Fundamentals of Computer Programming

Lecture slides - Variables and Types

Intro

- This lesson covers
  - Variables, assignment and Data-types within **Python**
  - Use of the *print* function, along with single, double and triple quotes
  - Getting input from the user
  - Manipulating Strings
  - Introduction to Lists

Variables

- In an earlier lesson we covered input and evaluation of basic *expressions*, e.g.

```
10 * 2 + (4 / 2.5 * 2)
```

- When programming we require values to be stored for later access
- We store the results of expressions, and other types of values using *variables*
- We refer to a variable using an *identifier*
- An *identifier* is basically a *case-sensitive* name that consists of letters, digits and underscores (_) but may not begin with a digit.

Variable Names

- A variable name (identifier) should be meaningful and indicate the nature of the value to be stored, e.g. good variables names may be -

```
age
average_value
highest_score
student_name
```

- The following would **NOT** be allowed as variable names -

```
1_person
&hello
```

Assignment

- A variable value is set using assignment (which is the '=' symbol)

```
age = 23
average_value = (10 + 20 + 30) / 3
highest_score = last_score
student_name = "Jon"
```

- Notice how the right side of the '=' can be an expression, or the name of another variable
- The '=' has a low *precedence*, which means that the right-hand side is evaluated prior to the value been stored in the variable

Updating Variable Values

- As the name “variable” suggests the *value* of a *variable* can change
- Variable values are often changed relative to their own current value, e.g.

```
age = age + 1
```

- When a variable name appears in an *expression*, the current *value* of the variable is used within the calculation
- Since the right hand side is evaluated first, this takes the current value of ‘`age`’, increases it by `1`, then assigns the result back to the ‘`age`’ variable
- Remember the ‘=’ symbol causes an assignment, it is NOT testing whether the left-hand value is equal to the right-hand value

Augmented Assignment

- Assignments which include the same *variable* to the left and right of the *assignment* symbol (=) are common, e.g. `count = count + 1`

- Because of this there are a set of short-cuts available for writing such expressions (inherited from the C programming language)

Assignment expression	*Augmented assignment equivalent*
`count = count + 10`	`count += 10`
`lives = lives - 1`	`lives -=1`
`score = score * 5`	`score *= 5`
`rate = rate / 1.24`	`rate /= 1.24`
`spare = spare % size`	`spare %= size`

Variable Data-Type

- All values are based on a *data-type*, such as '`int`', '`float`', or '`string`'
- Within **Python** a variable's data-type depends on the last value assigned to the variable, i.e. it is *dynamic* in nature
- Many other languages use *static* typing, which ensures each variable can only ever store one specific type of value
- The fact that variables can change *data-type* over their life-time can lead to unexpected run-time errors, so care must be taken when programming
- A `type()` function exists which will tell you the type of a value or variable

Common Data-Types

- Python has a number of built-in types, which are very commonly used. These *primitive-types* are -
  - `int` refers to values that are whole numbers
  - `float` refers to decimal values, i.e. numbers that include a decimal place
  - `bool` is used to store the values of either `True` or `False`
  - `str` refers to a string of characters, e.g. used to store text
- Python also has additional built-in types, that we shall examine later

More on Data-Types

- The type of a variable is often obvious, e.g

```
x = 10
type(x)
int

x = 10.2
type(x)
float

x = "hello"
type(x)
str
```

- Sometimes it is less obvious, e.g

```
x = (10 + 10) / 2
type(x)
float
```

More on Data-Types

- The data-type of a variable can influence how an *operator* is applied within an expression
- A good example of this is to consider what the '+' operator does when applied to a string, e.g.

```
greeting = "hello "
message = greeting + "and welcome"
```

- Since these *operands* are based on a *string* data-type, the '+' performs **concatenation** rather than addition
- It is also possible apply the '*' operator to strings, resulting in the content being repeated a number of times, e.g.

```
police_greeting = greeting * 3
```

