Variables and Data types

We use variables all the time, so this is a short recap on what they are. Variables are simply provide names for a value. They act as identifiers for a specific value, which we can use throughout our program. This is how we do this:

```python
x = 5
```
← This says, let x be equal to 5

Python is what we call a dynamically-typed language. This means that we do not have to specify in advance if a variable is going to be equal to a string or an integer and Python will figure this all out for us. This does allow us to do some unique things, like the following:

```python
x = 5
print(x)
x = "Hello World"
print(x)
```
→
```
5
Hello World
```

Due to Python being dynamically typed, it allows us to assign multiple different values of different types to the same variable. If you choose to progress with computer science, you will see that not every programming language allows this.

Data Types

We have ran into 2 data types mostly over the past few weeks. However, there are many more! This is a list of the more basic types:

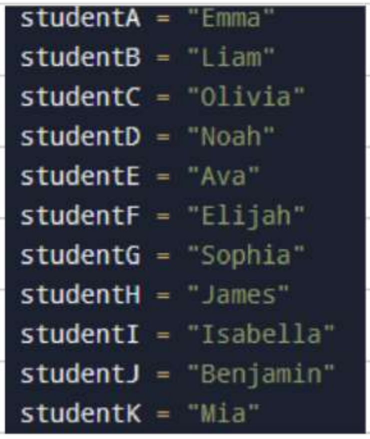
Type	Meaning	Examples	
Integers →	These are whole numbers →	`x = 5` `print(type(x))` →	`<class 'int'>`
Float →	These are decimal numbers →	`x = 5.0` `print(type(x))` →	`<class 'float'>`
String →	These are anything enclosed within " " →	`x = "Hello World"` `print(type(x))` →	`<class 'str'>`
Char →	Character; a single letter/number/symbol →	`x = 'H'` `print(type(x))` →	`<class 'str'>` *
Bool →	Boolean; this is either true or false →	`x = True` `y = False` `print(type(x))` `print(type(y))` →	`<class 'bool'>` `<class 'bool'>`

*Python, officially speaking, treats all chars (characters) as strings. However, it is useful to know this difference as this does not apply to all programming languages.

Note: you will see that I used type(x) a lot. All this does is return the type of a variable (the 'class' it belongs to)

Despite the above data types being the backbone of everything that we will do, the truth is that they're also slightly limited on what can be done with them. Let's consider the following example:

Imagine you're trying to write out the name of every student in your class, and you'd like to store all of their names so that you can access them at any time. From what we know already, we would have to do something like this:



```
studentA = "Emma"
studentB = "Liam"
studentC = "Olivia"
studentD = "Noah"
studentE = "Ava"
studentF = "Elijah"
studentG = "Sophia"
studentH = "James"
studentI = "Isabella"
studentJ = "Benjamin"
studentK = "Mia"
```

Eventhough this works, imagine we had 100 students - creating a variable for each one would be cumbersome and a waste of precious computer resources! This is where something called data structures will help us solve these types of problems! Let's look into them!

Data Structures

As the name suggests, data structures are structures that hold data - similar how a paragraph (a structural component when writing) holds words (data). There are a few different data structures, but we will only be focusing on 4. This may seem like a lot, but they are very simple to understand!
In this part, we will only be focusing on 2.

1. Lists

We have mentioned lists here and there, but never concretely. Now it is the time to do so!
A list is, as the name suggests, a container for storing a sequence of values. For example, a shopping list stores a sequence of grocery items to be bought! The values in this list can all be of different type - i.e. a list can have some integers, some strings, or even another list inside it! Let's see how we can make a list:

```python
students = []
```

Try printing type(students) to see

This is officially a list! To create a list, all we have to do is put two square brackets: [ ].
However, a list with no values is not particularly useful. Lets see how we can add data to it:

```python
students = ["Emma", "Liam", "Olivia", "Noah", "Ava", "Elijah", "Sophia", "James", "Isabella",
    "Benjamin", "Mia"]
```

That is all! We just have to write whatever we want in this list and seperate each item by a comma. This way, we stored all of the information we needed to, in just one variable. Now, you may ask, how do we use a list, let's find out!

Accessing a list

We use indexes to access a particular element in a list. Each element in a list has an index, starting from 0. So, in our case, "Emma" is at index 0, "Liam" is at index 1 and so on. This is because "Emma" is 0 spaces away from the start, "Liam" is one space away from the start, and so on so forth. To access these in code, we simply have to do the following:

```python
students[0]
```

The index we want

We enclose the index in these brackets

We make reference to the variable name storing the list

Lets print this out and see what it returns:

```python
students = ["Emma", "Liam", "Olivia", "Noah", "Ava", "Elijah", "Sophia", "James", "Isabella",
    "Benjamin", "Mia"]
print(students[0])
```

→

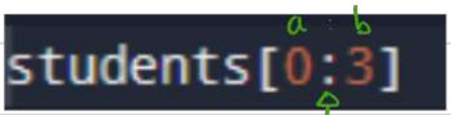
```
Emma
```

Success! Try running this example with other indexes (even negative values) and see what is happening!

