Introduction To Python

MSc / PGDip Bootcamps

Noel Cosgrave
noel.cosgrave@ncirl.ie

National College of Ireland

Session Overview

This introductory session is intended to cover the basics of the Python language, including:

- Data types, literals and variables;
- Operators and expressions;
- Conversion and coercion;
- Program control structures;
- Data structures; and
- Functions.

Python Bootcamp

Data Types, Literals, Variables, Operators & Expressions

Data Types

True / False | ’b’ | 94.304 | 431257 | ’Hello World’

In programming, we refer to data as having a type. What does this mean in practice?

Different data types have associated sets of valid values:

- A boolean datatype can take on values from the set {True, False}
- An integer datatype can take on values from the set {. . . , -2, -1, 0, +1, +2, . . . }

Data Types

Is taking the square root of a **string** value containing a person's name a valid operation?

Does it make sense to divide the **boolean** value True by the boolean value False?

Data types also determine the **set of valid operations** that can be performed on the data.

In other words, a datatype is **both** a set of possible values **and** the set of operations that can be performed on them.

Literals

A literal is a direct and constant (fixed) representation of a value.

Numeric Literals

```
45          # this is an integer literal
```

45

```
3.14        # this is a floating point literal
```

3.14

```
5.60204e-12 # this is another floating point literal
```

5.60204e-12

String Literals

```
'Hello World!'
"Hello Again!"
```

Literals

String Literals

A string literal is a direct representation of a sequence of characters:

- Delimited by a pair of matching single or double quotes
- may contain zero or more characters:
  - Letters
  - Numbers
  - Punctuation marks
  - Control characters

Literals

String Literals

Python Examples:

```
'Hello World!' # String literal
```

‘Hello World!’

```
"Python's World \t" # String literal with TAB
```

"Python’s World "͡

```
' ' # Single space character
```

’ ’

```
'' # Empty string
```

”

Literals

Character representation in string literals

Characters in a string need to be represented as binary numbers (encoded) in some way. Various encoding schemes are implemented in Python:

- The Unicode character set, encoded as UTF-8 or UTF-16
- Standard ASCII and its regional variants.

The ASCII encoding of the string '123' is:

$$\overbrace{00110001}^{\text{'1'}}\ \overbrace{00110010}^{\text{'2'}}\ \overbrace{00110011}^{\text{'3'}}$$

Compound Data Types

```
{
   id: 87558,
   name: “P. Erson”,
   dob: 01/01/1899
}
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The data type examples we have seen up to now can only take on a single value. These are known as **scalar** data types.

Later on in this session, we will meet data types that can take multiple values, either of the same fundamental base type (e.g. integer) or multiple different base types. These are known as **compound** types.

Variables and Identifiers

- In order to be able to build useful programs, we need the capability to store and process different values on each execution of the program and at different points in the program. We will consider the following related concepts:
  - The **type** - a set of values and their related operations;
  - The **value** - an element of the set of values associated with a type;
  - A **variable** - A location in the computer's memory having both a name and a type associated with it. A variable holds or contains a value of the associated type;
  - A **name** - an alias for an address in memory, used to locate the data it contains; and
  - **Memory** - a collection of variables as well as the computer code that operates on them.

Variables and Identifiers

Assignment

- In Python, we assign a value to a variable using the assignment operator **=**.

```
total = 9000
```

- We can think of a variable as a name (identifier) associated with a value.
- In the example above the variable **total** is associated with the value **9000**.

Variables and Identifiers

Assignment

- Wherever a variable appears in a program on the right-hand side of an assignment statement, it is the **value** associated with the variable that is used, and not the variable's **name**.

```
total = 9000
total = total + 1000
print(total)
```

```
10000
```

- In the above example, first the literal value **9000** is assigned to the variable **total**.
- Next the literal value **1000** is added to the value stored in the variable **total**.
- Finally the value stored in the variable **total** is output.

Variables and Identifiers

Identifiers

An **identifier** is a sequence of one or more characters used to assign a name for a given element of a program.

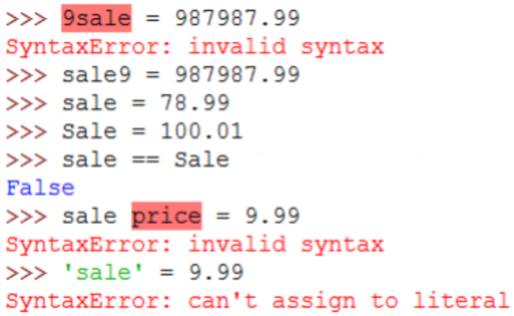
- Variable names are identifiers
- A programming language's keywords are identifiers

In Python, as in all programming languages, identifiers for variable names must adhere to a set of rules. The rules for Python are:

- Identifiers may contain letters and digits
- Identifiers cannot begin with a digit
- The underscore '_' character can be a part of an identifier, but not as the first character.
- Spaces are not permitted as part of an identifier.

Variables and Identifiers

Identifiers