Introducing Functions

- A **function** allows us to execute some pre-written code
- We *call* a function by using its *name*, followed by *parentheses* ()
- Within the *parentheses* we pass values, to be used by the function. These are often called **arguments** or **parameters**
- Python provides many built-in functions, and many *libraries* that contain functions
- On the previous slide we used the built-in `type()` function
- This function takes a single *argument*, then tells us the type of that value

Functions

- The number of *arguments* passed within the *parentheses* () depends on the function, and can even vary for the same function
- Passed *arguments* can be *literal* values, *variable* names, *expressions* or even calls to other *functions* (since they return a value)
- We have already seen the built-in `print()` function within an earlier lesson
- This function takes a variable number of *arguments*, e.g.

```
print("The average was", total/count)
```

- A function often **returns** a value, which can be used in expressions or stored in a variable

Getting Input

- We often need to read input from the user. We can use the built-in function `input()` to do this, e.g.

```
name = input("What's your name? ")
```

- The value typed by the user is **returned** as a *string*, and in the above example assigned to the '`name`' variable
- If the input value is to be used as something other than a string, then it needs to be converted, e.g.

```
age = input("What's your age? ")
print("In ten years you will be", int(age) + 10)
```

- This example calls to the function `int()` to convert the string to an integer

Strings: Single and Double Quotations

- A *string* type value can be delimited using single quotes or double, e.g.

```
print('hello, this is a string')
print("hello, this is a string")
```

- This can be helpful if the string includes a quote to be output, e.g.

```
print('hello, my name is "mark"')
```

- Strings can also contain **escape sequences**, to encode special characters
- These are identified using a backslash **** followed by the sequence

Strings: Escape Sequences

\n Insert a newline character in a string. When the string is displayed, for each newline, move the screen cursor to the beginning of the next line.

\t Insert a horizontal tab. When the string is displayed, for each tab, move the screen cursor to the next tab stop.

**** Insert a backslash character in a string.

\" Insert a double quote character in a string.

\' Insert a single quote character in a string.

```
print("A string\n\tShown over multiple \"lines\"")
```

Strings: Triple Quotations

- Strings can also be delimited using triple quotes, e.g.

```
print("""hello, this is a string""")
```

- Can be used for multi-line strings, strings containing both single and double quotes, or **docstrings** (to be discussed later), e.g.

```
triple_quoted_string = """This is a triple-quoted
string that spans two lines"""


print("""Display "hi" and 'bye' in quotes""")
```

Strings: Indexing

- Parts of a string can be accessed via a numerical **index**, placed between square brackets
- This allows individual characters from a string to be accessed, e.g.

```
name = "Black Knight"
first_letter = name[0]
```

- Notice how the **index** is *zero based* (starts at 0)
- Attempting to use an index that is out of range will result in an error, e.g. the following would cause an error -

```
first_letter = name[12]
```

Strings: Negative Indexing

- Unlike many other languages Python allows *-ve* index values to be used
- An index of less than 0 starts counting from the right-side of the string, e.g.

```
name = "Black Knight"
last_letter = name[-1]
```

- Notice how the *-ve* index is NOT *zero based* (since -0 is the same as 0)
- As with positive indices, attempting to use an index that is out of range will result in an error, e.g. the following would cause an error -

```
last_letter = name[-13]
```

Strings: Slicing

- Rather than access a single character it is possible to access several characters from a string, this is called **slicing**
- This allows a *sub-string* to be accessed using two indices, e.g.

```
name = "Roger the Shrubber"
word = name[6:9]        # get the string "the"
```

- The *start* indexed value is *included*, the *end* indexed value is *excluded*
- If either the *start* or *end* index is omitted then these default to be *0* (for the start) or the *length of the string* (for the end)
- As with single character access, *-ve* indexing is also allowed

