Python for Data Analysis
By: Atefeh Khazaei
Week#2 – Practicals

Python's data types

Python has a special function called **type** which gives the data type of any value, variable, or indeed any expression. First, let's find the type of some int, float, str and bool data values; enter:

```
type(3)
type(-64)
type(53.2173)
type(3.0)
type(123456789123456789)
type("hello")
type("123")
type('123')
type(True)
type("False")
```

Now, let's experiment with some variables:

```
x = 2
type(x)
y = 2.3
type(y)
x = 3.2
type(x)
x = "hello"
type(x)
x = True
type(x)
```

We see that the type of a variable is simply the type of its current value. Notice also, as seen in the case of x, that the type of a variable can change. Finally, let's try some longer (numerical) expressions:

```
x = 10
x + 27
type(x + 27)
x - 3.2
type(x - 3.2)
12 * x
```

```
type(12 * x)
x / 5
type(x / 5)
x / 2.5
type(x / 2.5)
```

Notice here that the type of an expression is simply the type of whatever it evaluates to (i.e. the type of its value). Also, where we have an operation with both an int and a float operand (e.g. as in x - 3.2) the type of the result is a float (the int value is automatically converted into a float value before the operation is performed). Also notice that divisions using / are always float divisions, even if both the operands are integers.

Experiment with the type function until you are satisfied that you understand the concept of a data-type, and how the arithmetic operators work with respect to the types of their operands.

Operators and precedence

The arithmetic operators +, -, *, /, and ** (power) obey the standard mathematical precedence rules. Try to predict the results of the expressions below before entering them:

```
x = 3
y = 4
1 + x * y
(1 + x) * y
3 ** 2
2 * 3 ** 2
15 - 12 / y * x
15 - 12 / (y * x)
```

Type conversions and built-in numeric functions

We can convert between ints, floats and strings as follows:

```
float(3)
int(5.6)
str(77.1)
int("456")
```

Try also:

```
x = 4.8
int(x)
round(x)
str(x)
```

It is important to realise that none of these functions changes the value (or type) of the variable x itself; they just give a new value of a different type. To check this, look at the value of x now; enter:

```
x
```

Integer division and remainder

Enter the following expressions:

```
23 / 5
23 // 5
23 % 5
```

The first expression here is a standard float division. The second is an ***integer division***; and the final one gives the ***remainder*** for this integer division. We see that 5 goes into 23 four times, with remainder 3.

Using the math module

Many useful functions and constants are not built into the Python language, but appear in external "modules" (other files of Python program code). A good example is the math module. To be able to use the facilities in this module, we first need to ***import*** it:

```
import math
```

Now, try entering:

```
math.pi
math.sqrt(2)
math.sqrt(-1)
math.sin(0)
math.sin(math.pi / 2)
math.log(1)
math.log(math.e)
```

Note: ***sqrt*** gives the square root of a number, ***sin*** gives the sine of an angle expressed in radians, and ***log*** gives the natural log of a number. Don't worry too much here if you don't remember much about trigonometric functions!

Warm-up exercise

For some practice using the numerical operators, first type the following lines:

```
x1 = 1
y1 = 2
x2 = 4
y2 = 6
```

to represent two points with coordinates (1;2) and (4;6) in two-dimensional space (Figure 1).

Now, try to compose Python expressions that are equivalent to the following mathematical expressions. The first one gives the ***slope*** of the line that passes through the two points:

$$slope = \frac{y2 - y1}{x2 - x1}$$

The next expression uses Pythagoras' theorem to give the ***distance*** between the two points:

$$distance = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

If you have translated the expressions correctly, they should give values 1.333 and 5.0 respectively.

