More containers

The data structures we studied previously are useful but what if we want to group multiple ones together?

Lists

Probably the most commonly used data structure in Python. Lists simply group multiple variables together. Lists are declared with square brackets `[]` and items within them are separated by commas. We can declare an empty list `x = []` but that's not very useful.

In [ ]:

```
fruits = ["apple", "orange", "tomato", "banana"]  # a list of strings
print(type(fruits))
print(fruits)
```

So why is that so useful? It allows us so to group similar data together and use it and apply a single operation to the whole group rather than applying the same operation multiple times for each item. More on that later.

We can access each individual item within the list by indexing it.

In [ ]:

```
fruits[2]
```

Huh? Why did that print `tomato` instead of `orange`? Is it supposed to print the 2nd item right? Not exactly. A list can be indexed starting from 0. Here is an image to illustrate this better:

What will happen if we want to access a index which is not currently assigned?

In [ ]:

```
fruits[4]
```

If there is no such item, then Python will simply throw out an error. So make sure you are always aware of the size of your list! One way to do this is using a len() which as implied returns the length(size) of any data structure:

In [ ]:

```
len(fruits)
```

Now for the serious question - is a tomato really a fruit? If you don't think so you can change it.

In [ ]:

```
fruits[2] = "apricot"
print(fruits)
```

This means that we can modify our list however we wish. For example to add a new fruit at the end of it and then remove our next not-a-fruit victim.

In [ ]:

```
fruits.append("lime")     # add new item to list
print(fruits)
fruits.remove("orange")   # remove orange from list
print(fruits)
```

That seems useful! Can we do the same with integers? Python actually offers various functions for generating numerical lists. The most useful out of which is:

In [ ]:

```
nums = list(range(10))
print(nums)
```

In [ ]:

```
nums = list(range(0, 100, 5))
print(nums)
```

You can also get only a part of a list you have already created. This is called slicing and can be done in various different ways:

In [ ]:

```
print(nums[0:3]) # Get items 0 through 3
print(nums[4:]) # Get items 4 onwards
print(nums[-1]) # Get the last item
```

Lists can also be used with other functions out of which some are:

In [ ]:

```
print(len(nums))    # number of items within the list
print(max(nums))    # the maximum value within the list
print(min(nums))    # the minimum value within the list
```

A list is an example of a mutable object - an object whose values can be changed.

A tuple is the immutable counterpart of the list. It has similar functionality and uses but the items within it can't be changed. Declaring a tuple is similar to a list but instead of square brackets, you have to use normal parenthesis.

In [ ]:

```
fruits = ("apple", "orange", "tomato", "banana")  # now the tomato is a fruit forever
print(type(fruits))
print(fruits)
```

What will happen if we try to change one of the items?

In [ ]:

```
fruits[2] = "avocado"
```

Exercise

The variable `increments` contains the numbers 0 to 99. Use list slicing to access the numbers from 50 to 75

In [1]:

```
increments = list(range(0, 100))
print(increments[50:76])
```

```
[50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 7
```