Strings: Slicing Examples

```
name = "Roger the Shrubber"

word = name[:3]# get the string "Rog"
word = name[6:]# get the string "the Shrubber"
word = name[-6:]   # get the string "rubber"
word = name[:]     # get the string "Roger the Shrubber"
```

- Note: Unlike single character indexing, *out of range* slice indices are handled gracefully, e.g. the following would NOT cause a run-time error -

```
word = name[6:100] # get the string "the Shrubber"
```

Introduction to Lists

- One of the strengths of Python is the built-in support for *compound* data-types, one of the most commonly used is **Lists**
- As the name suggest, a list groups other values together in an *ordered* list type structure
- A list is written as a comma separated list of values, appearing between square brackets, e.g.

```
names = ["Terry", "John", "Michael", "Eric", "Terry", "Graham" ]

scores = [100.2, 65.543, 26.4, 19.8, 25.2, 99.9 ]

primes = [2, 3, 5, 7, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47 ]
```

Lists

- Lists usually always contain elements all of the same type, but can contain values of any type (even other lists), e.g.

```
mixed = [10, 20.25, False, [1,2,3], "Mark"]
```

- Lists (like Strings) are a type of **sequence** within Python, and thus support common features such as *indexing* and *slicing*, e.g.

```
first_prime = primes[0]
last_prime = primes[-1]
mid_primes = primes[4:9]
```

- Lists (and Strings) also support operations such as *concatenation* and *multiplication*, e.g.

```
more_primes = primes + [53, 59, 61]
```

Lists vs Strings

- *Lists* and *Strings* are very similar but have one important difference, which is that Lists are **mutable**, whereas strings are **immutable**
- This basically means an existing list can be changed after it is created, whereas strings can never be modified following creation, e.g.

```
squares = [4, 9, 15, 25]
squares[2] = 16     # change the third element within the list

name = "Park"
name[0] = "M"       # would cause an error, since a string is immutable
```

- The solution would be to create a brand new string, e.g.

```
name = "M" + name[1:]
```

Mutating Lists

- Since Lists are **mutable**, we can update them in several ways
- We can add to the end of an existing list using the append *method*, e.g.

```
primes.append(67)          # add 67 to the end of the 'primes' list
```

- *Beware:* Concatenating of this style creates a **new** list, e.g.

```
primes + [71, 73]          # this does NOT update the 'primes' list
```

- Whereas an *augmented assignment* mutates the existing list, e.g.

```
primes += [71, 73]    # this does update the 'primes' list
```

Mutating Lists

- To mutate a list we can also *assign to slices*, allowing insertion, replacement and removal within any part of a list
- The *slice* to which we refer is replaced by the new elements provided
- Mutating a list in this way can cause the list to either remain the same size, grow in size, or shrink in size, e.g.

```
scores[2:4] = [28.4, 21.8]          # replace two scores (at index 2,3)
scores[0:0] = [38.2, 19.2, 65.2]    # insert three scores at beginning
scores[-2:] = []                    # remove the last two scores
scores[:] = []                      # remove all scores from the list
```

Getting the Length

- Python has useful built-in function called `len()` that can be used to find out how many elements are present within a *string* or *list*, e.g.

```
name = input("Enter your name : ")
print("your name contains", len(name), "characters")
```

- This function is often useful when using indexing and slicing, e.g.

```
names[len(names):] = ["Pete", "Josh"]
```

- The above example would insert two new names at the end of the list
- The `len()` function can be applied to other data-types which contain multiple values, such as *tuples*, *sets*, and *dictionaries* that we will see later

Summary

- We use **variables** to store values within a program, which are based on a **data-type**, such as *integer*, *float,* or *string*
- **Strings** can be specified using *single*, *double* and *triple* quotes
- We can call **functions** such as `print()` to execute predefined code
- We use the `input()` function to get information from the user, this always returns a *string* type value
- **Strings** and **lists** are both a type of *sequence*, that we can *index* and *slice*
- Strings are **immutable** whereas lists are **mutable**