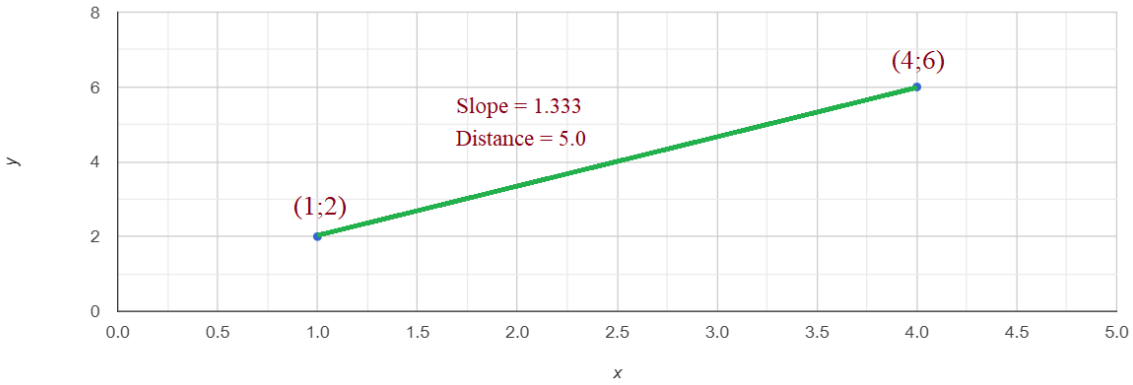

Figure 1: Slope and distance between two points.

Programming exercises

Do the following exercises and save your code. Your codes should include some comments (lines beginning with the # symbol) that explain your method.

1. Write a function circumferenceOfCircle that asks the user for the radius of a circle, and then outputs its circumference. (Use the formula *circumference* = $2\pi r$. For π, use *math.pi* from the *math* module.)

2. Write a function areaOfCircle that asks the user for the radius of a circle, and then outputs its area (using the formula *area* = πr^2).

3. Write a function costOfPizza that asks the user for the diameter (not the radius) of a pizza (in cm), and then outputs the cost of the pizza's ingredients (based on its area) in pence. Assume that the cost of the ingredients is 1.5p per square cm.

4. Write a function travelStatistics which asks the user to input the average speed (in km/hour) and duration (in hours) of a car journey. The function should then output the overall distance travelled (in km), and the amount of fuel used (in liter) assuming a fuel efficiency of 5 km/liter.

5. Write a function sumOfNumbers that outputs the sum of the first *n* positive integers, where *n* is provided by the user. For example, if the user enters 4, the function should output 10 (i.e. 1 + 2 + 3 + 4). (Hint: This function should use a loop.)

6. [harder] Write a function averageOfNumbers which outputs the average of a series of numbers entered by the user. The function should first ask the user how many numbers there are to be inputted.

7. [harder] Write a function selectCoins that asks the user to enter an amount of money (in pence) and then outputs the number of coins of each denomination (from £2 down to 1p) that should be used to make up that amount exactly (using the least possible number of coins). For example, if the input is 292, then the function should report: 1 × £2, 0 × £1, 1 × 50p, 2 × 20p, 0 × 10p, 0 × 5p, 1 × 2p, 0 × 1p. (Hint: use integer division and remainder).

Simple Boolean expressions

Any decision a program makes is based on testing whether a certain ***condition*** is true or false. Conditions are examples of Boolean expressions (i.e. expressions of the Boolean data type). Let's investigate this type by experimenting with some simple Boolean expressions using Jupyter notebook:

```
True
False
type(True)
type(False)
x = 19
y = 7
x > 15
x < y
x <= 19
x <= 18
x + y <= 30
x == 19
45 != x
17 != x + 2
type(x > 19)
name = "Tim"
name == "Sam"
"Tom" != name
name < "Tom"
name < "Sam"
len(name) == 3
name[0] == "T"
```

Keep experimenting until you are sure you understand how Boolean expressions like these work, and you are aware of the meanings of the operators ==, !=, <, >, <=, and >=.

If statements

Enter the following function that includes an ***if*** statement:

```
def greet(name):
    print("Hello", name + ".")
    if len(name) > 20:
        print("That's a long name!")
```

Now try calling the function a few times with names of different lengths; e.g.:

```
greet("Sam Smith")
```

```
greet("Charles Philip Arthur George Jones")
```

Ensure you understand what is happening with each function call before moving on.

