TASK

The String and Numerical Data Types

Visit our website

Introduction

WELCOME TO THE STRING AND NUMERICAL DATA TYPES TASK!

In this task you'll start to learn about data types. The two most fundamental categories are strings and numeric data types.

Strings are some of the most important and useful data types in programming. Why? Let's think about it like this. When you were born, your parents did not immediately teach you to do mathematical sums, not 1 + 1 or standard deviation. The first thing they taught you to do was to speak, to say words, to construct full sentences, to say "Mom" or "Dad". Well, this is why we are going to 'teach' the computer to first be able to communicate with the user - and the only way to do this is to have a good grasp of strings.

Next, we consider numbers. You are already very familiar with numbers as we encounter them daily and, in most cases, multiple times a day. It is likely that, through studying mathematics at school, you were introduced to different types of numbers such as integers and decimal numbers, as well as the operations we can perform on them, such as addition, subtraction, multiplication, and division. Computer programming languages like Python provide support for storing and manipulating many different types of numbers. In this task, you will learn how numbers are categorised based on their nature, as well as how to perform arithmetic operations in Python.

WHAT ARE STRINGS?

A *string* is a list of letters, numerals, symbols and special characters that are put together. The values that we can store in a string are vast. An example of what strings can store is the surname, name, address, etc. of a person.

In Python, strings must be written within "quotation marks" for the computer to be able to read them.

The smallest possible string contains zero characters and is called an *empty string* (i.e. **`string = ""`**).

Examples of strings:

```
name = "Linda"

song = "The Bird Song",

licence_plate = "CTA 456 GP"
```

Strings are probably the most important data type in programming. They are used as a medium of communication between the computer and the user. The user can enter information as a string and the program can use the data to perform calculations and finally display the calculated answer to the user.

```
name = "John"
joke = "Knock, knock, Who's there?"
```

You can use any name for your variable but the actual string you are assigning to the variable must be within “” (quotation marks).

Defining Multi-Line Strings

Sometimes, it's useful to have long strings that can go over one line. We use triple single quotes (‘’’ ‘’’) to define a multi-line string. Defining a multi-line string preserves the formatting of the string.

For example:

```
long_string = ''' This is a long string
 using triple quotes preserves everything inside it as a string
 even on different lines and with different \n spacing. '''
```

STRING FORMATTING

Strings can be added to one another. In the past we used to use a method called **concatenation**, which looked like this:

```
name = "Peter"
surname = "Parker"
full_name = name + surname
```

`full_name` will now store the value “PeterParker”.

The `+` symbol simply joins the strings. If you wanted to make your code more presentable, you could put spaces between the words.

```
full_name = name + " " + surname
```

We now added a blank space in between the two strings, so full_name will now store the value “Peter Parker”. **Note: you cannot concatenate a string and a non-string**. You need to cast the non-string to a string if you want to concatenate

it with another string value. If you try to run code that adds a string and a non-string, you will get an error. For example, if we wanted to add an age of 32 we would have to cast it as a string to print it.

```
print(full_name + str(32))
```

But this is a clunky way of formatting strings. This way of putting together a string is still used in older languages, such as Java, and does have its place, but it is much better practice to use the `format()` method.

```
name = "Peter Parker"
age = 32
sentence = "My name is {} and I'm {} years old.".format(name, age)
print(sentence)
```

In the example above, a set of opening and closing curly braces (`{}`) serve as a placeholder for variables. The variables that will be put into those placeholders are listed in the brackets after the keyword `format`. The variables will fill in the placeholders in the order in which they are listed. Therefore, the code above will result in the following output: `My name is Peter Parker and I'm 32 years old.`

Notice that you don't have to cast a variable that contains a number (`age`) to a string when you use the *format* method.

f-Strings

The shorthand for the format function is *f-strings*. Take a look at the example below:

```
name = "Peter Parker"
age = 32
sentence = f"My name is {name} and I'm {age} years old."
print(sentence)
```

In f-strings, instead of writing `.format()` with the variables at the end, we write an `f` before the string and put the variable names within the curly brackets. This is a neat and concise way of formatting strings.

NUMBERS AS STRINGS

We can even store numbers as strings. When we are storing a number (i.e. 9, 10, 231) as a string, we are essentially storing it as a word. The number will lose all its

number-defining characteristics. So the number will not be able to be used in any calculations. All you can do with it is read or display it.

In real-life sometimes we don't need numbers to do calculations, we just need them for information purposes. For example, the house number you live in (let's say: 45 2nd Street UK 2093) won't be used for performing any calculations. What benefit would there be for us to find out what the sum is of all the house numbers in an area? The only thing we need is to know what number the house is when visiting or delivering a package. The number just needs to be visible. This is the same concept when storing a number as a string; all we want is to be able to take in a value and display it to the user.

