Python Fundamentals

Variables

{codenation}®

Learning Objectives

- To use variables and operators to store values and do calculations
- To use snake_case when naming variables
- To access data in variables

First Things First!

First things first!

"All Around The World"

Take this string.
Write a program which prints just the 8th character in uppercase to the terminal.

First things first!

```python
print("All around the world"[7].upper())

# or

print("All around the world".upper()[7])
```

Which is more efficient and why?

Boxes

Boxes

Boxes are handy!

We store things in boxes to retrieve later.
We can add or remove things from the box.
We can access the contents when we need.

We can do something similar in coding with variables.

Variables

Variables

Variables allow us to store data to a name, access that data via the name, and place new data inside the variable whenever we need.

This means we can write code to work with data without knowing the value of the data – which is much easier!

Variables

Variables allow us to reuse code.

Imagine a cash machine.
What data does it need from the user to work?

Variables

Withdraw: £100
From: 12345678
Confirm pin: 1234

becomes

Withdraw: amount
From: account_number
Confirm pin: pin_number

Creating a Variable

Creating a variable

```
1 my_name="Dave"
```

To create a **variable**, we define the name we want to refer to the data by. Then we assign the data to the name using **=**, an **assignment operator**.

What data type is this example?

Creating a variable

```
my_name="Dave"

print(my_name)
```

To **access** the data inside the variable, we use the variable name.

Creating a variable

```
my_name="Dave"

print(my_name)
```

Prints out whatever the value of my_name is – very reusable!

```
my_name="Dave"

print("Dave")
```

Only ever prints "Dave"

Creating a variable

```
my_name="Dave"
my_age = 50
student = True
```

Variables can be any data type – and we don't need to tell Python what data type we're using.

Creating a variable

```
1 my_name="Dave"
2 my_age = 50
3 student = True
4
5 student = False
```

You can **update** the value of a variable.
On lines 3 + 4, student is **True**.
From line 5 onwards, it is **False**.

Creating a variable

```
my_name="Dave"
my_age = 50
student = True

student = False
```

Have you noticed anything about the way we name our variables?

Creating a variable

```
my_name="Dave"
my_age = 50
student = True

student = False
```

Clear, explanatory names.
Underscores between words instead of spaces.
All lower case.

Python uses snake_case as its naming convention.

Naming Conventions

Naming conventions

Naming conventions follow coding best practise and enhance code readability.

We should write code that is very easy for us, and other developers to understand.

Putting Variables in Context

Putting variables in context

```
fav_drink="hot chocolate"

print(fav_drink)
```

So far we've learnt what a variable is, how to create variables, how to assign data to a variable, and how to access the data – but what about using the variable in some context?

Putting variables in context

```
fav_drink="hot chocolate"

print("My favourite drink is ", fav_drink)
```

We can include variables by including them in our **print function**.
We would split it from the string with a comma.

Putting variables in context

```
fav_drink="hot chocolate"

print("My favourite drink is " + fav_drink)
```

We can use the + symbol to concatenate strings together.
This only works with strings, however.
The + symbol has another meaning when used with integer and floating data types.

Putting variables in context

```
my_name = "Dave"
fav_drink="hot chocolate"

print("My name is {}, my favourite drink is {}".format(my_name,fav_drink))
```

We can use the .format() method to create outputs.
In our string, we would use {} as a placeholder for the variable.

Putting variables in context

```
my_name = "Dave"
fav_drink="hot chocolate"

print(f"My name is {my_name}, my favourite drink is {fav_drink}")
```

We can use an **f-string** to create very readable code.

f-string

These are all good methods!

An f-string is the newest method – and generally used over .format().

Input

Input

So far, we've been hard coding our variables in, but this isn't usually how code works.

Our variable data might be generated in runtime, it might change as a result of something, or we might get the data from a user.

input() is a function that we can use for this.

Input

```
1 input("Type your name here: ")
```

Input allows us to write a prompt to the user, and then the terminal will wait for the user's response before continuing execution.

But where does that response go?

Input

```
1 user_name = input("Type your name here: ")
2
3 print(f"Hello {user_name}")
```

If we save the user's response as a variable, we can reuse their answer again and again!