If-else statements

Enter the following function definition that includes an ***if-else*** statement:

```
def canYouVote(age):
    if age >= 18:
        print("You can vote")
    else:
        print("Sorry, you can't vote")
```

Now, invoke the function a few times with different numerical arguments:

```
canYouVote(23)
canYouVote(16)
canYouVote(18)
```

Again, ensure you understand what is happening here before moving on.

If-elif-else statements

Let's now define a function that includes an ***if-elif-else*** statement and that returns a value to the caller:

```
def getDegreeClass(mark):
    if mark >= 70:
        return "1st"
    elif mark >= 60:
        return "2i"
    elif mark >= 50:
        return "2ii"
    elif mark >= 40:
        return "3rd"
    else:
        return "Fail"
```

Now enter:

```
print("Your degree is", getDegreeClass(78))
print("Your degree is", getDegreeClass(45))
```

Repeat the call with different argument values until you understand the operation of the if-elif-else statement.

Using and & or

We can combine two Boolean expressions using the Boolean operators ***and*** and ***or***. For now, let's see briefly some examples of their use:

```
x = 8
y = 3
x > 5 and y < 4
x == 4 and y < 4
x == 4 or y < 4
x == 4 or y > 5
```

Experiment with and & or for a while to try to fully understand their meanings.

Nested decisions and design issues

Copy the file **dates.py** from Moodle into your own file-space. This file contains some function definitions. The first one, ***isLeapYear***, takes a year as a parameter and returns True or False depending on whether the year is a leap year or not. (Years divisible by 4 are leap years, except those that are divisible by 100. However, years that are divisible by 400 are also leap years. So, 2000 and 2004 were both leap years, but 2009 and 2100 are not.)

Study the function definition to see how it works. Notice that it uses the % (remainder) operator to check if one value is divisible by another (e.g. year % 4 == 0 is true if year is divisible by 4). Check that you understand the logic of the function. Test the function using the following calls:

```
isLeapYear(2000)
isLeapYear(2004)
isLeapYear(2009)
isLeapYear(2100)
```

Next in the dates.py file is a function (***daysInMonth***) that give the number of days in a given month of a given year (where the month is expressed as a number between 1 and 12). Notice that this functions call the *isLeapYear* function when month is 2 (February) to find out if the given year is a leap year. *daysInMonth* considers the largest group of months (those that have 31 days) last so that they don't need to appear within a condition (they are dealt with in the else clause.

This example is intended to illustrate that you should think carefully when designing decision structures. There are often several solutions to any given problem, and one solution is often simpler (better) than the others. The *daysInMonth* function could be re-written in a better way using lists, which we'll study later in the unit.

Simple For loops

Let's begin by reviewing how simple for loops work from earlier on in the unit. Most for loops use the built-in function *range*, so let's remind ourselves of what this does:

```
range(10)
list(range(10))
len(range(10))
x = 5
range(x + 2)
list(range(x + 2))
len(range(x + 2))
```

We see that, given an integer argument *n*, range gives as an object of type range. A range object represents a range of values; in order to see all the values we need to pass this object to list. We see that the numbers within the range, and therefore in the resulting list, start with 0 and end with n-1. (Note that, assuming *n* is non-negative, the length of range(n) is n.) Now, type the following:

```
for i in range(10):
    print("Hello")
for i in range(10):
    print(i)
```

(Note: If you use shell, you will need to press return (enter) twice at the end of each loop to tell the shell that you have finished.)

The first example shows us that the body of the loop (here, just a print statement) is executed the same number of times (10) as there are elements in range(10).

The second example shows that the variable i takes the value of each element of the list in turn, for each execution of the loop body. This can be seen even more clearly in the following for loops that do not use range. The first one loops through the elements of a list that has not been made using range:

```
for i in [1, 9, 7, 8]:
    print(i)
```

The next one loops through the elements of a string:

```
for ch in "Hello There":
    print(ch)
```

Note that each of these examples introduces a new loop variable (here i and ch). There is nothing special about these variables or their names (loop variable names should be chosen using the same criteria as any other variable). It is, however, important that you do not change the value of a loop variable within the body of a loop, as in:

```
for i in range(5):
    i = i + 1
    print(i)
```

This leads to code that is difficult to understand (even though it might not actually cause an error).

