

**Programming
Level-up**

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

Programming Level-up

Lecture 1 - Introduction and Basic Python Programming

Jay Morgan

16th September 2022

Outline

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

1 Introduction

- Course introduction
- Contact information

2 Python

- Introducing Python
- Types of data
- Working with strings
- Compound data structures
- Conditional expressions
- Iteration
- Functions

3 Exercise

- Library system

What...? Why...?

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

- Programming is much more than the act of programming a small script. Even if you've programmed before, doing so for a research project requires a lot of rigour to ensure the results you're reporting are correct, and reproducible.
- There is so much surrounding the act of programming that it can get a little overwhelming. Things from setting up a programming environment to managing multiple experiments on the supercomputers can involve many languages and understanding of technologies.
- This course is designed to take you from not being able to program at all to being able to do it comfortably for your research and work.

What is this course going to teach me?

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

- 1 Programming with the Python Programming Language.
 - Basic syntax.
 - Introduction to the basics of object oriented programming (OOP).
 - Numerical computing with numpy/pandas/scipy.
- 2 Doing your programming in a Linux-based Environment (GNU/Linux) and being comfortable with the organisation of this Linux environment.
 - Setting up a research (reproducible) environment.
 - Executing experiments.
- 3 Interacting with the Super-computers/clusters.
 - Interaction with SLURM (management of jobs).
- 4 Taking the results from a program you've created, be able to visualise them and include them in reports/papers.
 - \LaTeX /Markdown.
 - Plotting.

How the course will be delivered

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

- 2/3 hour sessions over the next 2 months.
- Throughout the lecture, there will be small exercises to try out what we've learnt. We will go through the answers to these exercises.
- At the end of the lecture we will have a larger exercise that will become more challenging. These exercises are not marked, but again, just an opportunity to try out what you've learnt. The best way to learn how to program is to program.

Rough timeline

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Lecture	Topic	Description
1	Introduction	<ul style="list-style-type: none">- Course introduction- Basic Python programming
2	Python classes	<ul style="list-style-type: none">- Introduction to OOP
3	Project management	<ul style="list-style-type: none">- Creating/importing modules- Anaconda/pip
4	Programming environments	<ul style="list-style-type: none">- PyCharm- Jupyter notebooks
5	Numerical computing	<ul style="list-style-type: none">- Numpy- Scipy
6	Numerical computing	<ul style="list-style-type: none">- Pandas- Visualisations
7	Basics of GNU/Linux	<ul style="list-style-type: none">- Using the terminal
8	Bash scripting	<ul style="list-style-type: none">- Bash scripting
9	High performance computing	<ul style="list-style-type: none">- SLURM- Singularity
10	Reporting	<ul style="list-style-type: none">- L^AT_EX- Markdown

Where to find me

Programming
Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

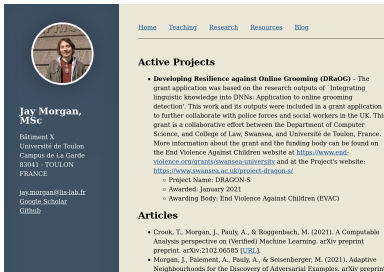
Functions


Exercise

Library system

My name is Dr Jay Morgan. I am a researcher work on Deep Learning in Astrophysics.

- Email: jay.morgan@univ-tln.fr
- Lecture slides and other contact on my website:
<https://pageperso.lis-lab.fr/jay.morgan/>





**Jay Morgan,
MSc**

Bâtiment X
Université de Toulon
Campus de La Garde
83041 - TOULON
FRANCE

jay.morgan@lis-lab.fr
Google Scholar
GitHub

[Home](#) [Teaching](#) [Research](#) [Resources](#) [Blog](#)

Active Projects

- **Developing Resilience against Online Grooming (DRaOG)** - The grant application was based on the research outputs of 'Integrating linguistic knowledge into DNNs: Application to online grooming detection'. This work and its outputs were included in a grant application to further collaborate with police forces and social workers in the UK. This grant is a collaborative effort between the Department of Computer Science, and College of Law, Swansea, and Université de Toulon, France. More information about the grant and the funding body can be found on the End Violence Against Children website at <https://evac.endviolence.org/grants/swansea-university> and at the Project's website: <https://www.swansea.ac.uk/project-dragon/sf>
 - Project Name: DRAGON-S
 - Awarded: January 2021
 - Awarding Body: End Violence Against Children (EVAC)

Articles

- Crook, T., Morgan, J., Pauly, A., & Roggenbach, M. (2021). A Computable Analysis perspective on (Verified) Machine Learning. arXiv preprint [arXiv:2102.06505](https://arxiv.org/abs/2102.06505) [URL]
- Morgan, J., Palement, A., Pauly, A., & Seisenberger, M. (2021). Adaptive Neighbourhoods for the Discovery of Adversarial Examples. arXiv preprint

Python

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system



Python

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

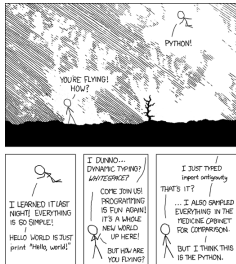
Iteration

Functions

Exercise

Library system

- Python is a *high-level*¹ programming language created in 1991.
- While it is an old language, it's become vastly popular thanks to its use in data science and other mathematics-based disciplines. While also being able to perform tasks such as GUI, web-development and much more.
- Because the language is high-level and *interpreted*, programmers can often find themselves more productive in Python than in other languages such as say C++.



¹As we go through our lectures we'll understand what it means for the language to be /high-level/ and /interpreted/ and why that is helpful for us.