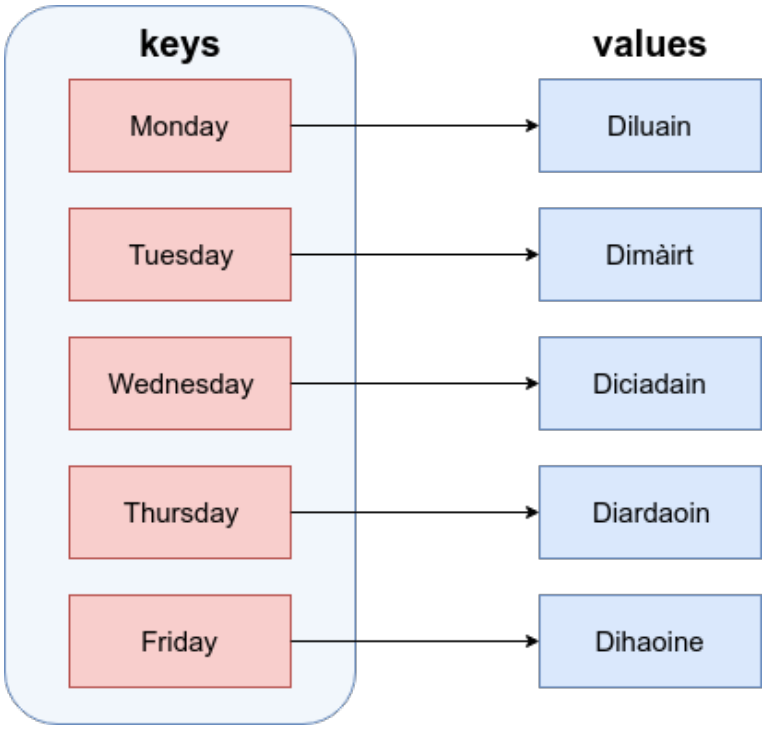
```
2, 73, 74, 75]
```

Dictionaries

As implied by the name, dictionaries are similar to actual dictionaries. They are similar to lists but instead of accessing values via their index, you access them via their key. For example, the Gaelic word *Halò*(Hello in English) and *Hello* will be our key. In that case, we can access the dictionary with:

`dict["Hello"]` **and it will return** `"Halò"`**.**

Here is an image to illustrate the same principle with the days of the week.



In dictionaries, the values are mutable but the keys are immutable. A value can't exist without a key. Definition of a dictionary has the following structure:

```
dict = { key1 : val1,
         key2 : val2,
         key3: val3}
```

Now we can define the dictionary from the image above:

In [ ]:

```
days = {"Monday": "Diluain", "Tuesday": "Dimàirt",
        "Wednesday": "Diciadain", "Thursday": "Diardaoin",
        "Friday": "Dihaoine"}
print(type(days))
print(days)
```

As mentioned previously, to access a value in a dictionary we need to input its key:

In [ ]:

```
days["Friday"]
```

Just like lists, we can modify, add and remove different items in the dictionary.

In [ ]:

```
days.update({"Saturday": "Disathairne"})
print(days)
days.pop("Monday")   # Remove Monday because nobody likes it
print(days)
```

If needed we can extract only the keys or only the values out of the dictionary.

In [ ]:

```
print(days.keys())    # get only the keys of the dictionary
print(days.values()) # get only the values of the dictionary
```

Exercise

The dictionary `months` below maps numbers to the names of months. But it is from an experimental [13-month calendar](#)! Correct the dictionary by

- **printing `months.keys()` and `months.values()` to identify the difference with the normal calendar.**
- **using `months.pop()`, and `months.update()`, to edit the dictionary so it matches up with the normal calendar.**

In [7]:

```
months = {1: "Jan",
          2: "Feb",
          3:"Mar",
          4: "Apr",
          5:"May",
          6:"Jun",
          7:"Sol",
          8:"Jul",
          9:"Aug",
          10:"Sep",
          11:"Oct",
          12:"Nov",
          13:"Dec"}

#Investigate/change the calendar!
print(months.keys())
print(months.values())
months.pop(13)
months.update({7:"Jul",
               8:"Aug",
               9:"Sep",
               10:"Oct",
               11:"Nov",
               12:"Dec"})
# check the updates
print(months.keys())
print(months.values())
print(months)
```

```
dict_keys([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13])
dict_values(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Sol', 'Jul', 'Aug', 'Sep', 'Oct',
'Nov', 'Dec'])
dict_keys([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
dict_values(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov',
'Dec'])
{1: 'Jan', 2: 'Feb', 3: 'Mar', 4: 'Apr', 5: 'May', 6: 'Jun', 7: 'Jul', 8: 'Aug', 9: 'Sep'
, 10: 'Oct', 11: 'Nov', 12: 'Dec'}
```

In [8]:

```
months = {1: "Jan",
          2: "Feb",
          3:"Mar",
          4: "Apr",
          5:"May",
```

```
          6:"Jun",
          7:"Sol",
          8:"Jul",
          9:"Aug",
          10:"Sep",
          11:"Oct",
          12:"Nov",
          13:"Dec"}
# alternatively(more efficient)
print('-'*30)
for i in range(7,13):
    months[i] = months[i+1]
months.pop(13)
# check the updates
print(months.keys())
print(months.values())
print(months)
```

```
------------------------------
dict_keys([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
dict_values(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov',
'Dec'])
{1: 'Jan', 2: 'Feb', 3: 'Mar', 4: 'Apr', 5: 'May', 6: 'Jun', 7: 'Jul', 8: 'Aug', 9: 'Sep'
, 10: 'Oct', 11: 'Nov', 12: 'Dec'}
```