Using extra arguments to range

If we want to count from 1 to 10 using a for loop and range, we might type:

```
for i in range(10):
    print(i + 1)
```

This code can be simplified slightly by using a second argument to range. Try:

```
list(range(1, 11))
list(range(-4, 20))
list(range(6, 2))
```

We see that range(m, n) gives a list that starts with m and ends at n-1. We can thus replace the above for loop by the slightly simpler:

```
for i in range(1, 11):
    print(i)
```

Finally, range can take a third argument. Try the following:

```
list(range(1, 10, 2))
list(range(4, 22, 3))
list(range(10, 0, -1))
```

We see that range(m, n, s) gives a range that starts with *m*, ends one value short of *n*, with a step of *s*. The final example shows how we can build a list with numbers in descending order. Try entering, and then invoking the following function:

```
def countdown():
    for i in range(10, 0, -1):
        print(i, "...", end=" ")
    print("Blast Off!")
```

Experiment using range with different numbers of parameters until you are sure that you fully understand what it does and can predict its behaviour.

Nested for loops

Loops can be ***nested*** (appear one within another), as the following example demonstrates:

```
for i in range(3):
    print("Outer loop with i =", i)
    for j in range(4):
        print("Inner loop with i =", i, "and j =", j)
    print()
```

Notice here that since the inner loop is part of the body of the outer loop, it is executed three times. (So the body of the inner loop is executed in total 3 × 4 = 12 times). Note that it is essential to choose different names for the loop variables of nested loops. Try now entering the following function definition that displays a “numbered” triangle:

```
def triangle(n):
    for i in range(1, n + 1):
        for j in range(1, i + 1):
            print(j, end=" ")
        print()
```

and invoke it with, for example:

```
triangle(4)
```

Try to fully understand how this function works. In particular, make sure you can answer the following questions:

Which loop provides the “numbers” for the lines of the triangle?

Which loop displays each line of text in the triangle?

What part of the code ensures that the lines are of different lengths?

What role does the end=" " play after the first print statement?

What role does the second print play, and is this statement part of the inner or the outer loop?

Programming exercises

Solutions to all of the exercises should use *if* statements, *for* loops or both. Try to ensure that your use of these structures is not only correct, but is as simple as possible.

1. A fast-food company charges £1.50 for delivery to your home, except for large orders of £10 or more where delivery is free. Write a function fastFoodOrderPrice that asks the user for the basic price of an order and prints a "The total price of your order is . . . " message containing the total price including any delivery charges. (Try to avoid using an else clause in your solution.)

2. Write a whatToDoToday function that asks the user to enter today's temperature, and then prints a message suggesting what to do. For temperatures of above 25 degrees, a swim in the sea should be suggested; for temperatures between 10 and 25 degrees (inclusive), shopping in Gunwharf Quays is a good idea, and for temperatures of below 10 degrees it's best to watch a film at home.

3. Write a function displaySquareRoots that has two parameters, start and end, and displays the square roots of numbers between these two values, shown to three decimal places. For example, the call displaySquareRoots(2, 4) should result in the following output:

The square root of 2 is 1.414

The square root of 3 is 1.732

The square root of 4 is 2.000

4. A school teacher marks her pupils' coursework out of 20, but needs to translate these marks to a grade of A, B, C or F (where marks of 16 or above get an A, marks between 12 and 15 result in a B, marks between 8 and 11 give a C, and marks below 8 get an F). Write a calculateGrade function that takes a mark as a parameter and ***returns*** the corresponding grade as a single-letter string. If the parameter value is too big or too small, the function should return a mark of X.

5. A train company prices tickets based on journey distance: tickets cost £3 plus 15p for each kilometre (e.g. a ticket for a 100 kilometre journey costs £18). However, senior citizens (people who are 60 or over) and children (15 or under) get a 40% discount. Write a ticketPrice function that takes the journey distance and passenger age as parameters (both integers), and ***returns*** the price of the ticket in pounds (i.e. a float)