Accessing multiple indexes

We can also access multiple indexes at once. We do this by:

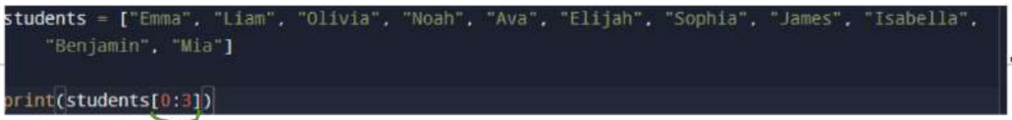
a : b



```
students[0:3]
```

NOTE
a<b or else this will not work!

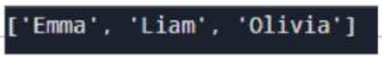
This colon says, slice the list in the students variable from index a to index b-1. That is, the values in index 0, 1 and 2 are returned. Note that this is called slicing a list, and it will return to you a new list with just the indexes you ask for. Let's see:



```
students = ["Emma", "Liam", "Olivia", "Noah", "Ava", "Elijah", "Sophia", "James", "Isabella",
    "Benjamin", "Mia"]

print(students[0:3])
```

$\rightarrow$

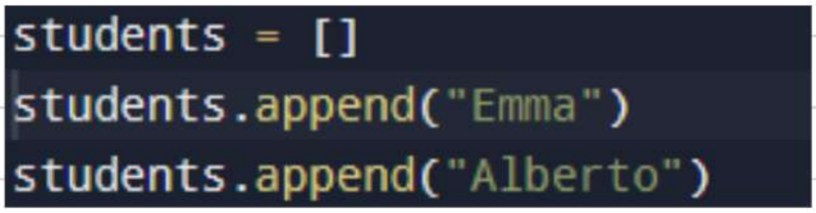


```
['Emma', 'Liam', 'Olivia']
```

Try using negative numbers here and see what happens!

Adding to a list

In Python, adding to a list is called appending. Conveniently, the method to add to a list is called exactly that - append! Lets see how to use it:



```
students = []
students.append("Emma")
students.append("Alberto")
```

our paragraph with instructions on how to add to a list

students.append("Emma")

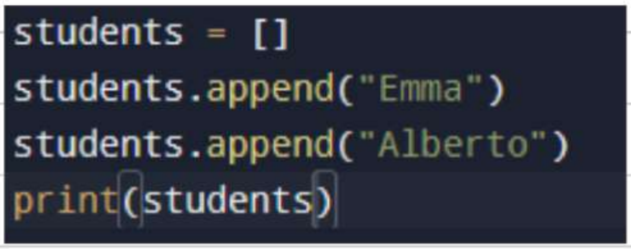
variable name (type: list)

This says, in the list chapter, look for the paragraph after the dot

what we want to add to the list

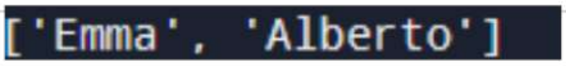
Look back at last week's content if you are unsure on what exactly is going on!

After having run this, we can print out the list and see what we get:



```python
students = []
students.append("Emma")
students.append("Alberto")
print(students)
```

$\longrightarrow$

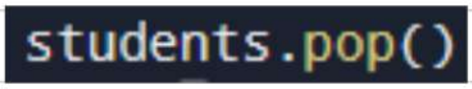


```
['Emma', 'Alberto']
```

Success! Our students list now has 2 elements in it.

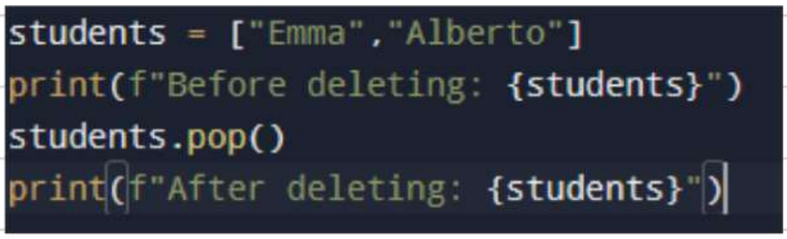
Deleting from a list

Deleting from a list in Python is called 'popping', similar to how when you pop a bubble, it disappears. We do this by writing:



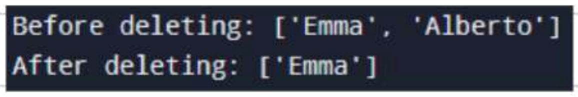
```python
students.pop()
```

Lets see it in action:



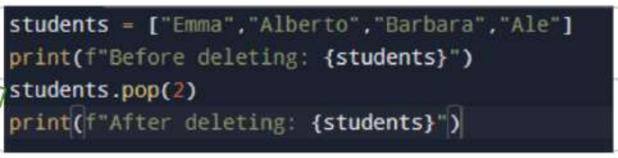
```python
students = ["Emma","Alberto"]
print(f"Before deleting: {students}")
students.pop()
print(f"After deleting: {students}")
```

