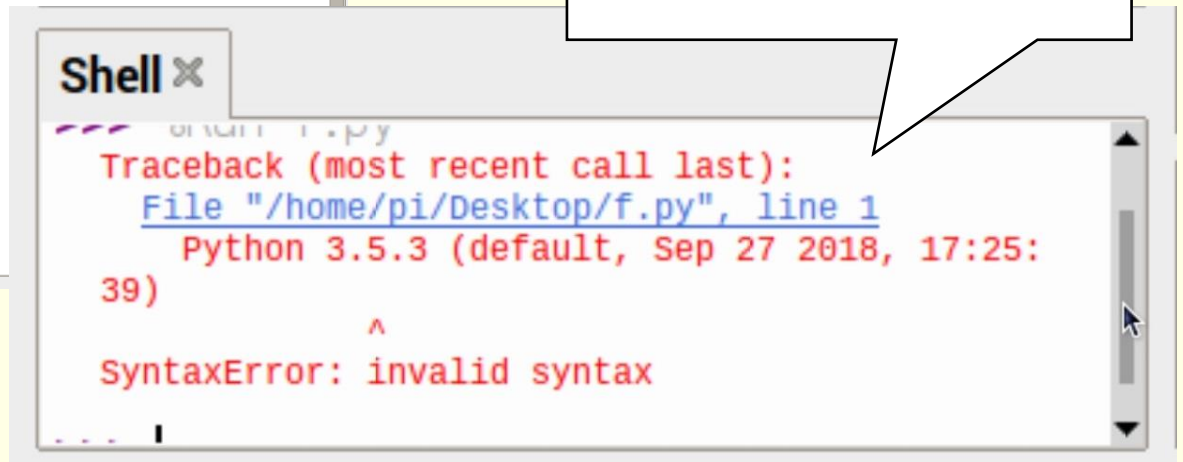


..... to enable program code to be entered / edited

.....to help make the code more readable and easier to maintain

..... to enable program to be run / to check for runtime errors / test the program

.....to display information about errors / location of errors / suggest solutions



The IDE

- **Software used to develop/debug a program.**
- At the very least an IDE should include:
  - An editor for writing **Source Code**
  - Facilities for automating the **Build**
  - A **Debugger**
  - **Breakpoints and steppers**
  - Features to help with code writing, such as **code completion.**

Key features

- **Code editors:** A text area used for programmers to enter code directly into the IDE. Often comes with additional features such as syntax highlighting, autocomplete code and self-indentation support.

- **Error diagnostics:** A feature which reports errors (mainly syntax) in the code and potential problems along with where they are found and often suggestions on how to fix.

- **Run-time environments:** Software to support the execution and running of programs. It allows the developer to run their code during development in order to check for logical errors.

- **Translators:** A program which converts high-level or assembly-level code into executable machine code by either compiling it or interpreting it.

- **Auto-documentation:** A feature which tracks variables declared by the programmer, modules and other special programmer comments in order to produce self-documentation to help program maintenance, debugging and support.

There are many other useful features depending on the IDE you are using…

- Underlines syntax errors dynamically
- Watch window
- Break points
- Error message list
- Step-mode
- Traces
- Crash-dump/post-mortem routine
- Stack contents
- Cross-referencers

Explained…

- **Underlines syntax errors dynamically** - can be corrected before running // saves times
- **Watch window -** View how variables change during running of the program
- **Break points -** Stop the program at set points to check the values of variables
- **Error message list -** Tells you where errors are and suggests corrections
- **Step-mode** (stepper) - Executes program one statement at a time to watch variable values and program pathways
- **Program Tracing -** Printouts of variable values for each statement execution within a program
- **Crash-dump/post-mortem routine** Shows the state of variables where an error occurs
- **Stack contents -** Shows sequencing through procedures/modules/ recursive algorithms
- **Cross-referencers** - Identifies where variables/constants are used in a program to avoid duplications

Test Data

Erroneous data

This type of data **should not be accepted** by the program or it will cause an error.