Sets

- **Lists can contain duplicate items, in contrast, a set contains no duplicates.**
- **Sets are mutable and have similar functionality to a list**
- **Can't be indexed or sliced similar to lists**
- **Can be created directly from lists using the `set()` function.**
- **Alternatively, we can also create them with curly brackets `{}`**

In [ ]:

```
x = set([1, 2, 3])  # a set created from a list
print(type(x))

x
```

In [ ]:

```
y = {3, 2, 1}       # a set created directly

x == y              # x and y are the same object
```

Exercise

Use the no-duplicate policy of sets to get all the *distinct* values in the list `x`, without repetition.

In [9]:

```
x = [11, 15, 19, 9, 11, 18, 10, 16, 14, 9, 15, 0, 1, 1, 12, 11, 14, 11, 10, 14]

print(set(x))
```

```
{0, 1, 9, 10, 11, 12, 14, 15, 16, 18, 19}
```

If Else

Sometimes when you try to close a program it asks you `Do you really want to close me?` with possible answers `Yes` and `No`. That is an if-else statement which a structure similar to:

```
if answer:
    close program
```

```
    else:
        continue running program
```

Matter of fact, most of computing relies on this simple concept. To explain it in more details: the statement after `if` must evaluate to a Boolean value (`True` or `False`), if it is `True` then we execute the code between the `if` and `else` statements and skip the code after the `else` statement. If the Boolean expression after `if` evaluates to `False` then we skip the code between `if` and `else` and execute only the code after `else` .

In [ ]:

```
x = True
if x:
    print("Executing if")
else:
    print("Executing else")
print("Prints regardless of the outcome of the if-else block")
```

Try changing the value of x to `False` .

Notice how the final print statement is always executing? That is because it is outside of the if-else block. Python groups code together based on its indentation - the whitespace characters before the code. All lines of code which are next to each other and have the same indentation are part of the same execution block. You can try adding more print statements to the if-else code above and see the results.

Note: Within Jupyter, indentation is 4 whitespaces long. When indenting it is recommended to use the `Tab` key which will simply insert 4 whitespaces.

Another thing you might have noticed in the if-else statement above is the `:` character. In Python, it signals the interpreter that you are starting a code block. After you use it, it will expect a new indented block.

elif

if-else statements can also be extended with `elif` which as implied combines both else + if = elif. This is useful if you want to have multiple conditions for example:

```
    if condition1:
        condition1 was True
    elif condition2:
        condition2 was True and condition1 was False
    else:
        neither condition1 or condition2 were True
```

Let us illustrate this in an example. Fill in the value of `age` in the below code block, and it will use `if` , `else` and `elif` to compute whether someone of this age is:

- **Too young to work full time (under 16)**
- **Of full time working age (16 - 67)**
- **Of state pension age (over 65)**

Since any age must fall into one of these three distinct brackets, `elif` is the perfect way to separate out the different age groups.

In [ ]:

```
age = 15

if age < 16:
    print("Someone of age", age, "is too young to work full-time.")
elif age <= 65:
    print("Someone of age", age, "is of full-time working age.")
else:
    print("Someone of age", age, "is able to draw a state pension.")
```

Exercise

Below, a random integer `n` within the range 0 to 9 is generated.

In [16]:

```
import random
n = random.randint(0, 10)
```

Fill in a value `guess` for the number `n`. Then in the next code block, use `if`, `else` and `elif` to test whether your guess was equal to `n`, or too high, or too low, and print out an appropriate message.

In [17]:

```
guess = 3 # Fill in a guess
```

In [18]:

```
# [ Write code to test if your guess is correct, too high, or too low ]
if guess==n:
    print("Correct!")
elif guess>n:
    print("Too high!")
else:
    print("Too low!")
```

```
Too low!
```

Loops

So far we have seen lists but are they really useful in comparison to normal variables? By themselves no, but if combined with loops like `for` and `while` lists become one of the most valuable concepts in programming!

for

Generally useful whenever you want to iterate over a list (or other data structure) of items and apply the same operation to all items within it. In general, `for` loops look like this and have indentation just like if-else statements:

```
for item in itemList:
    do something to item
```

For example:

In [ ]:

```
fruits = ["apple", "orange", "tomato", "banana"]
for f in fruits:
    print("The fruit", f, "has index", fruits.index(f))
```

is much more elegant than writing

In [ ]:

```
fruits = ["apple", "orange", "tomato", "banana"]
print("The fruit", fruits[0], "has index", fruits.index(fruits[0]))
print("The fruit", fruits[1], "has index", fruits.index(fruits[1]))
print("The fruit", fruits[2], "has index", fruits.index(fruits[2]))
print("The fruit", fruits[3], "has index", fruits.index(fruits[3]))
```

That can be really powerful! Let us try to find the squared value of numbers 0 to 10.