```
>>> 9sale = 987987.99
SyntaxError: invalid syntax
>>> sale9 = 987987.99
>>> sale = 78.99
>>> Sale = 100.01
>>> sale == Sale
False
>>> sale price = 9.99
SyntaxError: invalid syntax
>>> 'sale' = 9.99
SyntaxError: can't assign to literal
```

== is the comparison operator. This checks for equality between two entities and returns a boolean value.

Note: Python is a **case senstive** and **dynamically typed** language.

Variables and Identifiers

Keywords

A keyword is an identifier with a pre-defined meaning in a programming language. Python 3 has 35 keywords:

```
False       class      from       or
None        continue   global     pass
True        def        if         raise
and         del        import     return
as          elif       in         try
assert      else       is         while
async       except     lambda     with
await       finally    nonlocal   yield
break       for        not
```

Operators and Expressions

Operators

In earlier slides we saw that a set of operations is associated with a given datatype

- An operator is a symbol that represents an operation that may be performed on one or more values of a given datatype
- The values that are operated on are called the operands
- A **unary** operator operates on a single operand. The **negation** operator is a unary operator.
- A **binary** operator operates on a pair of operands. The **addition** operator is a binary operator.

Operators and Expressions

Expressions

- We can combine operators and operands to form expressions.
- An single literal or reference to a variable can also be an expression.
- We can think of expressions being made up of **sub-expressions**
- A **sub-expression** is any expression that forms part of a larger expression
- Expressions that evaluate to a numeric value are called **arithmetic expressions**

Consider the expression
(x + y) * 3
What are the sub-expressions?

Operators and Expressions

Arithmetic Operators

Operator	Operation	Example	Result
+	Unary Plus	+1	1
-	Unary Minus	-1	-1
+	Binary Plus	1 + 1	2
-	Binary Minus	4 - 1	3
*	Multiplcation	4 * 2	8
/	Division	9 / 1.5	6.0
%	Modulus	9 % 4	1
**	Exponentiation	10 ** 3	1000
//	Floor Division	10 // 3	3

Operators and Expressions

Comparison Operators

Operator	Operation	Example	Result
==	Equals	5 == 2	False
!=	Not Equals	5 != 2	True
>	Greater Than	6 > 3	True
<	Less Than	6 < 3	False
>=	Greater Than Or Equal To	10 >= 4	True
<=	Less Than Or Equal To	8 <= 4	False

Comparison operators are never used to compare two literal values, as is the case with the examples aboe. While they can be used in this way, it makes little sense. They are used to compare two variables or a variable and a literal, as shown on the following slide.

Operators and Expressions

Comparison Operators

```
a = 4
b = 10
print(a == b)
```

```
False
```

```
print(a != b)
```

```
True
```

```
print(a < b)
```

```
True
```

Operators and Expressions

Logical Operators

A logical operator is used to operate on one or more expressions that return logical (Boolean) values.

Operator	**Operation**	**Example**	**Result**
and	Logical AND	True and False	False
or	Logical OR	True or False	True
not	Logical negation	not True	False

Logical operators are commonly used to create compound conditional statements in *if*… *elif*.. *else*… and *while* constructs.

Operators and Expressions

Logical Operators

```python
a = True
b = not a
c = True
if a and b :
   print('Both a and b are True')
else :
   print('Either or both of a and b are False')
```

```
Either or both of a and b are False
```

```python
if a or b :
   print('Either one or both of a or b is True')
else :
   print('Neither a nor b is True')
```

```
Either one or both of a or b is True
```

Operators and Expressions

Bitwise Operators

Bitwise operators operate on individual bits (ones and zeros) in the operands.

Operator	Type	Operation	Example	Result
&	Binary	Binary AND	2 & 3	2
|	Binary	Binary OR	2 | 3	3
^	Binary	Binary NOT	2 ^ 3	1
~	Unary	Ones Complement (Bit flip)	~3	-4
«	Binary	Shift Left	4 « 2	16
»	Binary	Shift Right	4 » 2	1

Operators and Expressions

Bitwise Operators - AND

- Decimal 40 is `00101000` in binary.
- Decimal 14 is `00001110` in binary
- The result of a bitwise AND would be `00001000`
- `00001000` in Binary is 8 in Decimal

```
print(40 & 14)
```

```
8
```

Operators and Expressions

Bitwise Operators - OR

- Decimal 40 is `00101000` in binary.
- Decimal 14 is `00001110` in binary
- The result of a bitwise OR would be `00101110`
- `00101110` in Binary is 46 in Decimal

```
print(40 | 14)
```

```
46
```

Operators and Expressions

Bitwise Operators - XOR

- Decimal 40 is `00101000` in binary.
- Decimal 14 is `00001110` in binary
- The result of a bitwise XOR would be `00100110`
- `00100110` in Binary is 38 in Decimal