Input

```python
user_name = input("Type your name here: ")

print(f"Hello {user_name}")
```

What the user types is going to be a string by default.
Even if the user typed in 1234, user_name would be the string "1234", not the integer 1234.

Working with Integers

Working with integers

```
my_name="Dave"
```

An = is an assignment operator.
It is assigning the string “Dave” to the variable my_name.
There are many other operators.

Arithmetic operators

+	Addition
-	Subtraction
*	Multiplication
**	Exponential
/	Division
%	Modulo (Remainder)

```python
print(1+2)
# Result 3

print(10-4)
# Result 6

print(4*6)
# Result 24

print(3**3)
# Result 27

print(15/5)
# Result 3

print(15%5)
# Result 0 - the remainder of 15/5
```

Arithmetic operators

We can use variables, assignment operators and arithmetic operators to create useful code.

```
balance = 100
amount_withdrawn = 50

balance = balance - amount_withdrawn

print(balance)

# Result 50

```

Assignment operators

+=
-=
*=
/=

We can combine our assignment operator = with our arithmetic operators.

These operators do the sum and store the value all in one!

Assignment operators

Both of these are updating the value of balance to the result of the sum (line 4)

Which is faster to type?

Which is more readable?

```
1 balance = 100
2 amount_withdrawn = 50
3
4 balance = balance - amount_withdrawn
5
6 print(balance)
7
8 # Result 50
9
```

```
1 balance = 100
2 amount_withdrawn = 50
3
4 balance -=amount_withdrawn
5
6 print(balance)
7
8 #Result 50
```

Casting

Casting

There will be some cases where we need the user to provide us with a specific data type – working with integers is a good example of this.

Casting

```
1 print("Type in two numbers to multiply them")
2
3 num1 = input("Number 1:   >  ")
4 num2 = input("Number 2:   >  ")
5
6 print(num1*num2)
```

Here is a small program which takes two inputs from a user and multiplies them.

Casting

```
1 print("Type in two numbers to multiply them")
2
3 num1 = input("Number 1:   >  ")
4 num2 = input("Number 2:   >  ")
5
6 print(num1*num2)
```

Remember – input is a string by default.
If your user types in "3" and "4", your program is trying to multiply the character 3 by the character 4 – we want to multiply the integers 3 and 4.

Casting

```
Type in two numbers to multiply them
Number 1:    >    3
Number 1:    >    4
Traceback (most recent call last):
  File "c:\Users\FiercePC\Documents\final_flask_ref\if.py", line 6, in <module>
    print(num1*num2)
TypeError: can't multiply sequence by non-int of type 'str'
```

Inputting numbers will give us an error.
Luckily, the error message is very clear, and this can help us debug.
This error tells us we can't multiply strings.

Casting

```
1 print("Type in two numbers to multiply them")
2
3 num1 = int(input("Number 1:   >  "))
4 num2 = int(input("Number 2:   >  "))
5
6 print(num1*num2)
```

We can cast the input and change the data type!
On lines 3 and 4, we have wrapped our inputs in the int() constructor.

Casting

```
1 print("Type in two numbers to multiply them")
2
3 num1 = int(input("Number 1:   >  "))
4 num2 = int(input("Number 2:   >  "))
5
6 print(num1*num2)
```

When the user submits a response, the program will build that response as an integer instead!

Casting

```
int()
# constructs an integer number from an integer, a float (by removing all decimals), or a string (providing the string represents a whole number)

float()
# constructs a float number from an integer, a float, or a string (providing the string represents a float or an integer)

str()
# constructs a string from a wide variety of data types, including strings, integers and floats

```

We can use these constructors to manipulate data into the type we need, if it doesn't automatically get made that way!

Learning Objectives

- To use variables and operators to store values and do calculations
- To use snake_case when naming variables
- To access data in variables

Activity 1

Create a program which asks a user their name, age, and favourite colour.

Print these in a sentence using an f string.

Activity 2

Create a program which accepts two inputs from a user (num1 and num2), use these inputs with each operator (+, -, /, *, **, %).

Print the equation and the output.

Activity 3

A shop sells apples for 25p per apple.

Create a program which asks a user how many apples they want to buy.

Display the total cost of the apples in both pence, and pounds and pence.

Activity 3: Stretch

10 apples would cost £2.50
Your program will say £2.5

Research how to have your answer formatted to have two decimal places.