Example:

```
telephone_num = "041 123 1234"
```

THINGS YOU CAN DO TO STRINGS

You can use `len()` to get the number of characters in a string or length of a string. The print statement below prints out 12, because "Hello world!" is 12 characters long, including punctuation and spaces.

```
print(len("Hello World!"))
```

Output:

```
12
```

You can also *slice* a string. Slicing in Python extracts characters from a string, based on a starting index and ending index. It enables you to extract more than one character or "chunk" of characters from a string. The print statement below will print out a piece of the string. It will start at position/index 1, and end at position/index 4 (which is not included).

```
greeting = "Hello"
print(greeting[1:4])
```

Output:

```
ell
```

You can even put negative numbers inside the brackets. The characters are also indexed from right to left using negative numbers, where **-1** is the rightmost index

and so on. Using negative indices is an easy way of starting at the end of the string instead of the beginning. This way, `-3` means "3rd character from the end".

Look at the example below. The string is printed from the first index, 'e', all the way to the end of the string. Notice that you do not need to specify the end of the index range:

```
greeting = "Hello"
print(greeting[1:])
```

Output:

```
ello
```

In the example below, the slice begins from position `0` and goes up to but not including position `1`:

```
greeting = "Hello"
print(greeting[:1])
```

Output:

```
H
```

In the next example, the slice begins from position `1`, includes positions `1` and `2`, and then continues to the end of the string and skips/steps over every other position. This is known as an extended slice. The syntax for an extended slice is **[begin : end : step]**. If the end is left out, the slice continues to the end of the string.

```
greeting = "Hello"
print(greeting[1::2])
```

Output:

```
el
```

In this final example, you can think of the '`-1`' as a reverse order argument. The slice begins from position `4`, continues to position `1` (not included) and skips/steps backwards one position at a time:

```
greeting = "Hello"
print(greeting[4:1:-1])
```

Output:

```
oll
```

You can print a string in reverse by using `[::-1]`. Remember that the syntax for an extended slice is [begin : end : step]. By not including a beginning and end and specifying a step of -1, the slice will cover the entire string, backwards, so the string will be reversed. You can find out more about extended slices **here**.

Note that slicing a string does not modify the original string. You can capture a slice from one variable in a separate variable. Try typing the following into the interactive shell:

```
new_string = "Hello world!"
fizz = new_string[0:5]
print(fizz)
print(new_string)
```

By slicing and storing the resulting substring in another variable, you can have both the whole string and the substring handy for quick, easy access.

USING METHODS TO MANIPULATE STRINGS

Most programming languages provide built-in functions to manipulate strings, i.e., you can concatenate strings, you can search from a string, you can extract substrings from a string, etc. There is more to be learned about this topic, but for now we will stop here. If you're very keen to look ahead, you can find a number of resources online to **learn about string methods**.

Now, let's move on to learning about numeric data types.

NUMBERS IN PYTHON

There are three distinct numeric data types in Python: integers, floating-point numbers, and complex numbers.

- **Integers**: these are synonymous with whole numbers. Numbers which are stored as this type do not contain a fractional part or decimal. Integers can either be positive or negative and are normally used for counting or simple calculations. For example, you can store the number of items you wish to purchase from a store as an integer, e.g. num = int("-12").

- **Floats**: decimal numbers or numbers which contain a fractional component are stored as floats. They are useful when you need more precision, for example, when storing measurements for a building or amounts of money. Floats may also be in scientific notation, with E or e indicating the power of 10, e.g. x = float("15.20"), y = float("-32.54e100")

- **Complex**: complex numbers have a real and imaginary part, which are each a floating-point number, e.g. c = complex("45.j").

DECLARING NUMBERS IN PYTHON

When you declare a variable, Python will already know if it is a *float* or an *integer* based on its characteristics. If you use decimals, it will automatically be a *float* and if there are no decimals, then it will be an *integer*.

```
class_list = 25    # integer
interest_rate = 12.23    # float
```

CASTING BETWEEN NUMERIC DATA TYPES

To cast between numbers, make use of the `int()` or `float()` functions, depending on which is needed.

```
num1 = 12
num2 = 99.99
print(float(num1))
# Converting floats to ints, as below, causes data loss. int() removes values
# after the decimal point, returning only a whole number.
print(int(num2))
```

ARITHMETIC OPERATIONS

Doing calculations with numbers in Python works similarly to the way they would in normal maths.

```
sum = 2 + 4
print(sum)
# prints out 6
```

```
cents = 0.25 + 4.33
print(cents)
# prints out 4.58
```

The only difference between calculations in real mathematics and programming is the symbols you use:

Arithmetic Operations	Symbol used in Python
Addition: adds values on either side of the operator	+
Subtraction: subtracts the value on the right of the operator from the value on the left	-
Multiplication: multiplies values on either side of the operator	*
Division: divides the value on the left of the operator by the value on the right	/
Modulus: divides the value on the left of the operator by the value on the right and returns the remainder	%
Exponent: performs an exponential calculation, i.e. calculates the answer of the value on the left to the power of the value on the right	**

MATHEMATICAL FUNCTIONS

In Python, numerical data types play a crucial role in performing mathematical operations and calculations. To work with numerical data effectively, Python offers a variety of built-in functions (pre-written code) and libraries that can be utilised. Built-in functions are readily available within Python and provide fundamental mathematical operations, while the `math` module needs to be imported and offers an extensive collection of specialised mathematical functions.