A first program

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

We're going to start with the 'Hello, World' program that prints Hello, World! to the screen. In python this is as simple as writing:

```
1 print("Hello, World!") # this prints: Hello, World!
```

Results:

```
# => Hello, World!
```

NOTE anything following a # is a comment and is completely ignored by the computer. It is there for you to document your code for others, and most importantly, for yourself.

Running this program

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Before we can run this program, we need to save it somewhere. For this, we will create a new file, insert this text, and save it as `<filename>.py`, where `<filename>` is what we want to call the script. This name doesn't matter for its execution.

Once we have created the *script*, we can run it from the *command line*. We will get into the command line in a later lecture, but right now all you need to know is:

```
3 python3 <filename>.py
```

An alternative method of running python

Programming Level-up

Jay Morgan

Introduction

Course introduction
Contact information

Python

Introducing Python

Types of data
Working with strings
Compound data structures
Conditional expressions
Iteration
Functions

Exercise

Library system

You may notice that if you don't give `python` a filename to run, you will enter something called the REPL.

```
4 Python 3.9.5 (default, Jun 4 2021, 12:28:51)
5 [GCC 7.5.0] :: Anaconda, Inc. on linux
6 Type "help", "copyright", "credits" or "license" for more
  ↪ information.
7 >>>
```

REPL stands for READ, EXECUTE, PRINT, LOOP.

Variables

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

A *variable* is a *symbol* associated with a *value*. This value can differ widely, and we will take a look at different types of values/data later.

Nevertheless, variables are useful for *referring* to values and *storing* to the results of a computation.

```
8 x = 1
9 y = 2
10 z = x + y
11 print(z) # prints: 3
12
13 # variables can be /overwritten/
14 z = "hello, world"
15 print(z) # prints: hello, world
```

Results:

```
# => 3
```

```
# => hello, world
```

Primitive data types

Programming Level-up

Jay Morgan

Introduction

Course introduction
Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Primitive data types are the most fundamental parts of programming, they cannot be *broken* down.

```
4 "Hello" # string
5 1      # integer
6 1.0    # float
7 True   # Boolean (or bool for short)
```

Primitive data type

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

We can get the type of some data by using the `type(...)` function. For example,

```
8 print(type(5))
9 print(type(5.0))
10
11 x = "all cats meow"
12
13 print(type(x))
```

Results:

```
# => <class 'int'>
# => <class 'float'>
# => <class 'str'>
```

Basic Math with primitives

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Using these primitive data types, we can do some basic math operations!

```
5 print(1 + 2)      # Addition
6 print(1 - 2)      # Subtraction
7 print(1 * 2)      # Multiplication
8 print(1 / 2)      # Division
9 print(2 ** 2)     # Exponent
10 print(3 % 2)     # Modulo operator
```

Results:

```
# => 3
# => -1
# => 2
# => 0.5
# => 4
# => 1
```


Basic Math

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Sometimes types get converted to the same type:

```
8 print(1.0 + 2) # float + integer = float
```

Results:

```
# => 3.0
```

Even more interesting is with Booleans!

```
3 True + True
```

Results:

```
# => 2
```

BODMAS in Python

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

Like in mathematics, certain math operator take precedence over others.

- B - Brackets
- O - Orders (roots, exponents)
- D - division
- M - multiplication
- A - addition
- S - subtraction.

To make the context clear as to what operations to perform first, use brackets.

```
3 (5 / 5) + 1
4 5 / (5 + 1)
```

Results:

```
# => 2.0
```

```
# => 0.8333333333333334
```

Basic Math – Quick exercise

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Write the following equation in python:

$$(5 + 2) \times \left(\frac{10}{2} + 10\right)^2$$

Remember to use parentheses () to ensure that operations take precedence over others.

Your answer should come out as: 1575.0

Formatting strings

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

In many previous examples when we've printed strings, we've done something like:

```
4 age = 35
5
6 print("The value of age is", age)
```

Results:

```
# => The value of age is 35
```

While this works in this small context, it can get pretty cumbersome if we have many variables we want to print, and we also want to change how they are displayed when they are printed.

We're going to take a look now at much better ways of printing.

Better ways of printing strings - %

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

The first method is using %. When we print, we first construct a string with special delimiters, such as %s that denotes a string, and %d that denotes a number. This is telling Python where we want the values to be placed in the string.

Once we've created the string, we need to specify the data, which we do with % (...). Like, for example:

```
3 age = 35
4 name = "John"
5
6 print("%d years old" % age) # no tuple for one variable
7 print("%s is %d years old" % (name, age))
```

Results:

```
# => 35 years old
```

```
# => John is 35 years old
```

Here we are specifying the a string %s and number %d, and then giving the variables that correspond with that data type.

Better ways of printing strings – data specifiers

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

The special delimiters correspond with a data type. Here are some of the most common:

- `%s` – For strings
- `%d` – For numbers
- `%f` – For floating point numbers.

There are others such as `%x` that prints the hexadecimal representation, but these are less common. You can find the full list at: <https://docs.python.org/3/library/stdtypes.html#old-string-formatting>

Better ways of printing strings – floating points

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

When using these delimiters, we can add modifiers to how they format and display the value. Take a very common example, where we have a floating point value, and, when printing it, we only want to print to 3 decimal places. To accomplish this, we again use `%f` but add a `.3` to between the `%` and `f`. In this example, we are printing π to 3 decimal places.