In [ ]:

```
numbers = list(range(10))
for num in numbers:
    squared = num ** 2
    print(num, "squared is", squared)
```

Exercise

Below is a list of names. Use a `for` loop to count the number of times the name Jessica appears in the list. You should get the answer 3.

In [19]:

```
names = ["Jack", "James", "Jessica", "Jacob", "Joshua",
         "Jaxon", "Jack", "Jamie", "Jude", "Jessica",
         "Jackson", "James", "Jack", "Joseph", "Julia",
         "Joshua", "John", "Josh", "Jack", "Jacob",
         "Jake", "Jessica", "James", "Jayden", "Jax"]

#Complete this code
count = 0
for name in names:
    if name=="Jessica":
        count = count + 1 # count += 1
print("The number of times the name Jessica appears in the list is",count)
```

```
The number of times the name Jessica appears in the list is 3
```

while

Another useful loop which is a bit less controllable. It executes over and over until its condition becomes false. For example, we can make a loop that executes 5 times and then stops.

In [ ]:

```
n = 0
while n < 5:
    print("Executing while loop")
    n = n + 1

print("Finished while loop")
```

break

Not a loop but extremely useful within loops! As implied `break` literally breaks the loop and forces the program to go out of it. We can redo our `while` loop example using `break`.

In [ ]:

```
n = 0
while True:  # execute indefinitely
    print("Executing while loop")

    if n == 5:  # stop loop if n is 5
        break

    n = n + 1

print("Finished while loop")
```

Exercise

In the code cell below, type in a value for `n` which is an integer between 0 and 9. Then in the next code block, use a `while` loop and the function `random.randint(0, 10)` to repeatedly generate random numbers between 0 and 9, until you generate `n`. Keep a count of how many tries it takes and print this number at the

between 0 and 9, until you generate `n`. Keep a count of how many tries it takes and print this number at the end.

In [25]:

```
n = 6 #Your value of n
```

In [26]:

```
# [ Write code to keep generating random numbers until you generate n. ]
count = 0
while True:  # execute indefinitely
    guess = random.randint(0,10)
    count = count + 1

    if guess == n:  # stop loop if guess is n
        print("Success!")
        break


print("The number of tries it takes equals {0} and the number is {1}.".format(count,n))
```

```
Success!
The number of tries it takes equals 19 and the number is 6.
```

Functions

Also referred to as methods, functions are effectively small programs that take in arguments(ie. inputs) and return values(outputs). A function we have been using extensively until now is `print`. As obvious it takes various types of arguments and prints them to the console. It and all other functions share the same structure:

```
    def functionName(argument1, argument2, argument3, ... argumentN):
        statments..
        ..
        ..

        return returnValue
```

Functions are an incredibly useful concept since they allow us to package functionality in a convenient and easy to read manner and reproduce the same result without having to write it again and again. A quick example:

In [ ]:

```
def printNum(num):
    print("My favourite number is", num)

printNum(7)
printNum(14)
printNum(2)
```

Rule of thumb - if you are planning on using very similar code more than once, it may be wortwhile writing it as a reusable function.

Can we make a more useful function that returns values(outputs)? We can do that using `return`. When we use `return`, it immedately outputs the value after it and terminates the program. Can we make a program that rounds a number?

In [ ]:

```
x = 3.4
remainder = x % 1
if remainder < 0.5:
    print("Number rounded down")
    x = x - remainder
else:
    print("Number rounded up")
```

```
    x = x + (1 - remainder)

print("Final answer is", x)
```

That works but it will be tedious do have to write it all the time we need to round a number. Can we convert that to a function?