$\longrightarrow$



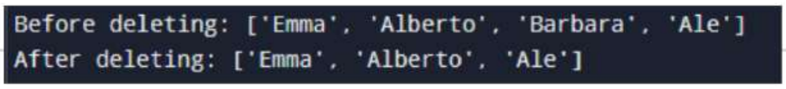
```
Before deleting: ['Emma', 'Alberto']
After deleting: ['Emma']
```

By default, if we do not specify, Python will pop the LAST element in a list. As you saw, this meant that "Alberto" was removed from our list of students. However, we can choose to delete specific indexes too, by specifying:

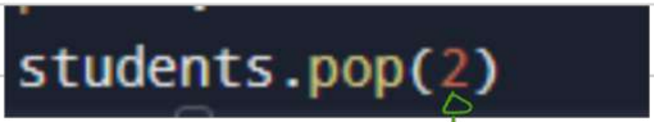


```python
students = ["Emma","Alberto","Barbara","Ale"]
print(f"Before deleting: {students}")
students.pop(2)
print(f"After deleting: {students}")
```

$\longrightarrow$

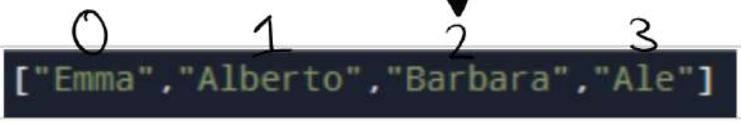


```
Before deleting: ['Emma', 'Alberto', 'Barbara', 'Ale']
After deleting: ['Emma', 'Alberto', 'Ale']
```



```python
students.pop(2)
```

This 2 refers to the index number 2. That is:

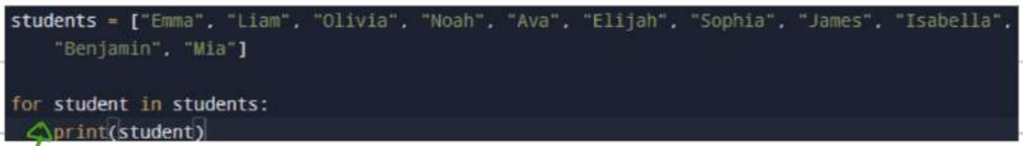


```
 0        1          2         3
["Emma","Alberto","Barbara","Ale"]
```

which is why "Barbara" was popped off our list.

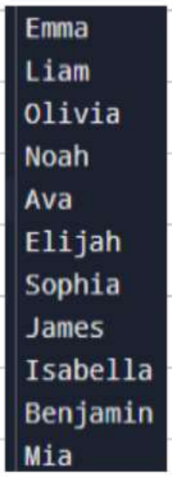
Example 1: Printing every item in a list

There are many different ways to do this, but I will show you the two most-used ways:



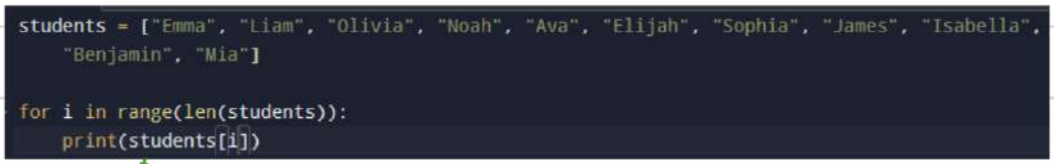
```
students = ["Emma", "Liam", "Olivia", "Noah", "Ava", "Elijah", "Sophia", "James", "Isabella",
    "Benjamin", "Mia"]

for student in students:
    print(student)
```



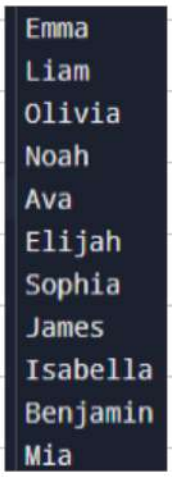
```
Emma
Liam
Olivia
Noah
Ava
Elijah
Sophia
James
Isabella
Benjamin
Mia
```

This loop, as we have seen similarly before, is behind the scenes going through each index, and assigning the contents to the student variable. This is useful when you do not need to know the indexes for any specific purpose. We could also do the following:



```
students = ["Emma", "Liam", "Olivia", "Noah", "Ava", "Elijah", "Sophia", "James", "Isabella",
    "Benjamin", "Mia"]

for i in range(len(students)):
    print(students[i])
```



```
Emma
Liam
Olivia
Noah
Ava
Elijah
Sophia
James
Isabella
Benjamin
Mia
```

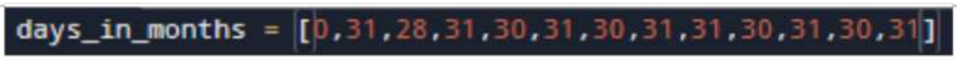
This method takes advantage of the accessing techniques that we learnt before. You may also see the 'len(students)', which we haven't yet gone through, but will be very shortly.

You will remember that I describe often a String as a list of characters. Though officially this may not be the best definition, it perfectly describes strings! For example, try running the examples above with a String rather than a list, and you will see that it will behave and perform in exactly the same way!