```python
print(40 ^ 14)
```

```
38
```

Operators and Expressions

Bitwise Operators - Ones Complement

- Decimal 40 is `00101000` in binary.
- The result of a ones complement (bit flip) would be `11010111`
- `11010111` in Binary is 215 in Decimal

```
print((~ 40) & 0xff)
```

```
215
```

Here be dragons!

Python does not have an unsigned integer type. The sign of the number is stored with the number as one bit. Flipping the bits will flip the sign bit too. In order to get the correct result, we need to *mask out* the sign using the & operator and a hexadecimal number, 0xf for 8 bit signed numbers, 0xff for 16-bit signed numbers, 0xffff for 32-bit signed numbers and so on.

Operators and Expressions

Identity Operators

- The **is** operator is used to test if two variables are the same object stored at the same memory location.
- Similarly, the **is not** operator can be used to test if two variables are not the same object.
- Because of **string interning**, two string variables set to the same literal value will be stored at the same memory location. Different strings will not. Note that the presence of a space in a string seems to disable string interning.

Operators and Expressions

Identity Operators

```python
a = 'Hello'
b = 'Hello'
print(a is b)
```

```
True
```

```python
b = 'Hello'[::-1] # reverses the string
print(a is not b)
```

```
True
```

Operators and Expressions

Membership Operators

Membership operators are used to test for membership of a set.

The **in** operator will return True if the result of an expression is a member of a set.

The **not in** operator will return True if the result of an expression is not a member of a set.

```
a = 4
b = [1,2,3,4,5]
print(a in b)
```

True

```
print(a not in b)
```

False

Operators and Expressions

Assignment Operators

Operator	Description	Example	Result
=	Assign	a = 4	a is 4
+=	Add and assign	a += 7	a is now 11
-=	Subtract and assign	a -= 2	a is now 9
*=	Multiply and assign	a *= 5	a is now 45
/=	Divide and assign	a /= 3	a is now 22.5
%=	Assign modulus	a %= 3	a is now 7.5
//=	Floor division and assign	a //= 3	a is now 2.0
**=	Raise to a power and assign	a **= 2	a is now 4.0

Operators and Expressions

Operator Precedence

Expressions with operators of different types are evaluated in descending order shown in the table below.

Operators	Operations
**	Exponentiation
~,-,+	Ones complement, unary minus and unary plus
*,/,//, %	Multiplication, division, floor division and modulus
+,-	Addition and Subtraction
«,»	Bitwise shift left and bitwise shift right
&	Bitwise AND

Operators and Expressions

Operator Precedence

Continued from the previous slide

Operators	Operations
|	Bitwise XOR
^	Bitwise OR
< ,<= ,== ,>= ,> in, not in, is, is not	Comparison, membership and identity operators
not	Logical NOT
and	Logical AND
or	Logical OR

Operators at the **same level of precedence** are evaluated according to the **rules of associativity**, which is **left to right** except for the exponentiation operator, which is evaluated right to left.

Operators and Expressions

Mixed-Type Expressions

- A mixed-type expression is an expression containing operands of different types
- The CPU requires that values have the same internal representation before operations can be performed
- Operands in a mixed-type expression need to be converted to a common rerpesentation prior to the operations being perfomed

Conversion   Coercion

Operators and Expressions

Conversion & Coercion

- **Coercion** is the automatic (implicit) conversion of operands to a common type. This will occur if there is **no loss of information**.

  e.g. int → float

- **Conversion** is the explicit changing of operands to a common type and **may result in loss of information**. It is done using type-conversion or **typecasting** functions.

  e.g. 2 + int(3.4) → 2 + 3 → 5

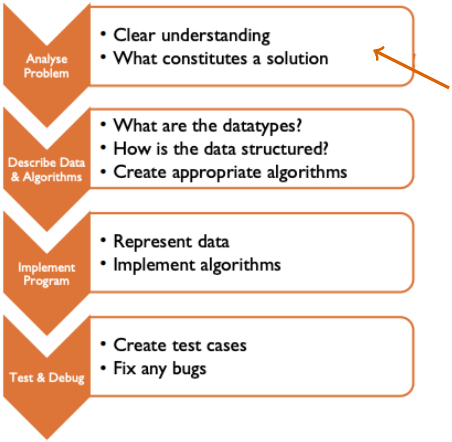
Summary So Far

In this section of the Python bootcamp we have looked at:

- Literals;
- Data types;
- Operators and expressions;
- Operator precedence and associativity; and
- Data type coercion and conversion.

Practice

Write a Python Program that transforms a value in **euro** into a value in **cents**. The amount of euro should be read into the program as input from the keyboard and the amount of cents should be printed as output to the screen.



Hints

INPUT: eurInput

OUTPUT: eurCentOutput

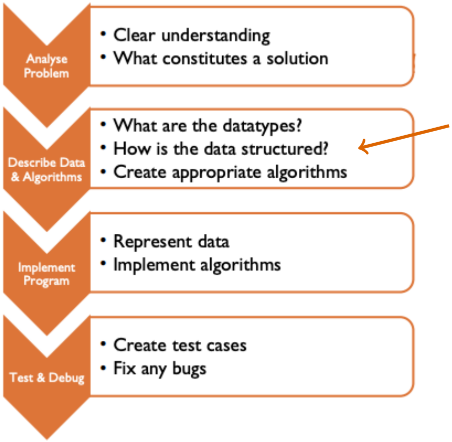
The following formula can be used to convert from euro to euro cents:

```
eurCentOutput = 100 * eurInput
```

where eurInput represents the input value and eurCentOutput represents the output value

Practice

Write a Python Program that transforms a value in **euro** into a value in **cents**. The amount of euro should be read into the program as input from the keyboard and the amount of cents should be printed as output to the screen.


Analyse Problem
• Clear understanding
• What constitutes a solution
Describe Data & Algorithms
• What are the datatypes?
• How is the data structured?
• Create appropriate algorithms
Implement Program
• Represent data
• Implement algorithms
Test & Debug
• Create test cases
• Fix any bugs

Hints

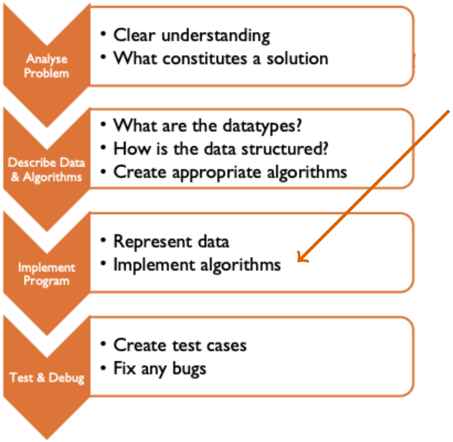
What type of data does the EUR variable will need to hold?

Will type conversion be required prior to performing calculations?

What are the steps that the algorithm will need to take?

Practice

Write a Python Program that transforms a value in **euro** into a value in **cents**. The amount of euro should be read into the program as input from the keyboard and the amount of cents should be printed as output to the screen.



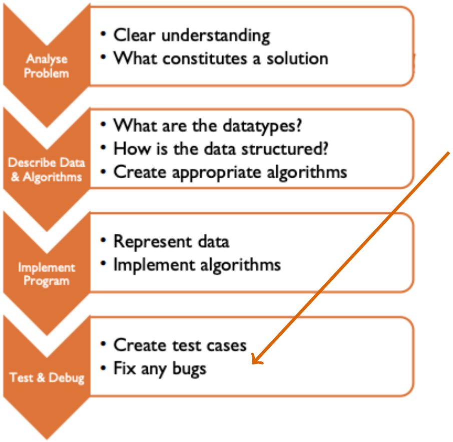
Analyse Problem
• Clear understanding
• What constitutes a solution
Describe Data & Algorithms
• What are the datatypes?
• How is the data structured?
• Create appropriate algorithms
Implement Program
• Represent data
• Implement algorithms
Test & Debug
• Create test cases
• Fix any bugs

Hints

To accept user input you can use the *input()* function. This function reads a line of input and converts the input to a string (removing the trailing new-line character). Convert from a string to a float by using the *float()* typecasting function.

Practice

Write a Python Program that transforms a value in **euro** into a value in **cents**. The amount of euro should be read into the program as input from the keyboard and the amount of cents should be printed as output to the screen.


Analyse Problem
• Clear understanding
• What constitutes a solution
Describe Data & Algorithms
• What are the datatypes?
• How is the data structured?
• Create appropriate algorithms
Implement Program
• Represent data
• Implement algorithms
Test & Debug
• Create test cases
• Fix any bugs

Hints

Create test cases for different steps of the algorithm and for different input numbers and test your program.

Fix any bugs

Python Bootcamp

Program Control

Program Control
Introduction

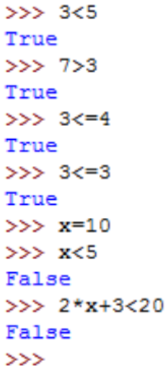
There are three fundamental types of program control statements:

- **Sequence statements** - Program instructions are evaluated in the order they are written and have a definite beginning and end point.
- **Selection statements** - Instructions can take one of a number of paths depending on the evaluation of a conditional expression.
- **Repetition statements** - A group of instructions can be executed multiple times, with the number of repeats being controlled by a conditional expression or counter.

Program Control

Selection statements

- Selection statements allow the program to **choose between two alternative paths**.
- A selection statement must include a **conditional expression**. Such expressions always return a boolean (True/False) value.