4

```
print("Pi to 3 digits is: %.3f" % 3.1415926535)
```

Results:

```
# => Pi to 3 digits is: 3.142
```

Better ways of printing strings – floating points

Programming Level-up

Jay Morgan

Introduction

Course introduction
Contact information

Python

Introducing Python
Types of data

Working with strings

Compound data structures
Conditional expressions

Iteration
Functions

Exercise

Library system

In the previous example, we used `.3` to specify 3 decimal places. If we put a number before the decimal, like `10.3` we are telling Python *make this float occupy 10 spaces and this float should have 3 decimal places printed*. When it gets printed, you will notice that it shifts to the right, it gets padded by space. If we use a negative number in front of the decimal place, we are telling python to shift it to the left.

```
3 print("Pi to 3 digits is: %10.3f" % 3.1415926535)
4 print("Pi to 3 digits is: %-10.3f" % 3.1415926535)
```

Results:

```
# => Pi to 3 digits is:      3.142
# => Pi to 3 digits is: 3.142
```


Better ways of printing strings – f-strings

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

The final method of formatting strings is a newcomer within the language, it is the so-called **f-string**. Where a **f** character is prefixed to the beginning of the string you're creating. **f-string's** allow you to use Python syntax within the string (again delimited by **{}**).

Take this for example where we are referencing the variables `name` and `age` directly.

```
4 name = "Jane"
5 age = 35
6
7 print(f"{name} is {age} years old")
```

Results:

```
# => Jane is 35 years old
```

Better ways of printing strings – f-strings

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

f-string's allow you to execute Python code within the string. Here we are accessing the value from the dictionary by specifying the key within the string itself! It certainly makes it a lot easier, especially if we only need to access the values for the string itself.

```
3 contact_info = {"name": "Jane", "age": 35}
4
5 print(f"{contact_info['name']} is {contact_info['age']} years
   ↪ old")
```

Results:

```
# => Jane is 35 years old
```

<https://pyformat.info/>

Better ways of printing strings – f-string

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

We can still format the values when using f-string. The method is similar to those using the %f specifiers.

```
3 pi = 3.1415926535
4 print(f"Pi is {pi:.3f} to 3 decimal places")
```

Results:

```
# => Pi is 3.142 to 3 decimal places
```

Many more examples can be found at:

<https://zetcode.com/python/fstring/>

Operations on strings – splitting

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

Apart from formatting, there are plenty more operations we can perform on strings. We are going to highlight some of the most common here.

The first we're going to look at is splitting a string by a delimiter character using the `.split()` method. If we don't pass any argument to the `.split()` method, then by default, it will split by spaces. However, we can change this by specifying the delimiter.

```
3 my_string = "This is a sentence, where each word is separated by  
   ↪ a space"  
4  
5 print(my_string.split())  
6 print(my_string.split(", "))
```

Results:

```
# => ['This', 'is', 'a', 'sentence,', 'where', 'each', 'word', 'is', 'separated',  
# => ['This is a sentence', ' where each word is separated by a space']
```

Operations on strings – joining

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

As `.split()` splits a single string into a list, `.join()` joins a list of strings into a single string. To use `.join()`, we first create a string of the delimiter we want to use to join the list of strings by. In this example we're going to use `"-"`. Then we call the `.join()` method, passing the list as an argument.

The result is a single string using the delimiter to separate the items of the list.

```
4 x = ['This', 'is', 'a', 'sentence,', 'where', 'each', 'word',  
↪   'is', 'separated', 'by', 'a', 'space']  
5  
6 print("-".join(x))
```

Results:

```
# => This-is-a-sentence,-where-each-word-is-separated-by-a-space
```

Operations on strings – changing case

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Other common operations on strings involve change the case. For example:

- Make the entire string uppercase or lowercase
- Making the string title case (every where starts with a capital letter).
- Stripping the string by removing any empty spaces either side of the string.

Note we can chain many methods together by doing `.method_1().method_2()`, but only if they return string. If they return `None`, then chaining will not work.

```
3 x = "    this String Can change case"
4
5 print(x.upper())
6 print(x.lower())
7 print(x.title())
8 print(x.strip())
9 print(x.strip().title())
```

Operations on strings – replacing strings

Programming Level-up

Jay Morgan

Introduction

Course introduction
Contact information

Python

Introducing Python
Types of data
Working with strings
Compound data structures
Conditional expressions
Iteration
Functions
Exercise
Library system

To replace a substring, we use the `.replace()` method. The first argument is the old string you want to replace. The second argument is what you want to replace it with.

```
7 x = "This is a string that contains some text"  
8  
9 print(x.replace("contains some", "definitely contains some"))
```

Results:

```
# => This is a string that definitely contains some text
```

Container data types/Data structures

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Container data types or data structures, as the name suggests, are used to contain other things. Types of containers are:

- Lists
- Dictionaries
- Tuples
- Sets

```
3 [1, "hello", 2]           # list
4 {"my-key": 2, "your-key": 1} # dictionary (or dict)
5 (1, 2)                   # tuple
6 set(1, 2)                 # set
```

We'll take a look at each of these different container types and explore why we might want to use each of them.

An aside on Terminology

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

To make our explanations clearer and reduce confusion, each of the different symbols have unique names.

I will use this terminology consistently throughout the course, and it is common to see the same use outside the course.

- [] brackets (square brackets).
- { } braces (curly braces).
- () parentheses.

Lists

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

A heterogeneous container. This means that it can store any type of data.

```
7 x = [1, "hello", 2]
```

Elements can be accessed using indexing [] notation. For example:

```
8 print(x[0]) # this will get the first element (i.e. 1)
9 print(x[1]) # the second element (i.e. "hello")
10 print(x[2]) # the third element (i.e. 2)
```

Results:

```
# => 1
# => hello
# => 2
```

notice how the first element is the 0-th item in the list/ we say that python is 0-indexed.