It is also important for you to know that a list is mutable. That means that you can change it in whichever way you like, by changing or adding or removing elements.

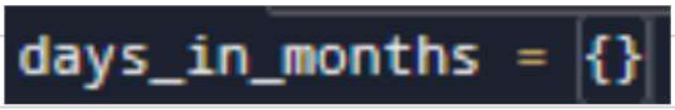
2. Dictionaries

Dictionaries in Python are the same as a dictiona in real life - ou look up a word (a key) and in return, you get its definition (a value). This is particularly useful where you want to give data some meaning, that without it, it does not make sense. For example:



```python
days_in_months = [0,31,28,31,30,31,30,31,31,30,31,30,31]
```

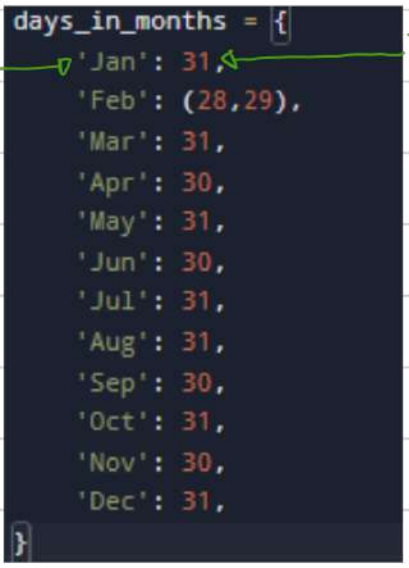
Eventhough you as a programmer may know what this is saying, it is unclear and can be interpreted in different ways. As such, a dictionary helps to solve this problem by giving meaning to each value. This is how it is done:



```python
days_in_months = {}
```

For a list, you saw that we used the square brackets [ ]. For a dictionary, we use curly brackets as seen above { }. However, as before with lists, this dictionary is currently empty. We can populate it as follows:

This is a key

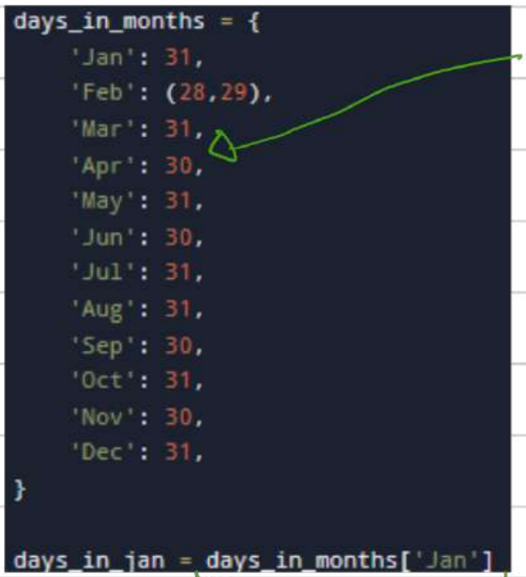


```python
days_in_months = {
    'Jan': 31,
    'Feb': (28,29),
    'Mar': 31,
    'Apr': 30,
    'May': 31,
    'Jun': 30,
    'Jul': 31,
    'Aug': 31,
    'Sep': 30,
    'Oct': 31,
    'Nov': 30,
    'Dec': 31,
}
```

This is a value

As you can see above, to populate the dictionary, we need a key and a value. We denote that a value belongs to a specific key using : . For example, 31 belongs to the key 'Jan'. Each key and value are seperated by a comma, as in the list. Lets see how we can access the values:

Accessing a value

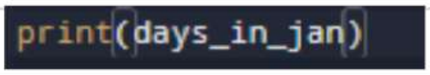


```
days_in_months = {
    'Jan': 31,
    'Feb': (28,29),
    'Mar': 31,
    'Apr': 30,
    'May': 31,
    'Jun': 30,
    'Jul': 31,
    'Aug': 31,
    'Sep': 30,
    'Oct': 31,
    'Nov': 30,
    'Dec': 31,
}

days_in_jan = days_in_months['Jan']
```

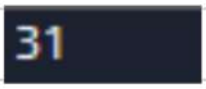
NOTE: Keys can be any data type you want - integers, strings etc

You can see that this process is similar to that of accessing an index in a list. We use the variable name of the dictionary, followed by two square brackets, and inside of them we put instead of a number like in lists, the key whose value we want. So, now lets print it to the screen:



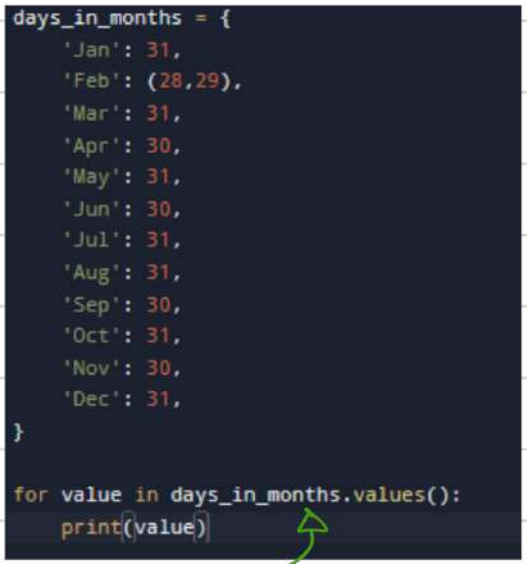
```
print(days_in_jan)
```

$\longrightarrow$



```
31
```

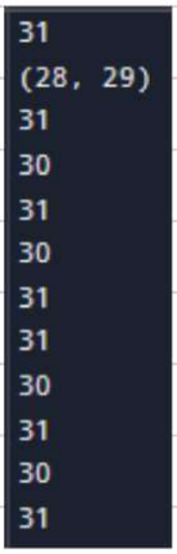
Lets say we wanted to print all of the values in our dictionary, how would we do that? We need to we the values() function, which is done as follows:



```
days_in_months = {
    'Jan': 31,
    'Feb': (28,29),
    'Mar': 31,
    'Apr': 30,
    'May': 31,
    'Jun': 30,
    'Jul': 31,
    'Aug': 31,
    'Sep': 30,
    'Oct': 31,
    'Nov': 30,
    'Dec': 31,
}

for value in days_in_months.values():
    print(value)
```

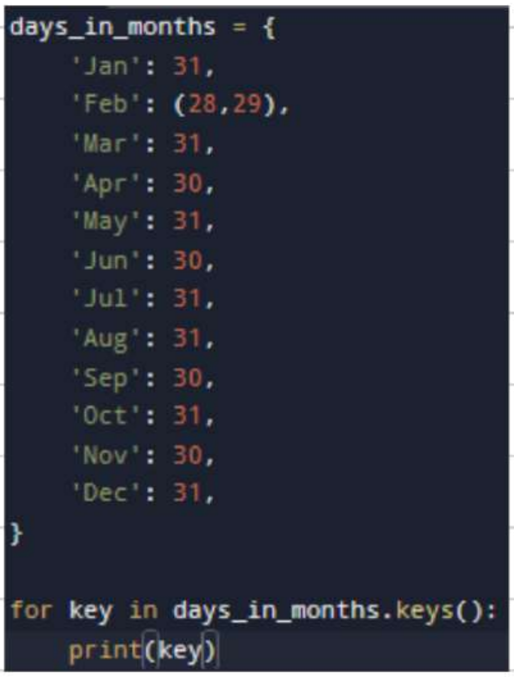
$\longrightarrow$



```
31
(28, 29)
31
30
31
30
31
31
30
31
30
31
```

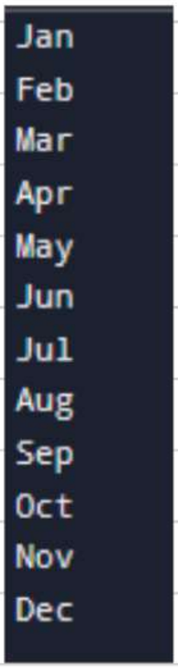
Here, the values() function is returning a list, which is why this for loop behaves like the examples we saw above.

If we wanted just the keys, we can also do that using the keys() method, as follows:



```
days_in_months = {
    'Jan': 31,
    'Feb': (28,29),
    'Mar': 31,
    'Apr': 30,
    'May': 31,
    'Jun': 30,
    'Jul': 31,
    'Aug': 31,
    'Sep': 30,
    'Oct': 31,
    'Nov': 30,
    'Dec': 31,
}

for key in days_in_months.keys():
    print(key)
```

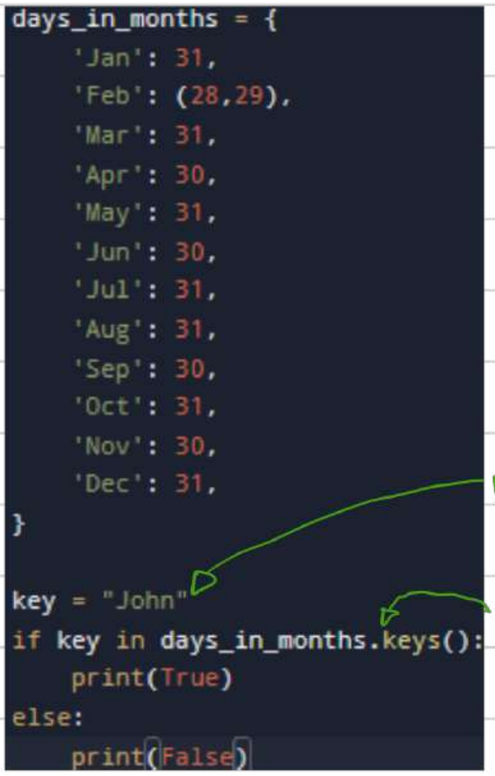


```
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
```

There is also a way to return both at the same time, but we will look into that the next time!

