Python Basics Reference Manual

Data Types 2
Variables 2
Operators 3
Useful Functions 3
Module Imports 4
Printing to the Console 4
Getting input from the user 5
`if` statements 5
`while` loops 6
Looping through lists 6
Looping through ranges of numbers 6
Working with Strings 7
Working with Lists 10
Creating and Calling Functions 11
Working With Objects 13

DATA TYPES

Common data types in Python include to following:

Type	Description	Example	Expression Evaluating to Type
int	Integer Number	`4`	`x = 3 + 5  # sets `x` to 8`
float	Decimal Number	`3.23`	`x = 5 / 2  # sets `x` to 2.5`
boolean	True or False	`True`	`x = 4 < 3  # sets `x` to False`
string	Sequence of characters (enclosed in quotes)	`"Hello"`	
list	Sequence of any data type (enclosed in square brackets)	`[2, 4, 3]`	

You can check the data type of a variable with the line:

```
print(type(my_variable))
```

VARIABLES

Rules for naming variables:

1. Only letters, numbers and underscores (_) are allowed
2. A variable name cannot start with a number

Conventions you should follow (not enforced, but considered good style):

- Start variable names with a lowercase letter
- Use underscores for multiple words in variables: **`volume_of_cube`**
- Variables that store numerical or string data should be adjectives or something descriptive:
  - **`width = 5`**
  - **`color = "green"`**
- Variables that store Boolean data should start with (or at least contain) ***is*** or ***has*** or something else that indicates it represents True/False:
  - **`has_wings`** (maybe describes an insect)
  - **`switch_is_on`**
- Lists names should be plural nouns:
  - **`fruits = ["apples", "pears", "grapes"]`**
  - **`ages = [33, 21, 13]`**

Examples of setting and manipulating values of variables

```
name = "Frank"      # Sets `name` to Frank
age = 17            # Sets `age` to 17
age += 1            # Increases `age` by one
age = age + 1       # Also increases `age` by one
age -= 5            # Decreases `age` by five
```

OPERATORS

Arithmetic			Boolean	
+	Plus		==	Equal to
-	Minus		!=	Not equal to
*	Times		<	Less than
/	Divided by		>	Greater than
**	To the power of		<=	Less than or equal to
//	Divided by, and round down (integer division)		>=	Greater than or equal to
%	Mod (find remainder after integer division)			

USEFUL FUNCTIONS

`abs(x)`	Returns absolute value of number x
`float(x)`	Returns decimal number representation of string x
`int(x)`	Returns integer representation of string x
`len(x)`	Returns the length of string or list x
`max(…)`	Returns the maximum value in the provided set
`min(…)`	Returns the minimum value in the provided set
`round(x, y)`	Rounds x to y decimals places. If y is omitted, it rounds x to the nearest integer.
`type(x)`	Returns the type of variable x
`math.ceil(x)`	Rounds x up to the next integer
`math.floor(x)`	Rounds x down to the previous integer
`math.sqrt(x)`	Returns the square root of x
`sys.exit()`	Exits the program
`random.randint(x, y)`	Returns a random integer between x and y
`random.choice(x)`	Returns a random element from list x
`random.uniform(x, y)`	Returns a random decimal number between x and y

Any of the above function names that contain a dot in the name require the appropriate module to be imported, as described in the next section.

MODULE IMPORTS

To use functions (or variables, classes, etc) from external modules, we must import them. There are a couple common ways to do this.