```
>>> 3<5
True
>>> 7>3
True
>>> 3<=4
True
>>> 3<=3
True
>>> x=10
>>> x<5
False
>>> 2*x+3<20
False
>>>
```

Program Control

*Selection statements - the simple **if** statement*

Python syntax:

```
if condition :
   indentedStatementBlock
```

Program flow:

if the condition is true then the *indentedStatementBlock* is executed otherwise the *indentedStatementBlock* is skipped.

Python example:

```python
x = 10
if x < 15 :
   print('x is less than 15')
```

```
x is less than 15
```

Try this with other literals on the right hand side of the conditional test.

Program Control

*Selection statements - the **if - else** statement*

Python syntax:

```
if condition :
  indentedStatementBlockForTrueCondition
else :
  indentedStatementBlockForFalseCondition
```

Program flow:

if the condition is true then the *indentedStatementBlock* is executed otherwise the *indentedStatementBlock* is skipped.

Program Control

*Selection statements - the **if - else** statement*

Python example:

```
number = int(input('Please enter a number: '))
if number % 2 == 0 :
    print(str(number) + " is even.")
else :
    print(str(number) + " is odd.")
```

```
Please enter a number: 42
42 is even.
```

Program Control

*Selection statements - the **if - elif - else** statement*

Python syntax:

```
if condition1 :
  indentedStatementBlockForCondition1
elif condition2 :
  indentedStatementBlockForCondition2
elif condition3 :
  indentedStatementBlockForCondition3
...
else :
  indentedStatementBlockForAllConditionsFalse
```

Program flow:

If *condition1* is true then the *indentedStatementBlockForCondition1* is executed, otherwise if *condition2* is true than the *indentedStatement-BlockForTrueCondition2* is executed, and so on until the else clause is reached. Note that the entire statement will terminate after the first condition evaluates to True.

Program Control

*Repetition statements - the **while** statement*

The while statement executes a group of instructions while a condition remains true.

Python syntax:

```
initialisation
while loop-continuation-condition :
  indentedStatementBlock
  prepare variable values for the next itera-
tion
```

Program flow:

While the *loop-continuation-condition* remains true the *indentedStatementBlock* is executed.

Program Control

*Repetition statements - the **while** statement*

Python example:

```
i = 5
while i < 10 :
    print(i)
    i = i + 2
```

```
5 7 9
```

Program Control

*Repetition statements - the **for** statement*

The **for** construct is a counter-controlled loop. We know how many times the loop will execute.

Python syntax:

```
for element in sequence :
  indentedStatementBlock
```

Program flow:

The indented statements are repeated for each element in the sequence.

Program Control

*Repetition statements - the **for** statement*

Python examples:

```
for value in [1,2,3] :
    print(value)
    print("repeat"*value)
```

1 repeat 2 repeatrepeat 3 repeatrepeatrepeat