Checking if a key exists

Lets say we weren't sure if a key exists. How could we check if it does? Lets see:

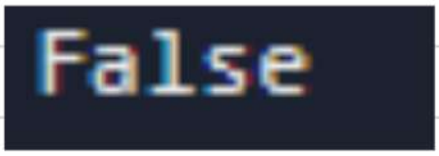


```
days_in_months = {
    'Jan': 31,
    'Feb': (28,29),
    'Mar': 31,
    'Apr': 30,
    'May': 31,
    'Jun': 30,
    'Jul': 31,
    'Aug': 31,
    'Sep': 30,
    'Oct': 31,
    'Nov': 30,
    'Dec': 31,
}

key = "John"
if key in days_in_months.keys():
    print(True)
else:
    print(False)
```

We create a variable assigned with the key to check

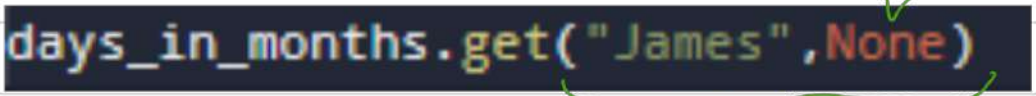
We use the keys method to check, using the 'in' keyword



```
False
```

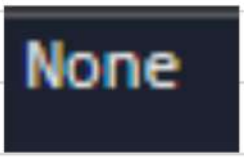
Alternatively, the nicer way of doing this is to use the get method. This lets us do something similar to both things above, all in one line:

None is a special keyword in Python.



```
days_in_months.get("James",None)
```

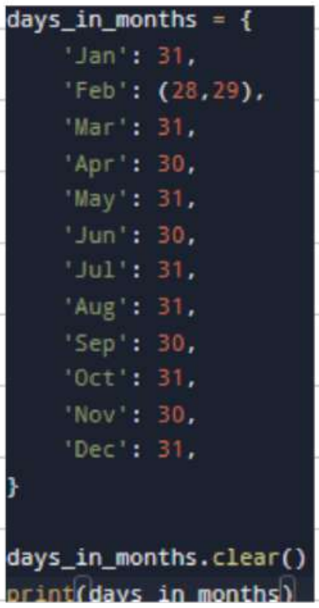
The get method takes in two arguments. The first, is the key you would like to check for. The second, is what will be returned if the key does not exist. So, in this case, when we print the result of this, we get:



```
None
```

Removing everything in a dictionary

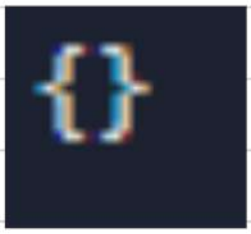
This is done using the clear() method, as follows:



```
days_in_months = {
    'Jan': 31,
    'Feb': (28,29),
    'Mar': 31,
    'Apr': 30,
    'May': 31,
    'Jun': 30,
    'Jul': 31,
    'Aug': 31,
    'Sep': 30,
    'Oct': 31,
    'Nov': 30,
    'Dec': 31,
}

days_in_months.clear()
print(days_in_months)
```

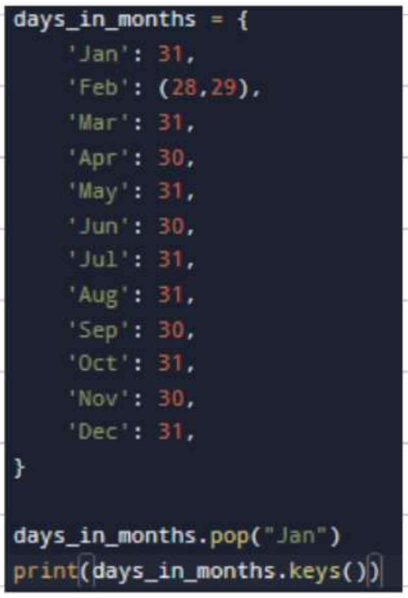
$\longrightarrow$



```
{}
```

Removing values

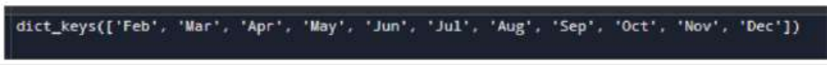
If we want to remove just one value, we can use the pop() method just like we did with lists. However, as with accessing the values, we make reference to what we want to delete using the key name. For example:



```python
days_in_months = {
    'Jan': 31,
    'Feb': (28,29),
    'Mar': 31,
    'Apr': 30,
    'May': 31,
    'Jun': 30,
    'Jul': 31,
    'Aug': 31,
    'Sep': 30,
    'Oct': 31,
    'Nov': 30,
    'Dec': 31,
}

days_in_months.pop("Jan")
print(days_in_months.keys())
```