Import Method 1

Import the full module, then access functions using the syntax **`module_name.function_name(…)`**.

```
import math
print(math.sqrt(49))      # Prints 7
```

Import Method 2

Import specific functions from the module, then access them using just the function name.

```
from math import sqrt
print(sqrt(25))           # Prints 5
```

PRINTING TO THE CONSOLE

The **`print`** function accepts an unlimited number of arguments, and will print each of them on one line separated by spaces. Each new **`print`** statement begins printing on a new line.

```
name = "Bill"
age = 34
print("Name is", name, "and age is", age)
```

To change the behaviour, you can add either or both of these keyword arguments:

- **`sep="separator"`** to replace the space between each item by a custom string
- **`end="end"`** to replace the new line at the end of each print statement with a custom string

Example

```
print("ABC", 2, 3.4, sep="|", end=" >> ")
print("Next print statement")
```

will output:

```
ABC|2|3.4 >> Next print statement
```

GETTING INPUT FROM THE USER

To get **string** data:

```
name = input("What is your name?")
```

To get **integer** data:

```
age = int(input("How old are you?"))
```

To get **decimal** data:

```
cost = float(input("Enter the cost in dollars and cents"))
```

By wrapping the `input` *statement with* `int(…)` *or* `float(…)`*, we convert the reply received to that specific numerical data type. This allows us to us the data received in mathematical expressions.*

`if` STATEMENTS

The basic `if` statement:

```
if age >= 18:
    print("You are a legal adult")
```

Using `if` with `else`:

```
if age >= 18:
    print("You are a legal adult")
else:
    print("You are not yet a legal adult")
```

Checking for multiple possibilities with `elif`:

```
reply = input("Will you have homework today?")

if reply == "yes":
    print("Too bad!")
elif reply == "no":
    print("Enjoy some free time!")
elif reply == "maybe":
    print("Hopefully not!")
else:
    print("I didn't understand your answer.")
```

Using `and` and `or`:

```
if num >= 10 and num <= 20:
    print("The number is between 10 and 20")
if color == "red" or color == "blue" or color == "gold":
    print("You chose an Eastview colour")
```

`while` LOOPS

```
guess = ""

while guess != "banana":
    guess = input("Guess the secret word:")

print("Correct, the secret word was banana!")
```

LOOPING THROUGH LISTS

```
numbers = [3, 5, 8]
for number in numbers:
    print(number, "times 10 equals", number*10)
```

Where possible, it's considered good practice to give the list a plural name representing the data it contains, and then to use the singular form to reference individual items within the loop. This is done in the example above with the variable named `numbers` for the list and `number` for each item.

LOOPING THROUGH RANGES OF NUMBERS

The `range` function will accept either 1, 2 or 3 arguments.

When provided with one argument, *end*, the range will contain the values from 0 to *end-1*.

Example: Prints the numbers 0 through 9

```
for x in range(10):
    print(x)
```

When provided with two arguments, *start* and *end*, the range will contain the values from *start* to *end-1*.

Example: Prints the numbers 5 through 9

```
for x in range(5, 10):
    print(x)
```

When provided with three arguments, *start*, *end* and *step*, the range will begin at *start*, increment by *step*, and will stop at the last value that is less than or equal to *end*.

Example: Prints 2, 5, 8, 11

```
for x in range(2, 14, 3):
    print(x)
```

WORKING WITH STRINGS

Strings can be enclosed in either single quotes or double quotes:

```
str1 = "I am a string"
str2 = 'I am also a string'
```

Use whatever style you prefer.

Triple quotes can be used for multiline strings:

```
limerick = """There once was a fellow from Perth
who was born on the day of his birth
He was married, they say
on his wife's wedding day
and he died on his last day on Earth"""
```

We can join (or more formally, **concatenate**) strings using the `+` operator:

```
name = input("Enter your name:")
greeting = "Good morning, "+name
```

A Python **f-string** can be used to more easily insert data into a string. The f-string must have the letter `f` prior to the starting quotes, which will allow data to be inserted within curly braces:

```
name = input("Enter your name:")
age = int(input("Enter your age"))
greeting = f"Good morning {name}. In five years, you'll be {age+5} years old."

# Or, more commonly, print the f-string directly:
print(f"Good morning {name}. In five years, you'll be {age+5} years old.")
```

The data inside the curly braces can contain variables, function calls, expressions, etc.