```
for colour in ["red","green","blue"] :
    print(colour)
```

red green blue

Program Control

Flow Charts

Flow charts are used to visually describes the flow of a program.

It uses different predefined symbols to denote different actions:

- Start & Stop
- Input & Output
- Flow Direction
- Generic processing steps
- Conditional jumps
- Junctions

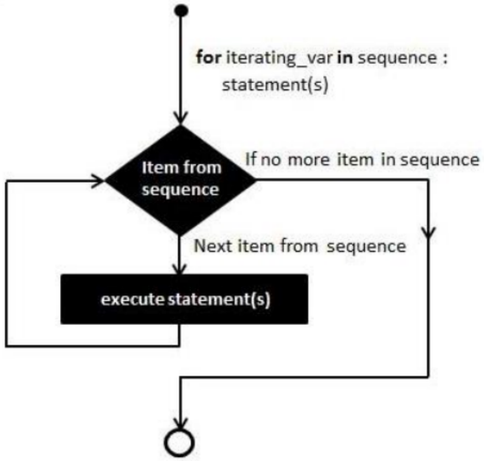
Program Control

Flow Charts

Name	Symbol	Use in Flowchart
Oval	⬭	Denotes the beginning or end of the program
Parallelogram	▱	Denotes an input/output operation
Rectangle	▭	Denotes a generic Process step to be carried out e.g. addition, subtraction, division etc.
Diamond	◇	Denotes a decision (or branch) to be made. The program should continue along one of two routes. (e.g. IF/THEN/ELSE)
Flow line	→	Denotes the direction of logic flow in the program

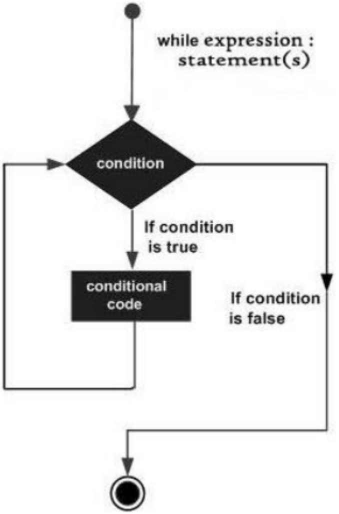
Program Control

*Flow Charts - **for** statement example*



Program Control

*Flow Charts - **while** statement example*



Summary So Far

In this section of the Python bootcamp, we have looked at:

- Program Control & Program Control Statements
- Sequence Statements
- Selection Statements
- Repetition Statements
- Flow diagrams

Putting It Into Practice

In-class exercise

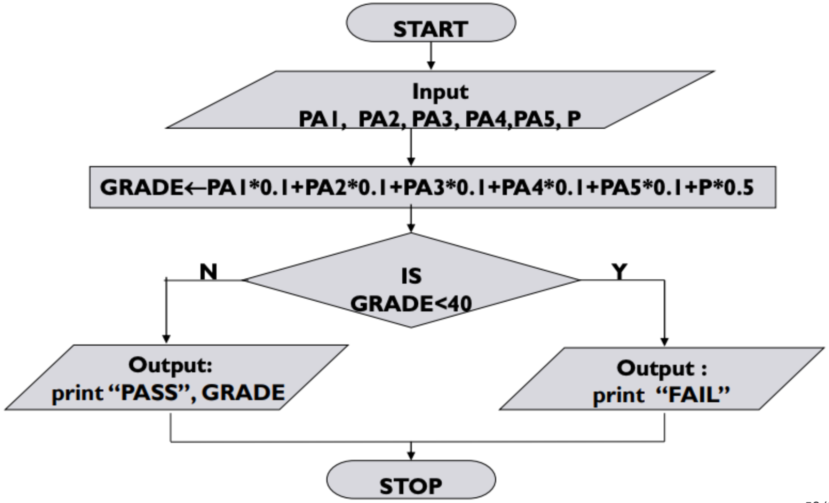
Create a Python program that implements the marking scheme for a module with the following specification:

- There are 5 practical assessments worth 10% each
- There is a project worth 50% .
- The mark for the practical assessments and project must be given as input.
- If a student's grade is less than 40 is a fail
- The program should print if a student passed the module and with what mark, or if the student failed

A sample solution will be made available on Moodle.

Putting It Into Practice

Flowchart for in-class exercise


START
Input
PA1, PA2, PA3, PA4,PA5, P
GRADE←PA1*0.1+PA2*0.1+PA3*0.1+PA4*0.1+PA5*0.1+P*0.5
IS
GRADE<40
N
Y
Output:
print "PASS", GRADE
Output :
print "FAIL"
STOP

Python Bootcamp

Input/Output, Data Structures & Functions

Input and Output

File I/O is needed when:

- We need to permanently store data when we need to access the results of a program again without re-running the program.
- We want to access the data using another computer.

File paths can be **relative** or **absolute**.

- An absolute file path contains the complete path to a file starting at the root of the file system, e.g.:

  `/Users/MyUser/Documents/file.txt or C:\Users\MyUser\Documents\file.txt`

- A relative file path is the path to the file relative to the current working directory.

  `../Documents/file.txt  or ..\Documents\file.txt`

Input and Output

Opening a file

The *open()* function is used to open a file. The function takes two arguments, a string specifying the filename and a string specifying the file access mode.

Python syntax:

```
filevariable = (filename,mode)
```

Python example:

```
fh = open('cities.txt','w')
```

Input and Output

Opening a file - file access modes

- r - Opens a text file for reading.
- w - Opens a text file for writing. This will overwrite the contents of the file if it already exists.
- a - Opens a text file, appending any new lines to the end of the file.
- rb - Opens a binary file for reading.
- rw - Opens a binary file for writing.

Input and Output

Reading from, writing to and closing a file

Writing to a file

```
fh.write('Dublin\n')
fh.write('London\n')
fh.write('Paris\n')
fh.close()
```

Reading from a file

```
fh = open('cities.txt','r')
print(', '.join(line.rstrip() for line in fh.readlines()))
```

Dublin, London, Paris

```
fh.close()
```

Input and Output

File methods

read(size) reads all of the data in a file if *size* is omitted and if *size* is specified, it will read the number of bytes (binary mode) or characters (text mode) specified by the *size* argument.

readline() reads a single line from the file (text mode), retaining the newline character at the end of the string.

readlines() reads the entire contents of a file (text mode) and returns a list with each line as an element.

write() takes a string (text mode) or a bytes object (binary mode) and writes it to the file.

writelines() takes a list of strings (text mode) and writes each element as a separate line.

Input and Output

File methods

tell() returns an integer giving the current position in the file (binary mode) or an opaque number (text mode). An opaque number in this instance means that the returned value can only be used as an argument to seek() and can't be otherwise used.

seek(offset,whence). Moves to a position in the file determined by adding an *offset* to the *whence* argument, where *whence=0* is from the beginning of the file, *whence=1* is the current file position, and *whence=2* is the end of the file. When no value is supplied for *whence* it defaults to 0.

Data Structures

Data structures are a method of organising information in the computer to permit the representation of complex data types and operations on those datatypes. There are numerous such structures:

- Lists
- Arrays
- Sequences
- Sets
- Dictionaries
- Abstract data types

Lists

A list is a ordered set of values, where each value is identified by an index.

The values of a list are called **elements**.

The order of an element in a list is called its **index**.

Python Examples:

```
a = [1,2,3]
print(a)
```

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

```
a = list([1,2,3])
print(a)
```

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

```
a = list('abcdefg')
print(a)
```

```
[‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’]
```

Lists

Accessing elements

Any element at the index position in a list can be accessed using the syntax:

```
list_name[index specification]
```

```
my_list = [0,1,2,3,4,5,6,7,8,9,10]
my_list
my_list[2]
```

```
2
```

```
my_list[11]
```

```
IndexError:  List index out of range
```

List Operations

Finding the number of elements in a list – the length of a list:

```
len(list_name)
```

```
my_list = [1,2,3,4,5]
list_length = len(my_list)
list_length
```

```
5
```

Summing all the elements in a list:

```
sum(list_name)
```

```
sum(my_list)
```

```
15
```

List Operations

Finding smallest element in a list:

```
min(list_name)
```

```
min(my_list)
```

```
1
```

Finding largest element in a list:

```
max(list_name)
```

```
max(my_list)
```

```
5
```

List Operations

Getting a slice from a list from a start index to an end index - 1:

```
list_name[start_index:end_index]
```

```
my_list = [0,1,2,3,4,5,6,7,8,9,10]
my_list
my_list[5:10]
```

```
[6, 7, 8, 9, 10]
```

Shuffling the elements of a list randomly (in place):

```
random.shuffle(list_name)
```

```
import random
print(random.shuffle(my_list))
```

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

List Methods

- Add an element to the list:

  ```
  list.append(x)
  ```

- Insert an item at a particular position in a list:

  ```
  list.insert(i, x)
  ```

- Remove the first item from the list whose value equals the argument *x*:

  ```
  list.remove(x)
  ```

- Remove the item at the given position in the list, and return it:

  ```
  list.pop([i])
  ```

  If no index is specified, *pop()* removes and returns the last item in the list.

List Methods

- Return the index in the list of the first item whose value equals the argument *x*:

  ```
  list.index(x)
  ```

- Return the number of times the argument *x* appears in the list.

  ```
  list.count(x)
  ```

- Sort the items of the list, in place

  ```
  list.sort()
  ```

- Reverse the elements of the list, in place.

  ```
  list.reverse()
  ```

Lists

Abstract data types based on lists

Stacks

- In a stack, the last element added is the first element retrieved. This is referred to as LIFO or “last-in, first-out”.
- To add elements to a stack use the *append()* method.
- To retrieve an element from a stack use *pop()* without supplying an argument for the index.

Queues

- In a queue, the first element added is the first element retrieved. This is referred to as FIFO or “first-in, first-out”.
- Performing inserts or pops from the beginning of a list is slow
- To implement a queue, use *collections.deque*.

List Comprehensions

List comprehensions are a simple, concise way of creating new lists based on other, pre-existing lists. The generic syntax of a list comprehension is:

```
expression for element in list
```

The expression is evaluated once for each element in the list.

Python Example:

Creating a list of all odd numbers in the range 0 to 30.

```
odd30 = [ x for x in range(31) if x % 2 == 1]
print(odd30)
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
```

Arrays

An **array** is a data structure similar to list, but unlike a list it is designed to store data all of the same type.

```
arrayName = array(typecode,[Initialiser]))
```

An initialiser (a list) can be passed to the array function, or an array can be created using the *fromlist()* or *fromstring()* methods.

A full list of supported data types and their respective type codes can be found in the Python documentation.

Arrays

Python example:

```
from array import *
my_array = array('i',[1,2,3,4,5])
for i in my_array : print (i)
```

1 2 3 4 5

The initialiser will be coerced to the specified typecode where possible.

```
my_array = array('f',[1,2,3,4,5])
for i in my_array : print (i)
```

1.0 2.0 3.0 4.0 5.0

Arrays

Accessing array elements

Any element at the index position in a array can be accessed using the syntax:

```
array_name[index specification]
```

```
my_array = array('i',[1,2,3,4,5])
my_array[2]
```

```
3
```

```
my_array[11]
```

```
IndexError:  array index out of range
```

Array Methods

- Append a new item with value equal to the argument *x* to the end of an array:

  `array.append(x)`

- Return the number of times the argument *x* appears in an array.

  `array.count(x)`

- Append items to an array from the list argument *l*.

  `array.fromlist(l)`

- Append items to an array from the string argument *s*.

  `array.fromstrings`

- Return the index position of the first occurrence of the argument *x*.

  `array.indexx`

Functions

Introduction

- A function is a sequence of **reusable** statements that performs a particular operation
- A function contains two distinct parts:
  - a **header**; and
  - a **body**

```
def functionName(parameter_list) :
   functionStatementblock
```

Functions

The function header and body

- The function **header** includes the **name** of the function and a list of **parameters**, telling the Python interpreter what kind of data it should expect to receive as arguments.
- Note that we do not provide data types for the function parameters as is the case with some other languages. The data types are determined through **type inference** at runtime.
- The function **body** consists of one or more statements that specify what the function does. This can include a optional statement that **returns a value**.

Functions

Example without a return statement

A function that takes a string argument containing the name of the user. The function then prints a personalised salutation to the user:

```
def salutation(name) :
    salutation = 'Hello ' + name
    print(salutation)
```

The function is then called or **invoked** by referring to it by name, followed by the list of arguments in parentheses.