$\longrightarrow$

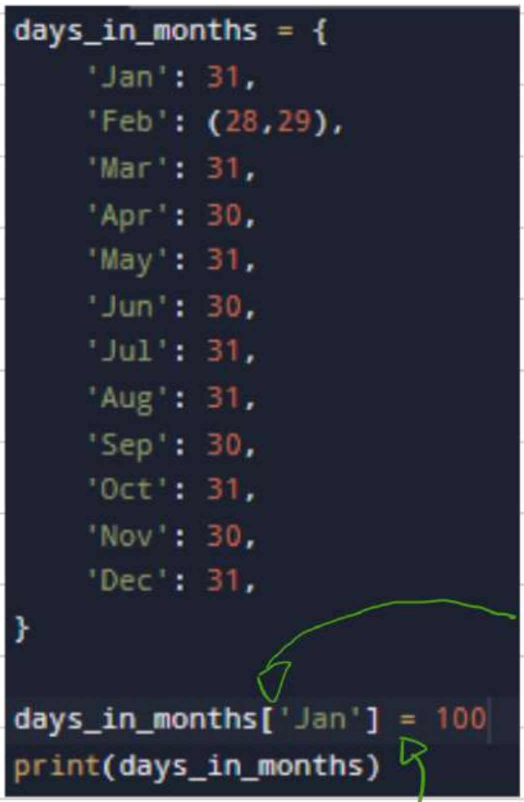


```
dict_keys(['Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])
```

Try popping a key that doesn't exist. What happens?

Updating existing values

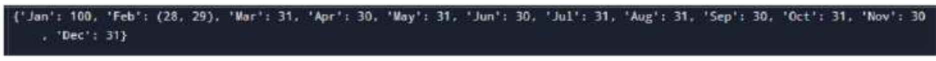
This is the same as in lists, but we use the key rather than an index.



```python
days_in_months = {
    'Jan': 31,
    'Feb': (28,29),
    'Mar': 31,
    'Apr': 30,
    'May': 31,
    'Jun': 30,
    'Jul': 31,
    'Aug': 31,
    'Sep': 30,
    'Oct': 31,
    'Nov': 30,
    'Dec': 31,
}

days_in_months['Jan'] = 100
print(days_in_months)
```

$\longrightarrow$



```
{'Jan': 100, 'Feb': (28, 29), 'Mar': 31, 'Apr': 30, 'May': 31, 'Jun': 30, 'Jul': 31, 'Aug': 31, 'Sep': 30, 'Oct': 31, 'Nov': 30
    , 'Dec': 31}
```

here, we are saying, let the key "Jan" have the value 100.

Try writing something like this for a key that doesn't exist. What happens?

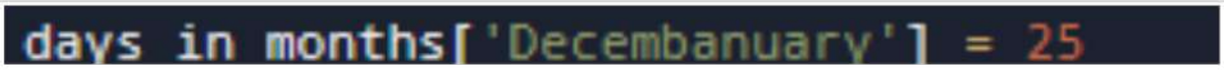
↳ P.S. You will learn how to add a new key and value!

Adding to a dictionary

As mentioned above, one way to add to a dictionary is by using the same process as above. That is:

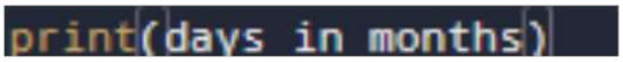
dictionaryVariableName [keyToAdd] = value

Therefore to add a new month to our dictionary in our example, we will say:



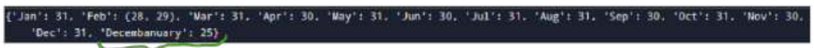
```
days_in_months['Decembanuary'] = 25
```

If we now print this out to confirm:



```
print(days_in_months)
```

→



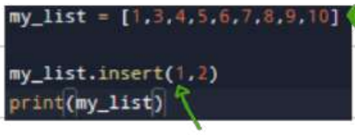
```
{'Jan': 31, 'Feb': (28, 29), 'Mar': 31, 'Apr': 30, 'May': 31, 'Jun': 30, 'Jul': 31, 'Aug': 31, 'Sep': 30, 'Oct': 31, 'Nov': 30,
    'Dec': 31, 'Decembanuary': 25}
```

Our fictitious month is now in the dictionary!

Going back to lists

You will remember I mentioned the len( ) method, and the time has come to explain this. Lists support various methods, but these are the most impotant ones alongside those that were previously explained:

insert(index, value): This allows you to add an element at a specific index.

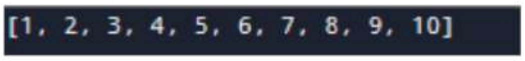


```
my_list = [1,3,4,5,6,7,8,9,10]

my_list.insert(1,2)
print(my_list)
```

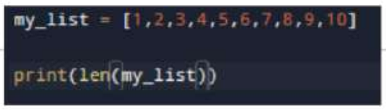
The 2 is missing!

remember indexes stat from 0!



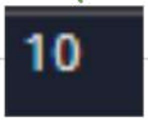
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

len(item): This returns you the length of something



```
my_list = [1,2,3,4,5,6,7,8,9,10]

print(len(my_list))
```



```
10
```

You will be using this pretty much all the time! It is extremely useful! You can use it for lists, or for strings! Make sure you're comfortable with this!

Your Exercises

Exc. 1: List less than 6 (Testing: functions, for loops, conditional statements, list operations)

Take a list like the following (or make your own):

```
num_list = [1, 1, 2, 3, 5, 8, 6, 15, 21]
```

Create a function, less_than_6(list_to_check) that returns the number of digits in the list that are less than 6. Print this number to the screen.