In [ ]:

```
def roundNum(num):
    remainder = num % 1
    if remainder < 0.5:
        return num - remainder
    else:
        return num + (1 - remainder)

# Will it work?
x = roundNum(3.4)
print (x)

y = roundNum(7.7)
print(y)

z = roundNum(9.2)
print(z)
```

That is a very powerful idea!

Note: that this such trivial functionality is already built into Python as the function `round()`.

We can also make a function that returns a tuple of often needed parameters of a list.

In [ ]:

```
def listFunc(my_list):
    maximum = max(my_list)
    minimum = min(my_list)
    first = my_list[0]
    last = my_list[-1]
    return maximum, minimum, first, last
```

In [ ]:

```
l = [24, 12, 68, 40, 120, 96]
params = listFunc(l)
print(params)
print("Max value is", params[0])
print("Min value is", params[1])
print("First value is", params[2])
print("Last value is", params[3])
```

Exercise

Write a function called `mean` **that accepts a list of numbers and returns the mean of the list.**

(Hint: you can use a `for` ***loop to add up the numbers in a list, or use the function*** `sum()`***)***

In [27]:

```
def mean(mylist):
    sum_list = sum(mylist)
    length_list = len(mylist)
    return sum_list/length_list
```

You can check your function using the following code.

In [28]:

```
############### Checking code ######################
# Please don't edit
x = [1, 58, 7, 43, 25, 2, 66, 17]
if mean(x) == 27.375:
    print ("Answer is correct. Good job!")
else:
    print("Wrong answer, please try again.")
```

```
Answer is correct. Good job!
```

Exercises

1. Divisors

Fill in `x` below with an integer. Then create a program that finds a list of all the divisors of `x`.

A divisor is a number that divides evenly into another number. For example, 13 is a divisor of 26 because 26 / 13 has no remainder.)

Hint: You might find `range()` useful here and remember that it can accept a variable as an argument

In [30]:

```
x = 26 # Fill in a number here

# store all possible divisors in a variable below
divisors = []

for each in range(1, x+1):
    print("trying",each)
    if (x%each)==0:
        divisors.append(each)

print(divisors)
```

```
trying 1
trying 2
trying 3
trying 4
trying 5
trying 6
trying 7
trying 8
trying 9
trying 10
trying 11
trying 12
trying 13
trying 14
trying 15
trying 16
trying 17
trying 18
trying 19
trying 20
trying 21
trying 22
trying 23
trying 24
trying 25
trying 26
[1, 2, 13, 26]
```

2. Sort and cube

Given the list `numList`:

1. **Copy it to `newList`.**
2. **Arrange all of the values in `newList` ascending order (Hint: using the built-in function of lists).**
3. **Cube all numbers within the list using a `for` loop. (Think about how you will store the values back to the list). HINT: Make sure you change the value of `newList` whenever you are iterating over it in the `for` loop. Notice how the sorted sequence of numbers in `numList` is very similar to the indexes of a list.**

In [33]:

```
numList = [5, 7, 2, 1, 3, 6, 4]
# Please store the new values in newList below
newList = numList
newList = sorted(newList)
for index, item in enumerate(newList):
    newList[index] = item ** 3
print(newList)
```

```
[1, 8, 27, 64, 125, 216, 343]
```

Run the block below to check your answer.

In [34]:

```
################ Checking code ########################
# Please don't edit
if newList == [1, 8, 27, 64, 125, 216, 343]:
    print ("Answer is correct. Good job!")
else:
    print("Wrong answer, please try again.")
```

```
Answer is correct. Good job!
```

3. sortAndCube()

Now do the same thing as exercise 2 but as a function. Remember that you have to return a value!

Hint: `print()` and `type()` would be helpful in fixing issues with your code.

In [35]:

```
def sortAndCube(numList):
    #Complete this function.
    newList = numList
    newList = sorted(newList)
    for index, item in enumerate(newList):
        newList[index] = item ** 3
    return newList
```

Run the block below to check your answer.

In [36]:

```
################ Checking code ########################
# Please don't edit
x = [5, 7, 2, 1, 3, 6, 4]
if sortAndCube(x) == [1, 8, 27, 64, 125, 216, 343]:
    print(x)
    print ("Answer is correct. Good job!")
else:
    print("Wrong answer, please try again.")
```

```
[5, 7, 2, 1, 3, 6, 4]
Answer is correct. Good job!
```