```
salutation('Hal')
```

```
Hello Hal
```

Functions

Example with a return statement

A function that takes a single number (integer) argument and returns a list of all possible factors of the number:

```
def factors(x):
   factorlist = []
   for i in range(1, x + 1):
       if x % i == 0:
           factorlist.append(i)
   return(factorlist)
```

Functions

Example with a return statement

The function is then invoked by referring to it by name, followed by the list of arguments in parentheses. The return value from the function can be stored to a variable or printed directly.

```
myfactors = factors(125)
print(myfactors)
```

```
[1, 5, 25, 125]
```

```
print(factors(248))
```

```
[1, 2, 4, 8, 31, 62, 124, 248]
```

Functions

Returning multiple values - the yield statement

If we want our functions to return multiple values, for instance outputting a value for each iteration through a for or while loop, the **return** statement is of no use to us. This is where the **yield** statement comes in. The example below will output each row in turn, as soon as it has split the row into columns:

```python
def csv_reader(file_name, file_mode, separator) :
    for row in open(file_name, file_mode) :
        yield row.split(separator)
```

Functions

Variable Scope

- The concept of **variable scope** refers to the accessibility or visibility of a variable a various points with in a program.
- A **local** variable is declare within the function body and can only be accessed from inside the function. It is not visible to code outside of the function.
- A **global** variable is declared outside a function and can be accessed anywhere in the program.
- Pay attention to variable scope, as it can be the source of errors.

Functions

Variable Scope

As you have already learned, variables declared inside a function are locally scoped. However, there is one exception to this rule. If a variable is declared inside a function using the **global** keyword, its scope will be global.

```
def globaldemo(x):
    global y
    y = x * x

globaldemo(10)

print(y) # without the global keyword, this would halt
         # with an error. Try it yourself!
```

```
100
```

Functions

Default Parameter Values

A function’s parameters may be given **default values**, meaning that specifying their value when invoking the function is optional. When the value is not specified, the default value will be used instead. This example will take a character string and will either print it horizontally or vertically

```python
def fancyprint(text, orientation='horizontal') :
    if orientation == 'horizontal' :
        print(text+'\n')
    elif orientation == 'vertical' :
        for character in text :
           print(character+'\n')
```

Functions

Default Parameter Values

```
fancyprint('Hello') # This should print the text horizontally
```

```
Hello
```

```
fancyprint('Hello',orientation='vertical') # and this vertically
```

```
H

e

l

l

o
```

Lambda Functions

- Inspired by the LISP language and lambda calculus, a lambda function is an unnamed or **anonymous** function that can have any number of parameters, but is limited to one expression on a single line of code.
- A lambda function is an expression , and not a statement, and because of this it can be used in parts of your code where a full function definition would not be permitted, such as inside list comprehensions or the arguments of a function call.
- Unlike a regular function, a lambda expression will return automatically . Because it is limited to a single expression, it is less expressive than a full function definition. By its very nature, we can only squeeze so much logic into a lambda body.

Lambda Functions

- Lambda functions come into their own when used as an anonymous function inside a regular function.
- For example, let's assume that you have a function definition that takes one argument, and the argument will be multiplied by an as yet unknown number.

```
def multiplier(n) :
    return lambda x : x * n
```

In true functional programming style, the function multiplier does not return a value, but instead returns a function.

Lambda Functions

- The function definition returned by the multiplier function can now be used, by supplying the relevant values as arguments and giving them an identifier.

```
doubler = multiplier(2)
tripler = multiplier(3)
```

- The identifiers can now be used as multiplication functions:

```
print(doubler(10))
```

```
20
```

```
print(tripler(10))
```

```
30
```

Summary

In this section of the Python bootcamp, we have looked at:

- File input and output
- Compound Data Structures
  - Lists, list operations and list methods
  - List comprehensions
  - Arrays & array methods
- Functions
  - Return and yield statements
  - Lambda functions

Putting It Into Practice

In-class exercise

Modify the Python program created earlier so that it reads the marks for the assessments from a file and writes the results of the calculation to another file.

Example input:

```
75,60,45,50,65,70
40,50,55,52,70,55
```

Example output:

```
64.5,H2.2
54.2,Pass
```