Better indexing – slices

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

If we wanted to access an element from a data structure, such as a list, we would use the `[]` accessor, specifying the index of the element we wish to retrieve (remember that indexes start at zero!). But what if we wanted to access many elements at once? Well to accomplish that, we have a slice or a range of indexes (not to be confused with the `range` function). A slice is defined as:

```
start_index:end_index
```

where the `end_index` is non inclusive – it doesn't get included in the result. Here is an example where we have a list of 6 numbers from 0 to 5, and we slice the list from index 0 to 3. Notice how the 3rd index is not included.

```
2 x = [0, 1, 2, 3, 4, 5]
3 print(x[0:3])
```

Results:

```
# => [0, 1, 2]
```

Better indexing – range

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

When we use `start_index:end_index`, the slice increments by 1 from `start_index` to `end_index`. If we wanted to increment by a different amount we can use the slicing form:

```
start_index:end_index:step
```

Here is an example where we step the indexes by 2:

```
2 x = list(range(100))
3 print(x[10:15:2])
```

Results:

```
# => [10, 12, 14]
```

Better indexing – reverse

Programming Level-up

Jay Morgan

Introduction

Course
introduction
Contact
information

Python

Introducing
Python
Types of data
Working with
strings
**Compound data
structures**

Conditional
expressions

Iteration
Functions

Exercise

Library system

One strange fact about the step is that if we specify a negative number for the step, Python will work backwards, and effectively reverse the list.

```
3 x = list(range(5))
4
5 print(x[::-1])
```

Results:

```
# => [4, 3, 2, 1, 0]
```

Better indexing – range

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

In a previous example, I created a slice like `0:3`. This was a little wasteful as we can write slightly less code. If we write `:end_index`, Python assumes and creates a slice from the first index (0) to the `end_index`. If we write `start_index:`, Python assumes and creates a slice from `start_index` to the end of the list.

```
3 x = list(range(100))
4
5 print(x[:10])
6 print(x[90:])
```

Results:

```
# => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
# => [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

Better indexing – backwards

Programming Level-up

Jay Morgan

Introduction

Course introduction
Contact information

Python

Introducing Python
Types of data
Working with strings
Compound data structures
Conditional expressions
Iteration
Functions

Exercise

Library system

Finally, we also work backwards from the end of list. If we use a negative number, such as `-1`, we are telling Python, take the elements from the end of the list. `-1` is the final index, and numbers lower than `-1` work further backwards through the list.

```
4 x = list(range(100))
5
6 print(x[-1])
7 print(x[-2])
```

Results:

```
# => 99
# => 98
```

Better indexing –backwards

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

Slicing with negative indexes, also works. Here we are creating a slice from the end of the list - 10, to the last (but not including) index.

```
4 x = list(range(100))
5
6 print(x[-10:-1])
```

Results:

```
# => [90, 91, 92, 93, 94, 95, 96, 97, 98]
```


Lists – adding data

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

If we want to add items to the end of the list, we use the `append` function:

```
3 my_list = []
4
5 my_list.append("all")
6 my_list.append("dogs")
7 my_list.append("bark")
8
9 print(my_list)
```

Results:

```
# => ['all', 'dogs', 'bark']
```

Dictionaries

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

Dictionaries are a little different from lists as each 'element' consists of a key-pair value. Let's have a look at some examples where the dictionary contains **one** element:

```
3 my_dictionary = {"key": "value"}
4 my_other_dict = {"age": 25}
```

To access the *value*, we get it using [key] notation:

```
5 my_other_dict["age"]
```

Results:
=> 25

NOTE keys are unique, i.e:

```
3 my_dictionary = {"age": 25, "age": 15}
4 my_dictionary["age"]
```

Dictionaries

Programming Level-up

Jay Morgan

Introduction

Course
introduction
Contact
information

Python

Introducing
Python
Types of data
Working with
strings

Compound data structures

Conditional
expressions
Iteration
Functions

Exercise

Library system

The key in the dictionary doesn't necessarily need to be a string. For example, in this case, we have created two key-pair elements, where the keys to both are tuples of numbers.

```
3 my_dictionary = {(1, 2): "square", (3, 4): "circle"}
4
5 print(my_dictionary[(1, 2)])
```

Results:
=> square

Dictionaries – adding data

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

**Compound data
structures**

Conditional
expressions

Iteration

Functions

Exercise

Library system

If we want to add data to a dictionary, we simply perform the accessor method with a key that is not in the dictionary:

```
3 my_dict = {}
4
5 my_dict["name"] = "James"
6 my_dict["age"] = 35
7
8 print(my_dict)
```

Results:

```
# => {'name': 'James', 'age': 35}
```

Dictionaries – Quick Exercise

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

- Create a dictionary for the following address, and assign it a variable name called `address`:

Key	Value
number	22
street	Bakers Street
city	London

- Print out the address's street name using the `[]` accessor with the correct key.

Tuples

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

```
3 my_tuple = (1, 56, -2)
```

Like lists, elements of the tuple can be accessed by their position in the list, starting with the 0-th element:

```
4 print(my_tuple[0]) # => 1
5 print(my_tuple[1]) # => 56
6 print(my_tuple[2]) # => -2
```

Results:

```
# => 1
# => 56
# => -2
```

Tuples

Programming Level-up

Jay Morgan

Introduction

Course introduction
Contact information

Python

Introducing Python
Types of data
Working with strings
Compound data structures
Conditional expressions
Iteration
Functions

Exercise

Library system

Unlike lists, tuples cannot be changed after they've been created. We say they are **immutable**. So this will **not** work:

```
5 my_tuple[2] = "dogs" # creates an Error
```

Results:

```
# => Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "/tmp/pyKdIIcx", line 18, in <module>  
  File "<string>", line 1, in <module>  
TypeError: 'tuple' object does not support item assignment
```

Sets

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

**Compound data
structures**

Conditional
expressions

Iteration

Functions

Exercise

Library system

Sets in Python are like tuples, but contain only unique elements.

You can use the `set()` function (**more on functions later!**), supplying a list, to create a set:

```
7 my_set = set([1, 2, 2, 2, 3, 4])
8 my_set
```

Results:

```
# => {1, 2, 3, 4}
```

Notice how there is only one '2' in the resulting set, duplicate elements are removed.

Sets – adding data

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

If we want to add data to a set, we use the `.add()` method. The element used as an argument to this function will only be added to the set if it is not already in the set.