Extracting Substrings

Each character in a string is assigned a position (or more formally, **index**) starting at 0. Suppose we had the code:

```
greeting = "Hi there!"
```

The `greeting` variable looks like this:

Character	H	i		t	h	e	r	e	!
Index	0	1	2	3	4	5	6	7	8

To extract a single character from the string, put the desired index into square brackets:

```
print(greeting[4])        # prints h

# The square brackets can also include variables or expressions

i = int(input("Enter the index you want to extract"))
print(greeting[i])        # prints the character at the index they chose
print(greeting[i+1])      # prints the character after the index they chose
```

To extract a range of character from a string, we put inside square brackets:

- The starting index
- Then a colon (`:`)
- Then one number past the ending index

For example, `greeting[3:8]` gives extracts the substring `"there"`.

Omitting the starting index will default to the beginning of the string, and omitting the ending index will default to the end of the string.

```
print(greeting[4:])       # prints "here!"
print(greeting[:5])       # prints "Hi th"
```

Use negative numbers to start counting from the end of the string:

```
print(greeting[:-1])      # prints "Hi there" (removes the !)
```

Special Characters (called "Escape Characters")

- To insert a new line character into a string, use the code `\n`
- To insert a quote into a string, use the code `\"` or `\'`

```
print("He only paused a moment when\nhe heard him holler, \"stop\"!")
```

The output would be:

```
He only paused a moment when
he heard him holler, "stop"!
```

Using string methods

Strings are objects which have built-in methods that return either variations of the string or information about the string. Here are some of the more useful ones:

Method	Description
`capitalize()`	Returns the string with the first character converted to upper case
`count(text)`	Returns the number of times **`text`** occurs in a string
`find(text)`	Searches the string for **`text`** and returns the position of where it was found
`isalpha()`	Returns **`True`** if all characters in the string are in the alphabet
`isdigit()`	Returns **`True`** if all characters in the string are digits
`lower()`	Returns the string with all characters converted to lower case
`replace(old, new)`	Returns the string with all occurrences of **`old`** replaced with **`new`**
`split(seperator)`	Splits the string at the specified separator, and returns a list
`strip()`	Removes whitespace from the start and end of the string, and returns the result.
`title()`	Returns the string with each word to upper case
`upper()`	Returns the string with all characters converted to upper case

Some usage examples:

```
greeting = "Hi there!"
print(greeting.upper())                  # HI THERE!
print(greeting.title())                  # Hi There!
print(greeting.find("r"))                # 1
print(greeting.replace("Hi", "Hello"))   # Hello there!

if greeting.isalpha():
    print("All characters are in the alphabet")
```

Here's a nice way to check if the user typed the word "yes", possibly with some capitalization, and possibly with spaces before or after the answer:

```
answer = input("Do you want to proceed?").lower().strip()

if answer == "yes":
    # Do something
```

Getting the Length of a String

Use the **`len`** function:

```
print(f"The length of {my_string} is {len(my_string)}")
```

WORKING WITH LISTS

List indexing, including the extraction of individual items and ranges of items, works exactly the same as strings. Please see that section on page 8.

Using list methods

Lists are objects which have built-in methods that return either variations of the list or information about the list. Here are some of the more useful ones:

Method	Description
`append(item)`	Adds the element **`item`** to the end of the list
`clear()`	Removes all the elements from the list
`copy()`	Returns a copy of the list
`count(x)`	Returns the number of elements with the value **`x`**
`extend(list2)`	Adds the elements of a **`list2`** to the end of the current list
`index(x)`	Returns the index of the first element with the value **`x`**
`insert(i, item)`	Adds the element **`item`** at index **`i`**
`pop(i)`	Removes the element at index **`i`**
`remove(item)`	Removes the first occurrence of the value **`item`**
`reverse()`	Reverses the order of the list
`sort()`	Sorts the list