Extension: if a number is less than 6, add it to a new list and return this list too.

Exc. 2: The Egg Farmer (Testing: functions, arithmetic operation) → leave this until last! It tests last lesson's concepts

An egg farmer picks up eggs every morning to sell in the market. The farmer puts all the eggs he picks up into egg cartons, each with 12 eggs or with 6. For example:

- The farmer picks up 128 eggs. This means they will use 10 cartons of 12 (12×10=120), leaving them with 8. They will then use 1 carton of 6, leaving them with 2 spare eggs. The farmer will have these 2 spare eggs for breakfast!

Your task is to write a program that will:

- ask the farmer to enter the number of eggs picked up
- work out how many boxes of 12 will be needed and return this
- work out if the farmer will also need a box of 6 and return this (if necessary)
- finally, return the number of spare eggs that they can have for breakfast.

Exc. 3: Add number to each element in list (Testing: lists, for loops)

Create a list with numbers - this can be any amount of numbers, of any value. Write a program to add a number to each element in the list. That is:

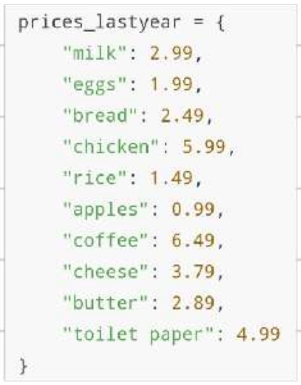
- take an input from the user (this is what you will add to each number in the list)
- return the list that has been added to

Exc 4: Birthdays (Testing dictionaries)

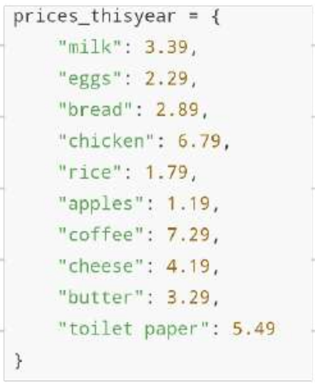
Create a dictionary of the form name: birthday, where the name is a key, and the birthday is the value. Put in the names and birthdays of a few people that you know. Then, ask the user to enter a name. If this person is in the dictionary, return their birthday. If they're not on the dictionary, ask the user again to input their birthday, and add this new person to the dictionary.

Exc 5: Inflation calculator (Testing: dictionaries, lists, arithmetic operations)

Given the following two dictionaries, prices_lastyear & prices_thisyear, work out the percentage difference in the prices of a shopping basket (this is what is generally described as inflation; specifically, something called CPI). These are the 2 dictionaries, and this is a simplified version of how to calculate inflation.



```
prices_lastyear = {
    "milk": 2.99,
    "eggs": 1.99,
    "bread": 2.49,
    "chicken": 5.99,
    "rice": 1.49,
    "apples": 0.99,
    "coffee": 6.49,
    "cheese": 3.79,
    "butter": 2.89,
    "toilet paper": 4.99
}
```



```
prices_thisyear = {
    "milk": 3.39,
    "eggs": 2.29,
    "bread": 2.89,
    "chicken": 6.79,
    "rice": 1.79,
    "apples": 1.19,
    "coffee": 7.29,
    "cheese": 4.19,
    "butter": 3.29,
    "toilet paper": 5.49
}
```

I would like you to:

1 - given each item in the dictionary, find the percentage change between this year and last year's prices for each item.

↳ reminder: to calculate % change = $\frac{\text{new value} - \text{old value}}{\text{old value}} \times 100$

2 - Once you have worked this out, append the % change to a list, called percentage_changes

3 - repeat steps 1 and 2 for all items in the dictionary

4 - Once that is complete, using the values we stored in percentage_changes, work out the average of all these values, and return this as our figure for inflation.

Hints

General hint: Break the problem down into smaller, more manageable pieces. Programming is a lot like cooking, we have to go step by step!

Exc 1

You get no hints for this one! :)

Exc 2

Hint 1: We need to first check for cartons of 12. We can do this by: number of eggs // 12. The '//' is the floored division operator. This will return the closest divisor between 2 numbers, eg:

```
13/3 = 4      100 // 12 = 8
11/2 = 5
```

We can then use this result to work out how many eggs are left, and repeat this process for cartons of 6.

Hint 2: If we have 130 eggs, then 130 // 12 = 10. Therefore, we will use 10 cartons of 12, leaving us with 130 - (12×10) = 10. Then, we do the same thing with 6, 10//6 = 1, so 10 - (6×1) = 4. So our final answer is 4! Now, just write this into code!

Exc 3

Hint 1: You will need a for loop, and you will need to say something along the lines of:

```
my_list[index] = my_list[index] + value
```

Exc 4

Hint 1: perhaps the .get( ) method will be useful here

Exc 5

This problem can be approached in many different ways. However, all will use functions!

Hint 1: We need to loop between each key, so we should use the for key in dictionary.keys() loop in the example. From this, we can access the values in each dictionary and work out the % difference

Try to break this problem down as much as possible! Write down the steps before you start actually writing code!