Overview of the data programming language

Analytics Programming & Data Visualisation

Noel Cosgrave
Associate Faculty, National College of Ireland
noel.cosgrave@ncirl.ie

National College of Ireland

Introduction

This second session of the Analytics Programming & Data Visualisation module will provide an overview of:

- Syntax and semantics in Computer Programming;
- Python Expressions and statements;
- Atomic data types in Python;
- Conversion and coercion;
- Compound Types, including built in data structures such as arrays, matrices and lists;
- Indexing and slicing into arrays/lists/matrices; and
- Program flow control structures.

Syntax and Semantics

- Just as in human languages, the **syntax** rules of a programming language define how symbols, reserved words, and identifiers are put together to make a valid program. It is, in effect, the the grammar of a language.
- The **semantics** of a program or routine defines its intent or meaning.
- It is worth noting that a program may be **syntactically correct**, but it does not necessarily mean that it is semantically or logically correct.
- A program or routine will always do what we **tell** it to do, not what we **intended** it to do.

Atomic data types

- In all computer languages, variables and literals (constants) must have a type, whether explicitly declared or implicit from the value it stores.
- Python has several atomic (indivisible) data types
  - Integers
  - Floating-Point Numbers
  - Complex Numbers
  - Characters
  - Boolean

Atomic data types

Integers

Python provides regular *int* and *longint* types, but the latter is not limited to a particular number of bits, instead being limited by available memory. Both are signed. Unsigned integers are available through the *numpy* or *struct* packages.

Even though the longint is theoretically unbounded, calculations with any number less than 2^{-31} or greater than $2^{31}-1$ on a 32-bit platform (less than 2^{-63} or greater than 2^{63} - 1 on a 64-bit platform) will be significantly slower than for numbers within this range. These ranges are the *native* ranges for the CPU, in other words, they can be carried out with single CPU instructions.

As a general rule, any native *n*-bit integer will range from -2^{n-1} to $2^{n-1}-1$.

Atomic data types

Floating-Point Numbers

- If a decimal point is used in a numeric literal then the data value represents a floating-point value

```
3.14
```

- Numbers specified using exponential notation are also created as floating-point values

```
6.62606957e-34
```

Atomic data types

Complex Numbers

- A complex number is a number with a *real* and *imaginary* part.
- Complex numbers take the form shown below, where *x* is the real part and *y* the imaginary part

```
x + yi
```

- Complex numbers can also be created using the *complex()* function. The real and imaginary parts can also be extracted using the *real* and *imag* methods.

```
myComplexNumber = complex(5,3)
print(myComplexNumber.real)
```

```
## 5.0
```

```
print(myComplexNumber.imag)
```

Atomic data types

Boolean

A Boolean type can take on one of two values, true or false. Every object has an implicit boolean value. The following elements or objects are false:

- the *None* type
- False
- 0 (integer, float or complex)
- Empty collections: “ ”, (), [], {}
- Objects from classes that have the special method **nonzero**
- Objects from classes that implements **len** to return False or zero

Expressions and statements

Identifiers

- An identifier is a sequence of one or more characters used to name a given element of a program
- Variable names and function names are identifiers
- A programming language's keywords are identifiers
- In Python, identifiers must adhere to a set of rules:
  - They may contain letters and digits
  - They can not start with a digit
  - The underscore '_' character is allowed but should not be used as the first character in the name
  - Spaces are not allowed as part of a variable name

Python's keywords can be found at
https://docs.python.org/3/reference/lexical_analysis.html#identifiers

Data types

Determining the type and id of a variable or literal

The *type()* function returns the **data type** of the argument passed into the function.