```
3 my_set = set([])
4
5 my_set.add(1)
6 my_set.add(2)
7 my_set.add(1)
8
9 print(my_set)
```

Results:
=> {1, 2}

If statement

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

If statements allow for branching paths of execution. In other words, we can execute some statements if some conditions holds (or does not hold).

The structure of a simple if statement is:

```
3 if <condition>:  
4     <body>
```

```
5 x = 2  
6 y = "stop"  
7  
8 if x < 5:  
9     print("X is less than five")  
10 if y == "go":  
11     print("All systems go!!!")
```

Results:

```
# => X is less than five
```

If statement

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

In the previous example, the first print statement was only executed if the $x < 5$ evaluates to True, but in python, we can add another *branch* if the condition evaluates to False. This branch is denoted by the else keyword.

```
3 x = 10
4
5 if x < 5:
6     print("X is less than five")
7 else:
8     print("X is greater than or equal to five")
```

Results:

```
# => X is greater than or equal to five
```

If statement – does it contain a substring?

Programming Level-up

Jay Morgan

Introduction

Course
introduction
Contact
information

Python

Introducing
Python
Types of data
Working with
strings
Compound data
structures

Conditional expressions

Iteration
Functions

Exercise

Library system

We can check if a string exists within another string using the `in` keyword. This returns a Boolean value, so we can use it as a condition to an `if` statement.

```
3 x = "This is a string that contains some text"
4
5 if "text" in x:
6     print("It exists")
```

Results:

```
# => It exists
```

If statement – Quick Exercise 1

Programming Level-up

Jay Morgan

Introduction

Course
introduction
Contact
information

Python

Introducing
Python
Types of data
Working with
strings
Compound data
structures

Conditional expressions

Iteration
Functions

Exercise

Library system

- Create a variable called `age` and assign the value of this variable 35.
- Create an `if` statement that prints the square of `age` if the value of `age` is more than 24.
- This `if` statement should have an `else` condition, that prints `age` divided by 2.
- What is the printed value?

If statement

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

If we wanted to add multiple potential paths, we can add more using the `elif <condition>` keywords.

Note: The conditions are checked from top to bottom, only executing the `else` if none evaluate to `True`. The first condition that evaluates to `True` is executed, the rest are skipped.

```
3 x = 15
4
5 if x < 5:
6     print("X is less than five")
7 elif x > 10:
8     print("X is greater than ten")
9 else:
10    print("X is between five and ten")
```

Results:

```
# => X is greater than ten
```

If statement

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Sometimes, we might want to conditionally set a variable a value. For this, we can use an *inline* if statement. The form of an inline if statement is:

```
<value-if-true> if <condition> else <value-if-false>
```

```
3 x = 10
4
5 y = 5 if x > 5 else 2
6
7 print(x + y)
```

Results:

```
# => 15
```

Boolean Logic

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

As we've seen, if statements are checking for conditions to evaluate to True or False. In python we use various comparison operators to check for conditions that evaluate to Booleans.

Comparison operators

- $<$ less than
- $<=$ less than or equal to
- $>$ greater than
- $>=$ greater than or equal to
- $==$ is equal to
- not negation

If we want to check for multiple conditions, we can use conjunctives or disjunctive operators to combine the Boolean formulas.

Conjunctives/Disjunctives

- and all boolean expressions must evaluate to true
- or only one expression needs to be true

Boolean Logic

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Using `not` you can invert the Boolean result of the expression.

```
3 print(not True)
```

Results:

```
# => False
```

```
3 x = 10
4
5 if not x == 11:
6     print("X is not 11")
```

Results:

```
# => X is not 11
```

Boolean Logic

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Let's take an example using the `and` keyword. `and` here is checking that `x` is above or equal to 10 **and** `y` is exactly 5. If either of the conditions is `False`, python will execute the `else` path (if there is one, of course!).

```
3 x = 10
4 y = 5
5
6 if x >= 10 and y == 5:
7     z = x + y
8 else:
9     z = x * y
10
11 print(z)
```

Results:

```
# => 15
```

Boolean Logic

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Here we see the use of the `or` keyword. If any of the conditions evaluates to `True` then the whole condition evaluates to `True`.

```
3 x = 10
4 y = 5
5
6 if x < 5 or y == 5:
7     print("We got here!")
8 else:
9     print("We got here instead...")
```

Results:

```
# => We got here!
```

Boolean Logic

Programming Level-up

Jay Morgan

Introduction

Course introduction
Contact information

Python

Introducing Python
Types of data
Working with strings
Compound data structures

Conditional expressions

Iteration
Functions

Exercise

Library system

Note: `or` is short-circuiting. This means that it tests the conditions left-to-right, and when it finds something that is `True` it stops evaluating the rest of the conditions.

```
3 x = 10
4
5 if x < 20 or print("We got to this condition"):
6     print("The value of x is", x)
```

Results:

```
# => The value of x is 10
```

Boolean Logic

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

If your Boolean logic refers to a single variable, you can combine the logic without the `and` and `or`. But its not always common.

For example,

```
3 x = 7
4
5 if x < 10 and x > 4:
6     print("X is between 5 and 10")
```

Can be the same as:

```
7 x = 7
8
9 if 5 < x < 10:
10     print("X is between 5 and 10")
```

Results:

```
# => X is between 5 and 10
```

For loop

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Looping or iteration allows us to perform a series of actions multiple times. We are going to start with the more useful for loop in python. The syntax of a for loop is:

```
3 for <variable_name> in <iterable>:  
4     <body>
```

```
5 for i in range(3):  
6     print(i)
```

Results:

```
# => 0  
# => 1  
# => 2
```

For loop – break

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

The previous example loops over the body a fix number of times. But what if we wanted to stop looping early? Well, we can use the `break` keyword. This keyword will exit the body of the loop.