Some of the most commonly used functions are listed in the tables below.

Built-in mathematical functions:

Function	Description	Example

`round()`	Rounds a floating-point number to the nearest whole number, or decimal places as specified by the second argument.	`number = 66.6564544`
		`print(round(number,2))`
		will output `66.66`

`min()`	Returns the smallest value from an iterable, such as a list or tuple.	`numbers_list = [6,4,66,35,1]`
		`print(min(numbers_list))`
		will output `1`

`max()`	Returns the largest value from an iterable, such as a list or tuple.	`numbers_list = [6,4,66,35,1]`
		`print(max(numbers_list))`
		will output `66`

`sum()`	Calculates the total sum of all elements in iterable, such as a list or tuple.	`numbers_list = [6,4,66,35,1]`
		`print(sum(numbers_list))`
		will output `112`

Mathematical functions available through the `math` module:

To use the functions in the `math` module, add this line of code to the top of your program:

```
import math
```

Function	Description	Example
`math.floor()`	Rounds a number down	`print(math.floor(30.3333))` will output `30.0`
`math.ceil()`	Rounds a number up	`print(math.ceil(30.3333))` will output `31.0`
`math.trunc()`	Cuts off the decimal part of the float	`print(math.trunc(30.33333))` will output `30`
`math.sqrt()`	Finds the square root of a number	`print(math.sqrt(4))` will output `2.0`
`math.pi()`	Returns the value for pi where pi is the number used to calculate the area of a circle	`print(math.pi)` will output `3.141592653589793`

Feel free to explore some more functions in the `math` module **here**.

Instructions

Before you get started, we strongly suggest you use an editor such as VS Code to open all text files (.txt) and Python files (.py).

First, read the accompanying Python example files. These examples should help you understand some simple Python. You may run the examples to see the output. Feel free to also write and run your own example code before doing the practical tasks to become more comfortable with Python.

Practical Task 1

Follow these steps:

- Create a new Python file called **replace.py**.
- Save the sentence: “The!quick!brown!fox!jumps!over!the!lazy!dog.” as a single string.
- Reprint this sentence as “The quick brown fox jumps over the lazy dog.” using the `replace()` function to replace every “!” exclamation mark with a blank space.
- Reprint that sentence as: “THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.” using the `upper()` function
- Print the sentence in reverse. (Hint: review what you learned about slicing!)

Practical Task 2

Follow these steps:

- Create a new Python file in the Dropbox folder for this task, and call it **manipulation.py**.
- Ask the user to enter a sentence using the `input()` method. Save the user's response in a variable called `str_manip`.
- Using this string value, write the code to do the following:
    - Calculate and display the length of `str_manip`.
    - Find the last letter in `str_manip` sentence. Replace every occurrence of this letter in `str_manip` with '@'.
        - e.g. if `str_manip` = "This is a bunch of words", the output would be: "Thi@ i@ a bunch of word@"
    - Print the last 3 characters in `str_manip` backwards.
        - e.g. if `str_manip` = "This is a bunch of words", the output would be: "sdr".
    - Create a five-letter word that is made up of the first three characters and the last two characters in `str_manip`.
        - e.g. if `str_manip` = "**Thi**s is a bunch of wor**ds**", the output would be: "Thids".

Practical Task 3

Follow these steps:

- Create a new Python file called **numbers.py**.
- Ask the user to enter three different integers.
- Then print out:
    - The sum of all the numbers
    - The first number minus the second number
    - The third number multiplied by the first number
    - The sum of all three numbers divided by the third number

Challenge 1

Use this opportunity to extend yourself by completing an optional challenge activity.

Let's try some slightly more complex maths. Follow these steps:

- Create a new Python file in the Dropbox folder for this task, and call it **challenge_1.py**.
- Ask the user to enter the lengths of all three sides of a triangle.
- Calculate the area of the triangle.
- Print out the area.
- Hints:
  - If `side1`, `side2` and `side3` are the sides of the triangle:
    - `s = (side1 + side2 + side3)/2` and
    - `area = √(s(s-a)*(s-b)*(s-c))`
  - You'll need to be able to calculate the square root (**this may help**)

Challenge 2

Use this opportunity to extend yourself by completing an optional challenge activity.

Let's practise some casting. Follow these steps:

- Create a new file called **challenge_2.py**.
- Write Python code to take the name of a user's favourite restaurant and store it in a variable called `string_fav`.
- Below this, write a statement to take in the user's favourite number. Use casting to store it in an integer variable called `int_fav`.
- Print out both of these using two separate print statements below what you have written. Be careful!
- Once this is working, try to cast `string_fav` to an integer. What happens? Add a comment in your code to explain why this is.

Things to look out for:

- Make sure that you have installed and set up all programs correctly. You have set up Dropbox correctly if you are reading this, but it is still possible to encounter problems with Python or your editor if these are not installed correctly.
- If you are not using Windows, please ask a reviewer for alternative instructions.

Rate us

Share your thoughts

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.