The *id()* function returns the **identity** of an object. This is an integer and is guaranteed to be unique and constant for any object during its lifetime. Note that two objects with non-overlapping lifetimes may have the same id.

```
a = 'I have a type and an identity'
print(type(a))
```

```
## <class 'str'>
```

```
print(id(a))
```

```
## 4936816624
```

Expressions and statements

Variables

In order to allow our programs to process different values each time the program runs we need

- A **variable** is a memory location whose contents can change during the program execution.
- A **variable name** is an identifier used to reference the variable.
- A variable has a **value** associated with of a particular data type.
- Values are associated with variables using the **assignment** operator **=**

```
total = 9000 # variable named total is assigned value 9000
```

Expressions and statements

Operators

- 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, e.g. the negation operator
- A binary operator operates on a pair of operands, e.g. the addition operator

Expressions and statements

Operators

Operator	Description	Example	Result
+	Addition	2 + 3	5
-	Subtraction (binary operator)	3 - 2	1
-	Negation (unary operator)	-3	-3
$*$	Multiplication	7*5	35
/	Division	5/2	2.5
%	Modulus	34%2	0
//	Truncated division	6//4.0	1.0
$**$	Exponentiation	2**3	2**3

The result of truncated division will depend on the datatypes of the operands. The resultant value will be a whole number equivalent represented as either an integer or a float.

Expressions and statements

Operator Precedence

Programming languages specify an operator precedence so that there is a defined order in which to apply the operations in such cases. Python applies the following operator precedence rules:

Operator	Associativity
Exponentiation	right to left
Negation	left to right
Multiplication, Division, Truncating Division and Modulus	left to right
Addition and Subtraction	left to right

Expressions and statements

Operator Precedence

- If two operators are operating on the same operand, the operator with the higher precedence is evaluated first.
- For example, 2 + 3 * 4 is evaluated as 2 + (3 * 4), whereas 2 * 3 + 4 is evaluated as (2 * 3) + 4, because multiplication has a higher precedence than addition.
- If an expression has two operators with the same precedence, the expression is evaluated according the rules of associativity. This is normally left-to-right, with the exception being the exponentiation operation.
- The rules of precedence can be overridden by enclosing parts of the expressions in parentheses.

Expressions and statements

Logical Operators

The following logical operators are designed to work on integer values

- » bitwise left shift (shifts bits left)
- « bitwise right shift (shifts bits right)
- & bitwise and
- b̂itwise exclusive or
- | bitwise or
- ~ bitwise not

Logical (non-bitwise) versions of the *and*, *exclusive or* and *or* operators work on boolean types.

Expressions and statements

Functions

A function is a sequence of reusable statements that performs a particular operation. The syntax of a function in Python without a return value is:

```
def functionName(argument list):
  functionStatementBlock
```

and with a return value, the syntax is:

```
def functionName(argument list):
  functionStatementBlock
  return Expression
```

Expressions and statements

Functions

In the syntax examples on the previous slide, the line starting with *def* is the function header, which contains the function name and the arguments the function takes. The “arity” of a function is the number of arguments it takes. A function that takes an indefinite number of arguments is termed **variadic**.

The *functionStatementBlock* is the body of the function and may contain many lines of code. To return a value from the function, the *return* keyword is used. It takes an expression, such as a variable name as its argument.

Compound Types

Strings

- A string is direct representation of a sequence of characters
- It is delimited by a pair of matching single or double quotes
- Strings may contain zero or more alphanumeric, punctuation or control characters
- Strings in Python are **immutable**, meaning that individual characters in a string can not be changed. However, changes to a string can be saved to a new variable.
- Strings in Python are **interned**, meaning that two strings with the same value reference the same memory location. However, this will only happen if it is initially set as a constant string.

Compound Types

String Immutability

In Python, it is not posssible to change an element in a string. In the example below, the original string contains a spelling error. If the string was mutable, it would be possible to replace the first *i* with a *y*. However, as the string is immutable, this returns an error.

```
a = 'Data Analitics'
a[9] = 'y'
print(a)
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: 'str' object does not support item assignment
```

Compound Types

String Immutability

So how do we modify strings? We can use the *replace()* method, which returns a copy of the string with all or a specified number of occurrences of a substring replaced with another substring.

```
a = 'Data Analitics'
a = a.replace('i','y',1)
print(a)
```

```
## Data Analytics
```

It is also possible to change parts of a string by converting it to a *bytearray*, which is mutable, and then convert back to a string. However, care needs to be exercised if the string contains Unicode characters.