For example, an integer is entered when a string is required.

Boundary data

This type of data **should be accepted** by the program and is **valid**.

It is used to check that values entered at the **boundary** of a range will be accepted.

It also checks that data **just outside** the range is handled by the program.

Normal data

This type of data **should be accepted** by the program and is **valid**.

It is the normal data that you would expect to be entered into your program.

Worked Example

The following algorithm is intended to calculate and print the average mark for each student in a class, for all the tests they have attempted:

```
// average mark
students = input("How many students? ")

for n = 1 to students
   name = input("Enter student name ")
   totalMarks = input("Enter total marks for ", name)
   numTests =  input("How many tests has this student taken? ")
   averageMark = round(totalMarks/numTests)
   print ("Average mark = ",averageMark)
next n
```

The test plan will look something like this:

Test number	Test data	Purpose of test	Expected result	Actual result
1	Number of students = 4 for tests 1-4 Jo: total marks 27, tests 3	Normal data, integer result	9	9
2	Tom: total marks 31, tests 4	Normal data, non-integer result rounded up	8	8
3	Beth: total marks 28, tests 3	Normal data, result rounded down	9	9
4	Amina: total marks 0, tests 0	No tests taken	0	Program crashes
5	Number of students abc	Test invalid data	Program terminates	

You can probably think of some other input data that would make the program crash.
For example, what if the user enters 31.5 for the total marks? The program should validate all user input, so some amendments will have to be made to the program before general release!

Why have standards?

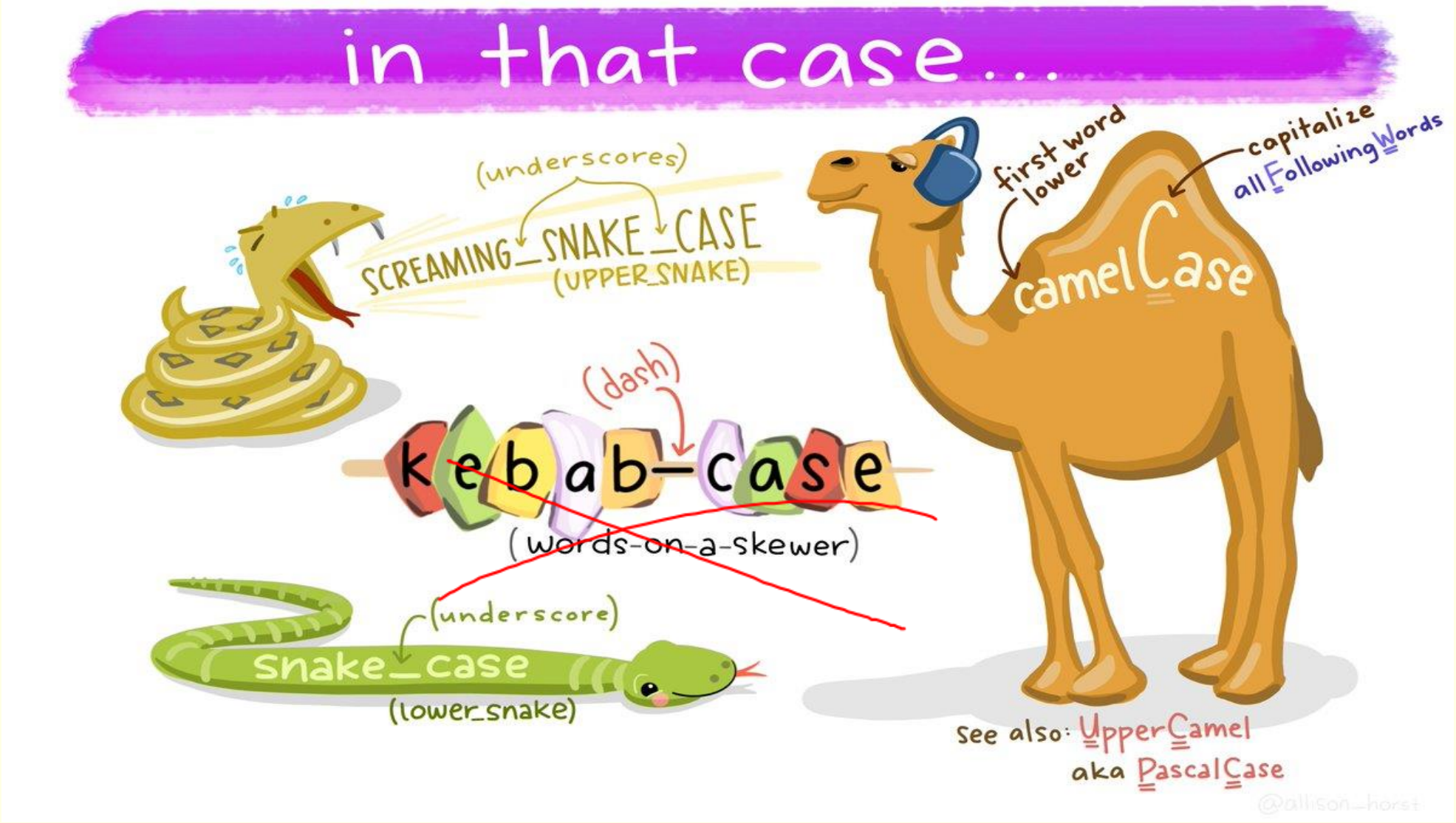
- When working with many team members and multiple projects in an organisation, using a coding standards policy is recommended
- Reduce complexity and aid readability
- This reduces complexity and makes the search for any bugs more straightforward.
- Reduce the errors.
- Easy to maintain
- Cost-efficient