Some usage examples:

```
my_list = [2, 6, 5]
my_list.append(6)          # my_list becomes [2, 6, 5, 6]
print(my_list.count(6))    # prints 2
my_list.extend([4, 8])     # my_list becomes [2, 6, 5, 6, 4, 8]
my_list.remove(6)          # my_list becomes [2, 5, 6, 4, 8]
my_list.reverse()          # my_list becomes [8, 4, 6, 5, 2]
my_list.sort()             # my_list becomes [2, 4, 5, 6, 8]
```

Like strings, the **`len`** function can be used to get the length of the list:

```
print(f"The length of {my_list} is {len(my_list)}")
```

CREATING AND CALLING FUNCTIONS

Here is an example of creating a function:

```
def get_volume_of_box(length, width, height):
    return length * width * height
```

This function expects three arguments, which will go into the variables **`length`**, **`width`**, and **`height`**. These values are then used to calculate the return value of the function.

Here is how we call the function:

```
shoe_box_volume = get_volume_of_box(30, 20, 15)
```

The function call, once evaluated, will be replaced by whatever that function returned. So in our example, the volume of the box.

Naming Functions

The rules for naming functions are the same as for variables (see page 2).

Conventions you should follow (not enforced, but considered good style):

- Start function names with a lowercase letter
- Use underscores for multiple words in variables: **`get_volume_of_cube()`**
- Function names usually start with a verb, to indicate what they do.
- Functions that return Boolean values should start with (or at least contain) ***is*** or ***has*** or something else that indicates it returns True/False:
    - **`has_wings()`** (maybe describes an insect)
    - **`is_switch_on()`**

Optional Arguments (Keyword Arguments)

Here is an example of a function that accepts two optional arguments:

```
def draw_circle(x, y, radius, color="black", width=1):
    # Add code to draw the circle here.
```

Here, **`color`** and **`width`** are optional, since they've been provided with default values of **black** and **1** respectively.

There are various valid ways to call this function:

```
# Use default values for color and width
draw_circle(100, 100, 30)

# Use red for the colour, default value of 1 for width.
draw_circle(100, 100, 30, "red")

# Use red for the colour and 3 for the width
draw_circle(100, 100, 30, "red", 3)

# Use 3 for the width, and the default of black for the colour
draw_circle(100, 100, 30, width=3)
```

Rules for calling functions with optional arguments:

- All mandatory arguments must be specified first, **in order**
- As many optional arguments as desired may be added, **in order**
- Further optional arguments can be added by specifying the name and value. For instance, **`width=3`**
    - When specifying arguments in this manner, you can go in whatever order you'd like, and skip some optional arguments to use their default values.

WORKING WITH OBJECTS

Objects contain:

- **attributes** – characteristics about the object. These are really just ***variables*** attached to the object.
- **methods** – things the object can do. These are really just ***functions*** attached to the object.

This example supposes we have an object called **`car`**.

```
# Working with attributes

car.color = "red"              # sets the car's `color` attribute
print(car.model)               # prints the car's model


# Working with methods

car.turn_right()               # calls a method that accepts no arguments
car.accelerate_to(60)          # calls a method that accepts one argument
```

As seen in the example, to access the attributes and methods of an object, you need to start by specifying the object's name, then add a dot followed by the attribute or method name.

To create the object in the first place, we need to do something like this:

```
car = Vehicle("Honda", "Civic")
```

In this example, **`Vehicle`** is a predefined **class**. A class is used to create objects. The class specifies:

- What attributes the object will have to start, and their values
- What methods the object will have
- What arguments are required upon creation of the object (in the example above, we might guess that the **`Vehicle`** class requires the make and model of the vehicle to be specified upon its creation).

We can create our own classes, but mostly we'll be using predefined classes defined in other modules. You will be provided with documentation for any class you need to use, so you know what attributes and methods are available, as well as what arguments need to be specified when the class is used to create a new object.