Compound Types

String operators

- Strings can be joined (concatenated) using the **+** operator:

```
a = "Hello "
b = "World"
c = a+b
print(c)
```

```
## Hello World
```

- Strings can be repeated using the * operator:

```
print("Data "*3)
```

```
## Data Data Data
```

Compound Types

String operators

- We can also test if a string is part of (is a substring of) another string using the **in** operator:

```
print("ta" in "Data Analytics")
```

```
## True
```

- The inverse of this test can also be carried out using the **not in** operator:

```
print("ytics" not in "Data Analytics")
```

```
## False
```

Note that these operators can also be used on lists, with similar results.

Compound Types

String formatting

The *format()* method of the string type is allows for multiple placeholders (substitutions) and formatting options. This method can also be used to concatenate elements using positional formatting. The curled braces **{}** are used as placeholder markers.

```
a = "Data Analytics"
b = "Students"
c = "Hello {} {}"
print(c.format(a,b))
```

```
## Hello Data Analytics Students
```

Compound Types

String formatting

A format specifier can also be included inside the placeholder. This determines how the value should be formatted.

- s – as a string
- d – as base 10 (decimal) integers
- f – as floating point
- c – as character
- b – as a binary number
- o – as an octal number
- x – as hexadecimal with lowercase letters
- X – as hexadecimal with uppercase letters
- e – in exponent notation

Compound Types

String formatting examples

```
myMark = 82.15
print("My mark in the exam was {0:3.1f}!".format(myMark))
```

```
## My mark in the exam was 82.2!
```

```
print("My mark in the exam was {0:3.2f}!".format(myMark))
```

```
## My mark in the exam was 82.15!
```

```
print("In hexadecimal that is {0:x}".format(int(myMark)))
```

```
## In hexadecimal that is 52
```

Compound Types

String formatting examples

Note that in examples on the previous slides, **0** is the (zero-based) index of the argument to format. Format is a **variadic** function and as such can take an indefinite number of arguments.

For instance, if it had two arguments, the indexes would be *0* and *1*.

After the colon, the format specifier takes the form [width][.precision]type. In the examples with floating point formatting, the width is 3, the precision is either 2 or 3 and the type is *f* for floating point.

Compound Types

Lists

A list is an ordered set of values.

- Each of the values in the list is termed an *element*
- The location of an element in the list is called its *index*
- The index starts with 0 and has a maximum of $n - 1$ where n is the number of elements in the list

Lists can be created in two ways:

- As a literal, by separating the elements by a comma and enclosing the comma-separated list in square brackets
- By using the list or range constructors

Compound Types

Lists

List creation example, using the literal style

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

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

```
["A","B","C","D"]
```

```
## ['A', 'B', 'C', 'D']
```

Compound Types

List Creation

List creation example, using the list and range constructor

```
list([1,2,3,4])
```

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

```
list()
```

```
## []
```

```
list("abcdef")
```

```
## ['a', 'b', 'c', 'd', 'e', 'f']
```

```
range(1,6)
```

```
## range(1, 6)
```

Compound Types

List Indexing

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

```
listName[index]
```

where *index* is the numeric index of the position of the element, starting at zero.

For example:

```
myList = [9,8,7,6,5,4,3,2,1]
print(myList[4])
```

```
## 5
```

```
print(myList[6])
```

Compound Types

List Indexing

A slice of the list from the start index to the end index less 1 can be returned as follows:

```
listName[start:end]
```

For example:

```
myList = [9,8,7,6,5,4,3,2,1]
print(myList[4:6])
```

```
## [5, 4]
```

Compound Types

List Operations

- Finding the number of elements in a list (the length of a list)

```
myList = range(1,11)
print(len(myList))
```

```
## 10
```

- Finding smallest element in a list

```
print(min(myList))
```

```
## 1
```

Compound Types

List Operations

- Finding largest element in a list

```
print(max(myList))
```

```
## 10
```

- Returning the sum of all list elements

```
print(sum(myList))
```

```
## 55
```