Standards

Guidelines could include:

- You should always try to use meaningful names for variables, rather than x, y and z, as this helps to make the program easy to follow and update when required.
- It is also helpful, within a team of programmers, to have standards for naming variables and constants, as this will leave less room for errors and inconsistencies in the names in a large program.
- Start all variable names with a lowercase letter
- Do not use underscores in the middle of variable names
- Use "camelCaps" to separate parts of a variable name – for example, timeInMinutes, maxTemperature
- Do not use overly long names but keep them meaningful – maxTemp is better than maximumTemperature if there is not likely to be any confusion over the meaning of max
- Use all UPPERCASE letters for constants, which are then instantly identifiable
- When defining a class in object-oriented programming, start with an uppercase letter, with the rest of the class name lowercase
- Variables must not be set up outside the scope of a function: **this sets a limit on where to look for bugs and reduces the likelihood of a problem spread across many modules.**

in that case...


(underscores)
SCREAMING_SNAKE_CASE
(UPPER_SNAKE)
(dash)
kebab-case
(words-on-a-skewer)
(underscore)
snake_case
(lower_snake)
first word lower
capitalize all Following Words
camelCase
See also: UpperCamel aka PascalCase
@allison_horst

Function rules

- No function may be longer than a single complexity and aid readability.
- Comment your code - Do not do line-by look almost unreadable
- Indentation to aid readability
- Follow the single-entry/single-exit rule.
- Never write multiple return statements in the san

```
if (condition)
   return 42;
else
   return 97;
```

"This is ugly, you have to use a local variable!"