```
5 for i in range(10):
6     if i > 5:
7         break
8     print(i)
```

Results:

```
# => 0
# => 1
# => 2
# => 3
# => 4
# => 5
```

For loop – continue

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

A different keyword you might want to use is `continue`. `Continue` allows you to move/skip onto the next iteration without executing the entire body of the `for` loop.

```
8  for i in range(10):
9      if i % 2 == 0:
10         continue
11         print(i)
```

Results:

```
# => 1
# => 3
# => 5
# => 7
# => 9
```


For loop – ranges

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Instead of using `continue` like in the previous slide, the `range` function provides us with some options:

```
range(start, stop, step)
```

In this example, we are starting our iteration at 10, ending at 15, but stepping the counter 2 steps.

```
7 for i in range(10, 15, 2):  
8     print(i)
```

Results:

```
# => 10  
# => 12  
# => 14
```

For loop – loop over collections

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

For loops allow us to iterate over a collection, taking one element at a time. Take for example, a list, and for every item in the list we print its square.

```
5 my_list = [1, 5, 2, 3, 5.5]
6
7 for el in my_list:
8     print(el**2)
```

Results:

```
# => 1
# => 25
# => 4
# => 9
# => 30.25
```

For loop – loop over collections

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

This kind of looping can work for tuples and sets, but as we have seen, dictionaries are a little different. Every 'element' in a dictionary consists of a key and a value. Therefore when we iterate over items in a dictionary, we can assign the key and value to different variables in the loop.

Note the use of the `.items()` after the dictionary. We will explore this later.

```
7 my_dict = {"name": "jane", "age": 35, "loc": "France"}
8
9 for el_key, el_val in my_dict.items():
10     print("Key is:", el_key, " value is: ", el_val)
```

Results:

```
# => Key is: name and the value is: jane
# => Key is: age and the value is: 35
# => Key is: location and the value is: France
```

For loop – loop over collections

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

We could also loop over the keys in the dictionary using the `.keys()` method instead of `.items()`.

```
5 my_dict = {"name": "jane", "age": 35, "loc": "France"}
6
7 for the_key in my_dict.keys():
8     print(the_key)
```

Results:

```
# => name
# => age
# => loc
```

For loop – loop over collections

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

Or, the values using `.values()`.

```
5 my_dict = {"name": "jane", "age": 35, "loc": "France"}
6
7 for the_value in my_dict.values():
8     print(the_value)
```

Results:

```
# => jane
# => 35
# => France
```

For loop – List comprehensions

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

We have seen previously how for loops work. Knowing the syntax of a for loop and wanting to populate a list with some data, we might be tempted to write:

```
5 x = []
6 for i in range(3):
7     x.append(i)
8
9 print(x)
```

Results:

```
# => [0, 1, 2]
```

While this is perfectly valid Python code, Python itself provides 'List comprehensions' to make this process easier.

```
3 x = [i for i in range(3)]
```

For loop – List comprehensions – syntax

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

The syntax of a list comprehensions is:

```
4 [ <variable> for <variable> in <iterable> ]
```

We can also perform similar actions with a dictionary

```
5 [ <key>, <value> for <key>, <value> in <dictionary.items()> ]
```

For loop – List comprehensions – using if's

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Perhaps we only want to optionally perform an action within the list comprehension? Python allows us to do this with the inline `if` statement we've seen in the previous lecture.

```
6 x = [i if i < 5 else -1 for i in range(7)]
7 print(x)
```

Results:

```
# => [0, 1, 2, 3, 4, -1, -1]
```

We add the inline `<var> if <condition> else <other-var>` before the `for` loop part of the comprehension.

For loop – List comprehension – using if's

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

There is another type of `if` statement in a list comprehension, this occurs when we don't have an `else`.

```
3 x = [i for i in range(7) if i < 3]
4 print(x)
```

Results:

```
# => [0, 1, 2]
```

In this example, we're only 'adding' to the list if the condition ($i < 3$) is true, else the element is not included in the resulting list.

For loop – List comprehensions – multiple for's

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

If we like, we can also use nested for loops by simply adding another for loop into the comprehension.

```
3 x = [(i, j) for i in range(2) for j in range(2)]
4
5 print(x)
```

Results:

```
# => [(0, 0), (0, 1), (1, 0), (1, 1)]
```

In this example, we're creating a tuple for each element, effectively each combination of 1 and 0.

For loop – List comprehensions – dictionary

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

Python doesn't restrict us to list comprehensions, but we can do a similar operation to create a dictionary.

```
3 x = [2, 5, 6]
4 y = {idx: val for idx, val in enumerate(x)}
5 print(y)
```

Results:

```
# => {0: 2, 1: 5, 2: 6}
```

Here, every item in `x` has been associated with its numerical index as a key thanks to the `enumerate` function that returns both the index and value at iteration in the for loop.

For loop – Quick Exercise

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

- Create a list of elements:
 - 2
 - "NA"
 - 24
 - 5
- Use a for loop to iterate over this list.
- In the body of the for loop, compute $2x + 1$, where x is the current element of the list.
- Store the result of this computation in a new variable y , and then print y .

Note You cannot compute $2x + 1$ of "NA", therefore you will to use an if statement to skip onto the next iteration if it encounters this.

Hint try: `type(...) != str`

While loop

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

A while loop is another looping concept like for but it can loop for an arbitrary amount of times. A while loop looks to see if the condition is True, and if it is, it will execute the body.

The syntax of the while loop is:

```
3 while <condition>:  
4     <body>
```

```
5 i = 0  
6  
7 while i < 3:  
8     print(i)  
9     i = i + 1
```

Results:

```
# => 0  
# => 1  
# => 2
```

While loop

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

```
5 x = 0
6 y = 1
7
8 while x + y < 10:
9     print("X is,", x, "and y is", y)
10    x = x + 1
11    y = y * 2
12
13 print("X ended as", x, ", while y is", y)
```

Results:

```
# => X is, 0 and y is 1
# => X is, 1 and y is 2
# => X is, 2 and y is 4
# => X ended as 3 , while y is 8
```

Functions

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

Functions are a re-usable set of instructions that can take some arguments and possibly return something.

The basic structure of a function is as follows:

```
6 def <function_name>(args*):  
7     <body>  
8     (optional) return
```

- `args*` are 0 to many comma separated symbols.
- `body` is to be indented by 4 spaces.

This is only the function *definition* however. To make it do something, we must '*call*' the function, and supply the arguments as specified in the definition.

```
9 def say_hello(): # function definition  
10     print("Hello, World!")  
11  
12 say_hello() # calling the function
```

Functions

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

We've already seen some functions provided by Python.

`print` itself is a function with a single argument: what we want to print.

```
13 print("Hello, World!")
14 # ~           ~
15 # /           /
16 # / user supplied argument
17 # /
18 # function name
```

`set` is another function that takes a single argument: a collection of data with which to make a set:

```
19 set([1, 2, 2, 3, 4])
```


Example usage of a function

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Let's make a function that takes two numbers and adds them together:

```
20 def my_addition(a, b):
21     result = a + b
22     return result
23
24 x = 2
25 y = 3
26 z = my_addition(2, 3) # return 5 and stores in z
27 print(z)
```

Results:

```
# => 5
```

Functions – Quick Exercise

Programming Level-up

Jay Morgan

Introduction

Course
introduction
Contact
information

Python

Introducing
Python
Types of data
Working with
strings
Compound data
structures
Conditional
expressions
Iteration
Functions

Exercise

Library system

- Create a function called `my_square`. This function should take one argument (you can call this argument what you like).
- The body of the function should compute and return the square of the argument.
- Call this function with `5.556`.
- Store the result of calling this function, and print it.
- What is the result?

Re-usability with Functions

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Functions are better illustrated through some examples, so let's see some!

```
3 name_1 = "john"
4 name_2 = "mary"
5 name_3 = "michael"
6
7 print("Hello " + name_1 + ", how are you?")
8 print("Hello " + name_2 + ", how are you?")
9 print("Hello " + name_3 + ", how are you?")
```

The above is pretty wasteful. Why? Because we are performing the exact same operation multiple times, with only the variable changed.

Re-usability with Functions

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

By abstracting the actions we want to perform into a function, we can ultimately reduce the amount of code we write. *Be a lazy programmer!*

```
10 name_1 = "john"
11 name_2 = "mary"
12 name_3 = "michael"
13
14 def say_hello(name):
15     print("Hello " + name + ", how are you?")
16
17 say_hello(name_1)
18 say_hello(name_2)
19 say_hello(name_3)
```

In this example, we've used the function as defined with the `def` pattern to write the `print` statement once. Then, we've called the function with each variable as its argument.

Named parameters

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

We've seen in previous examples that, when we create a function, we give each of the arguments (if there are any) a name.

When calling this function, we can specify these same names such as:

```
20 def say_hello(name):
21     print("Hello,", name)
22
23     say_hello("Micheal")
24     say_hello(name="Micheal")
```

Results:

```
# => Hello, Micheal
# => Hello, Micheal
```

Named parameters

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

By specifying the name of the parameter we're using with the called function, we can change the order

```
4 def say_greeting(greeting, name):  
5     print(greeting, name, "I hope you're having a good day")  
6  
7 say_greeting(name="John", greeting="Hi")
```

Results:

```
# => Hi John I hope you're having a good day
```

Optional/Default/Positional arguments

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

When we call a function with arguments without naming them, we are supplying them by *position*.

```
3 def say_greeting(greeting, name):
4     print(greeting, name, "I hope you're having a good day")
5
6 say_greeting(#first position, #section position)
```

The first position gets mapped to variable name of `greeting` inside the body of the `say_greeting` function, while the second position gets mapped to `name`.

Optional/Default/Positional arguments

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Sometimes when creating a function we may want to use default arguments, these are arguments that are used if the call to the function does not specify what their value should be. For example.

```
7 def say_greeting(name, greeting="Hello"):
8     print(greeting, name, "I hope you're having a good day")
9
10 say_greeting("John")
11 say_greeting("John", "Hi") # supply greeting as positional
    ↪ argument
```

Results:

```
# => Hello John I hope you're having a good day
# => Hi John I hope you're having a good day
```


Optional/Default/Positional arguments

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

Note if you supply a default argument in the function definition, all arguments after this default argument must also supply a default argument.

So, this **won't** work:

```
4 def say_greeting(name="Jane", greeting):
5     print(greeting, name, "I hope you're having a good day")
6
7 say_greeting("John", "Hi")
```

Recap on arguments

Programming Level-up

Jay Morgan

Introduction

Course
introduction
Contact
information

Python

Introducing
Python
Types of data
Working with
strings
Compound data
structures
Conditional
expressions
Iteration
Functions

Exercise

Library system