Compound Types

List Methods

A list is an object, and as such has methods that can operate on it

- The method index() will return the index of the first item whose value is the same as that specified as the argument.

```
myList = ['f','a','c','d','d','e','c','b','g']
print(myList.index('c'))
```

```
## 2
```

- The count() method will return the number of times a value appears in the list.

```
print(myList.count('f'))
```

```
## 1
```

Compound Types

List Methods

- The method *sort()* will sort the items of the list, in place

```
myList.sort()
print(myList)
```

```
## ['a', 'b', 'c', 'c', 'd', 'd', 'e', 'f', 'g']
```

- To reverse the elements of the list, in place, use the *reverse()* method

```
myList.reverse()
print(myList)
```

```
## ['g', 'f', 'e', 'd', 'd', 'c', 'c', 'b', 'a']
```

Compound Types

List as stacks and queues

Stacks

- In a stack, the last element added is the first element retrieved ("last-in,first-out")
- Elements are added to a stack using *append()*
- Elements are retrieved from the stack using *pop()* without index

Queues

- In a queue, the first element added is the first element retrieved ("first-in,first-out")
- Performing inserts or pops from the beginning of a list is slow
- To implement a queue, *collections.deque* is used

Compound Types

Arrays

Arrays are similar to lists, except that each element must be of a predeclared datatype.

Arrays are created using the following syntax:

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

The typecode specifies the data type that the list should hold. A full list of typecodes can be found at https://docs.python.org/3/library/array.html

Arrays can be created from lists or strings using the *fromlist()* and *fromstring()* methods.

Compound Types

Array Methods

- To append a new item with value *x* to the end of the array, use *array.append(x)*
- To return the number of occurrences of *x* in an array, use *array.count(x)*
- To append items to an array from a list use *array.fromlist(list)*
- To append items to an array from a string use *array.fromstring(s)*
- To return the index of the first occurrence of of *x* in an array use *array.index(x)*

Compound Types

Matrices

A matrix is a special case of a vector. Whereas a vector is a one-dimensional object, a matrix is a *n* dimensional object.

A simple two dimensional matrix can be created as shown below:

```
myMatrix = {}
myMatrix[0,0] = 2
myMatrix[0,1] = 3
myMatrix[1,0] = 4
myMatrix[1,1] = 5
print(myMatrix)
```

```
## {(0, 0): 2, (0, 1): 3, (1, 0): 4, (1, 1): 5}
```

Compound Types

Matrices

Matrices can also be created and populated using list comprehension syntax. The example below creates a matrix of width 3 and height 4. All elements in the matrix are initially set to zero. This is effectively a list of lists.

```
myMatrix = [[0 for x in range(3)] for y in range(4)]
myMatrix[0][0] = 5
myMatrix[1][1] = 6
myMatrix[2][2] = 2
print(myMatrix)
```

```
## [[5, 0, 0], [0, 6, 0], [0, 0, 2], [0, 0, 0]]
```

Compound Types

Matrices

Matrices can also be created using the *numpy* library. The first example below uses the *empty()* function to create a 2 by 3 matrix with empty cells. The second example is the same, but the cells are filled with zeros and in the third example they are filled with ones.

```
import numpy
myMatrix = numpy.empty((2, 3))
```

```
import numpy
myMatrix = numpy.zeros((2, 3))
```

```
import numpy
myMatrix = numpy.ones((2, 3))
```

Compound Types

Matrices

Matrices can also be created using *numpy*'s low level constructor, *ndarray()*.

```
myMatrix = numpy.ndarray((2, 3))
```

As you can see, *numpy* is not short on ways to create matrices. In the example below, a matrix is created from a reshaped one dimensional range.

```
myMatrix = numpy.arange(9).reshape((3, 3))
print(myMatrix)
```

```
## [[0 1 2]
##  [3 4 5]
##  [6 7 8]]
```