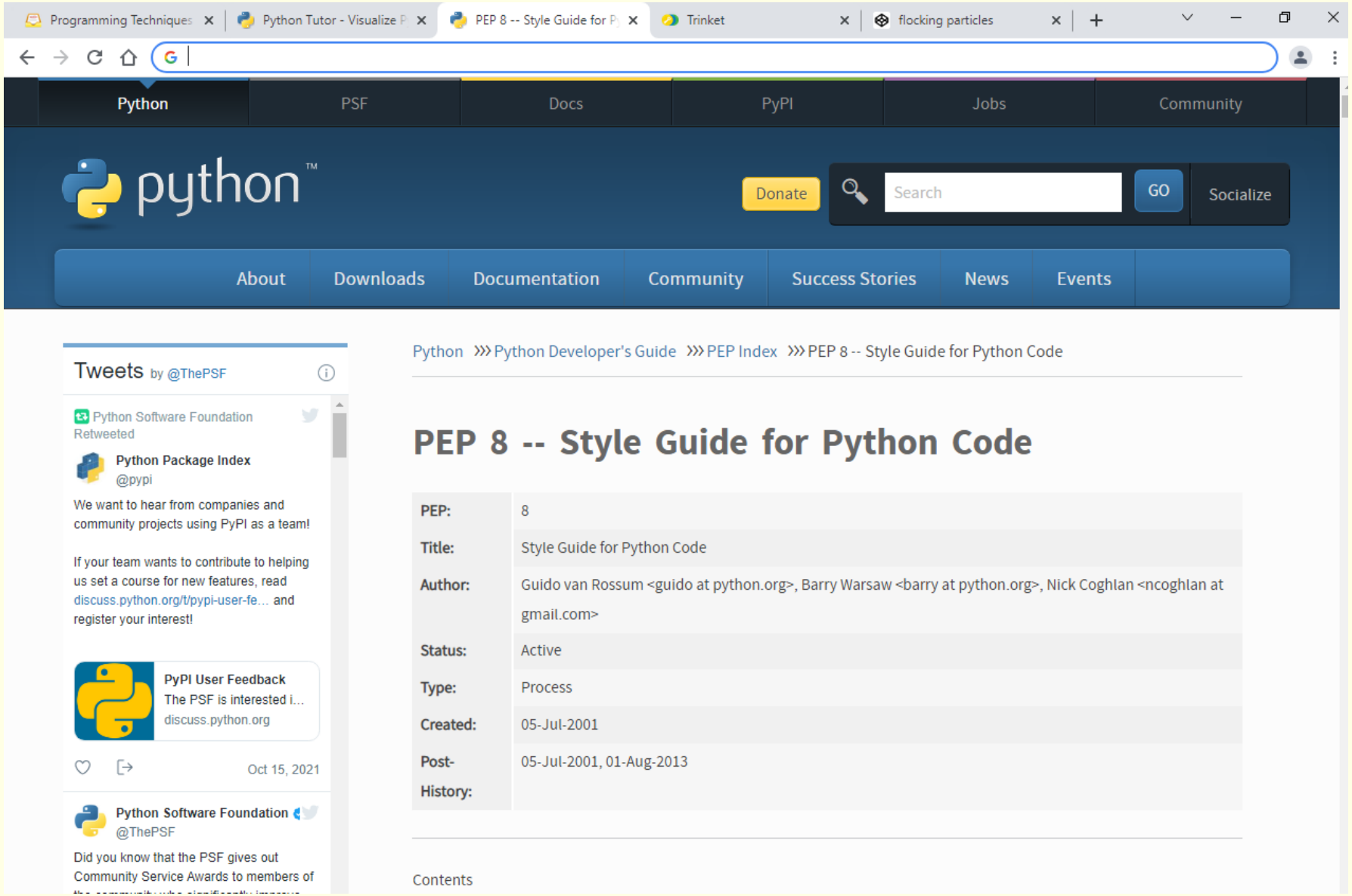
```
int result;
if (condition)
   result = 42;
else
   result = 97;
return result;
```

The main reason multiple exit points are bad is that they complicate control flow.

The more complicated the control flow is, the harder the code is to understand.

The harder the code is to understand, the greater the change of introduction bugs whenever the code is modified.

Official Python programming standards (PEP 8)



Programming Standards Summary

- Code Comment and Documentation
- Use of Indentation
- Commenting on Obvious Things
- Grouping Code
- Proper and Consistent Scheme for Naming
- CamelCase & UnderScore
- Deep nesting structure should be avoided