```
8  # defining the function
9
10 def say_greeting(name, greeting) # no default arguments
11 def say_greeting(name, greeting="Hello") # greeting is a default
    ↪ argument
12 def say_greeting(name="Jane", greeting="Hello") # both arguments
    ↪ have a default
13
14 # calling the functions
15
16 say_greeting("John", "Hi") # both arguments are provided by
    ↪ position
17 say_greeting(name="John", greeting="Hi") # arguments are
    ↪ supplied by name
18 say_greeting(greeting="Hi", name="John") # the position of named
    ↪ arguments do not matter
```

Function doc-strings

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

To make it clear for a human to quickly understand what a function is doing, you can add an optional doc-string. This is a string that is added directly after the initial definition of the function:

```
19 def my_function(x, y):  
20     """I am a docstring!!!"""  
21     return x + y
```

Some common use cases for docstrings are explaining what the parameters are that it expects, and what it returns.

Multi-line docstrings

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

If your explanation is a little longer than a line, a multiline docstring can be created as long as you're using `"""` three quotation marks either side of the string

```
22 def my_function(x, y):
23     """
24     This is my realllly long docstring
25     that explains how the function works. But sometimes
26     its best not to explain the obvious
27     """
28     return x + y
```

Understanding scope

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

In this example we have two scopes which can be easily seen by the indentation. The first is the *global* scope. The second scope is the scope of the function. The scope of the function can reference variables in the larger scope. But once the function scope exits, we can no longer reference the variables from the function.

```
29 x = 10
30
31 def compute_addition(y):
32     return x + y
33
34 print(compute_addition(10))
35 print(x)
36 print(y)  # does not work
```

Results:

```
# => 20
```

```
# => 10
```

Understanding scope

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Even though we can reference the global scope variable from the scope of the function, we can't modify it like this:

```
4 x = 10
5
6 def compute_addition_2(y):
7     x = x + 5 # error local variable referenced before
8             ↪ assignment
9     return x + y
10 print(compute_addition_2(10))
```

Understanding scope

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

If we really wanted to reference a variable in a global scope and modify its value, we could use the `global` keyword. Doing this makes the function output something different every time it is called. This can make it difficult to debug incorrect programs.

```
11 x = 10
12
13 def compute_addition_2(y):
14     global x
15     x = x + 5
16     return x + y
17
18 print(compute_addition_2(10))
19 print(x)
20 print(compute_addition_2(10))
```

Results:

```
# => 25
```

```
# => 15
```

```
# => 30
```

Understanding scope

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

In almost all cases, avoid using global variables. Instead pass the variables as parameters. This can reduce a source of potential errors and ensure that if a function is called multiple times, the output can be more consistent and expected.

```
5 x = 10
6
7 def compute_addition_3(x, y):
8     x = x + 5
9     return x + y
10
11 print(compute_addition_3(x, 10))
12 print(x)
13 print(compute_addition_3(x, 10))
```

Results:

```
# => 25
# => 10
# => 25
```


Use what you've learnt!

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

We're going to create a library system to help locate and lookup information about books. For example, we want to know the author of book called 'Moby Dick'.

To create this system, we are going to do it in stages. First, we will want to create our database of books:

Title	Author	Release Date
Moby Dick	Herman Melville	1851
A Study in Scarlet	Sir Arthur Conan Doyle	1887
Frankenstein	Mary Shelley	1818
Hitchhikers Guide to the Galaxy	Douglas Adams	1879

Our database is going to be a list of dictionaries. Where each dictionary is a row from this table. For example, one of the dictionaries will have the key "title" and a value "Moby Dick".

Create this database and call it db

Locating Books

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

- Create a function called `locate_by_title` that takes the database to look through, and the title to look up as arguments.
- This function should check each dictionary, and if the title is the same as what was searched for, it should return the whole dictionary.
- Test this function by calling the `locate_by_title` function with `db` and `"Frankenstein"`. You should get `{"title": "Frankenstein", "author": ...}`.

Note you should include docstrings to describe the arguments to the function, and what it will return.

Selecting a subset

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

Now that we can find books by the title name, we also want to find all books that were released after a certain data.

- Create a function called `books_released_after` that takes two arguments: the database to look through, and the year.
- This function should look through the database, if it finds a book that was released after the year, it should add it to a list of books that is returned from this function.
- Test this function by calling `books_released_after` with `db` and `1850`. This function call should return a list containing three dictionaries. The first entry should be 'Moby Dick' and the section should be 'A Study in Scarlet', etc.

Updating our database

Programming Level-up

Jay Morgan

Introduction

Course introduction

Contact information

Python

Introducing Python

Types of data

Working with strings

Compound data structures

Conditional expressions

Iteration

Functions

Exercise

Library system

Oh no! 'Hitchhikers Guide to the Galaxy' was released in 1979 not 1879, there must have been a typo. Let's create a function to update this.

- Create a function called `update`, that takes 5 arguments: 1) the database to update, 2) the key of the value we want to update 3) the value we want to update it to 4) the key we want to check to find out if we have the correct book and 5) the value of the key to check if we have the correct book.

```
5 update(db,  
6     key="release year",  
7     value=1979,  
8     where_key="title",  
9     where_value="Hitchhikers Guide to the Galaxy")
```

Extended exercise

Programming Level-up

Jay Morgan

Introduction

Course
introduction

Contact
information

Python

Introducing
Python

Types of data

Working with
strings

Compound data
structures

Conditional
expressions

Iteration

Functions

Exercise

Library system

- In the previous steps we created functions `locate_by_title` and `books_released_after`. These two functions are similar in a way that they are selecting a subset of our database (just by different criteria).
- For this harder exercise, can we create a single function called `query` that allows us to do both `locate_by_title` and `books_released_after`.
- An example call to this query function may look like:

```
10 results = query(db,  
11     where_key="title",  
12     where_value="Moby Dick",  
13     where_qualifier="exactly")
```

- `where_qualifier` should accept strings like "exactly", "greater than", and "less than".