Compound Types

Matrices

We are not quite finished looking at the multitude of ways matrices can be created with *numpy*. A matrix can be created from a reshaped list. In this example, the matrix is filled with zeros.

```
myMatrix = numpy.array([0] * 6).reshape((2, 3))
```

Finally, a matrix can be created using Matlab syntax.

```
myMatrix = numpy.matrix('9 8; 7 6')
print(myMatrix)
```

```
## [[9 8]
##  [7 6]]
```

Conversion and Coercion

- A **mixed-type expression** is an expression containing operands of different types. The CPU requires that values have the same internal representation scheme before operations can be performed.
- It is necessary for operands in a mixed-type expression to be converted to a common type prior to the operations being performed.
- **Coercion** is the implicit (automatic) conversion of operands to a common type. This can only occur if there is no loss of information.
- **Conversion** is the explicit conversion of operands to a common type. This may result in loss of information and utilises type-conversion functions.

```
2 + int(3.4)  # This will result in 2 + 3 = 5
```

Program flow

Sequence statements

Sequence statements are instructions given in a sequence. They have a begin and an end point. At the top level of the program, the begin point is where the program starts, and the end is where the program terminates.

```
celsiusInput = input("Please enter Celcius value:") # Statement 1
celsiusFloat = float(celsiusInput)                  # Statement 2
fahrResult = ((9/5.0)*celsiusFloat) + 32            # Statement 3
fahrString = format(fahrResult, ".2f")              # Statement 4
message = "The result is: " + fahrString            # Statement 5
print(message)                                      # Statement 6
```

The execution of this code flows from Statement 1 to Statement 6.

Program flow

Conditional statements

Conditional statements allow the program flow to switch between alternative paths. Such statements must involve a conditional expression.

The simple *if* statement

```
if condition:
   indentedStatementBlock
```

If the *conditionalExpression* returns true then the *indentedStatementBlock* is executed otherwise the *indentedStatementBlock* is skipped.

Program flow

Conditional statements

The simple *if* statement - example

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

```
## x is less than 15
```

```
if x<5 :
    print("x is less than 5")
```

Program flow

Conditional statements

The if-else statement

```
if conditionalExpression:
   trueStatementBlock
else:
   falseStatementBlock
```

This extension of the simple if statement will execute the *trueStatementBlock* if the *conditionalExpression* returns true, otherwise the *falseStatementBlock* is executed.

Program flow

Conditional statements

The if-else statement - example

```
x = 3
if x % 2 == 0 :
    print("x is even")
else :
    print("x is odd")
```

```
## x is odd
```

Program flow

Conditional statements

The if-elif-else statement

```
if conditionalExpression1:
   statementBlock1
elif conditionalExpression2:
   statementBlock2
else:
   statementBlock3
```

If *conditionalExpression1* is true then *statementBlock1* is executed, otherwise the flow moves to the *elif* statement. If *conditionalExpression2* returns true then *statementBlock2* is executed, otherwise flow moves to the *else* statement and *statementBlock3* is executed.

Note that there may be multiple *elif* blocks.

Program flow

Conditional statements

The if-elif-else statement - example

```
x = 5
if x < 5 :
    print("x is less than 5")
elif x == 5:
    print("x is equal to 5")
else :
    print("x is greater than 5")
```

```
## x is equal to 5
```

Program flow

Repetition statements

The while statement

```
while conditionalExpression:
   statementBlock
```

The *while* statement results in the *statementBlock* being executed repeatedly while the *conditionalExpression* returns true. See the example below:

```python
i = 4
while i < 9 :
    print(i)
    i = i+2
```

```
## 4
## 6
## 8
```

Program flow

Repetition statements

The for statement

```
for item in sequence:
   statementBlock
```

The *for* statement is a counter-controlled loop. The *statementBlock* will execute once for each item in *sequence*. Note that *item* is a placeholder variable that receives each of the values in the list in turn. See the example below.

```
myList = ["A","B","C"]
for count in myList:
    print(count)
```

```
## A
## B
## C
```