# DATA SOCIETY®

Introduction to Python - Day 2

*"One should look for what is and not what he thinks should be."*
*-Albert Einstein.*

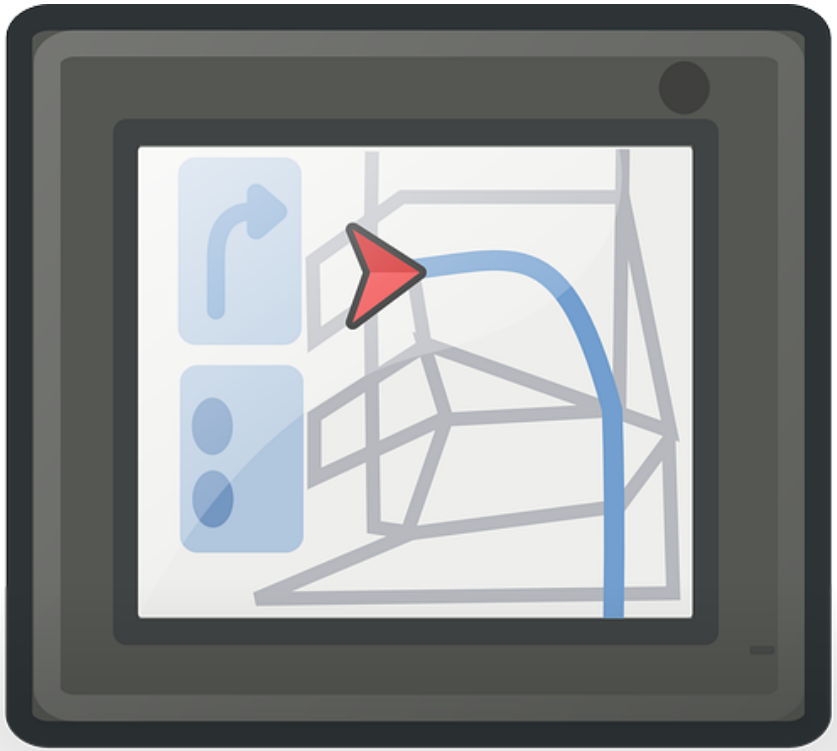
# Last session's material recap

- Did you have any problems working through the materials of the last session?
- Are there any concepts from last session that are still unclear?
- The topics we covered in the last session included:
  - Accessing Jupyter Notebooks
  - Working with variables
  - Using best practices for writing code
  - Working with various data types
  - Creating different data structures

# Module completion checklist

| Objective | Complete |
| --- | --- |
| Discuss control flow structures and practice of writing of modular code | |
| Use conditional statements such as if / else | |
| Use for / while loops and list comprehensions | |
| Build functions | |

# Getting from point A to point B



# Control flow

- When writing a computer program, we are giving a computer a set of directions just like a GPS software is giving us
- We control the program's flow, allowing it to choose which direction to pick based on a set of conditions
- We let the computer go through the program (sometimes repeating actions) until it reaches its end or we interrupt a program


*A set of directions with options for a program to take, which control the program's outcome, is called a* ***control flow***

# Control flow structures

- The directions that allow us to control the flow of a program are called **control flow structures** in programming languages
- Control flow structures are very similar across languages in their concept (but not syntax!)
- **Can you name an example where directions are used to "navigate" through a problem?**

# Basic control flow structures in Python

- We will be discussing the following control structures in this class:
  - Conditionals (`if-else` statements)
  - Loops (`for` and `while` structures)
  - Functions (`def` structure)

# Control flow structure capabilities

- Control flow structures allow us to:
  - Point the program in the right direction *(like conditional statements)*
  - Perform lots of the same tasks over and over again without having to explicitly write out every step *(like loops)*
  - Abstract out some actions that can be re-used later by another part of the program or another program all together *(like functions)*
  - Save us **time and space**
  - Make our programs **clean and readable**
  - Make our code **modular and re-usable**

# Knowledge check 1

# Module completion checklist

| Objective | Complete |
| --- | --- |
| Discuss control flow structures and practice of writing of modular code | ✔ |
| Use conditional statements such as if / else | |
| Use for / while loops and list comprehensions | |
| Build functions | |

# Conditional statements

- *Conditional statements* allow our program to decide whether or not to run certain sections of code, based on some condition
  - If the condition is **true**, the program takes one turn
  - If the condition is **false**, the program takes another turn


- In Python, we use `if` and `else` blocks to build *conditional statements*
- The *condition* itself follows the `if` statement
  - The outcome if the condition is **true** follows on the next line
- The *alternative* route is given by the `else` block
  - The outcome in case the condition is **false** is on the next line

# Conditional statements (cont'd)


```
if 2 + 2 == 4:
    print('Good, we are sane!')
else:
    print('We are living in 1984!')
```

```
Good, we are sane!
```

# Logical operators: recap

- The most common way to create Booleans (i.e. **true** or **false** values) is by using *logical operators*

| Operator | Example |
|---|---|
| Greater than | x > y |
| Less than | x < y |
| Equal to | x == y |
| Not equal to | x != y |
| Greater than or equal to | x >= y |
| Less than or equal to | x <= y |

# Combining Booleans: recap

- We can use `and` and `or` to check for a combination of conditions

```
# Check if 2 conditions are true
# for the expression to return `True`!
x = 8
print(x > 5 and x < 10)
```

```
True
```

- We can ask as many statements as we want with `and`: this tells the program to check all of the conditions, and only return a `True` if **all** of the conditions are met

```
# Every single condition much be true for this expression to return `True`!
print(x > 5 and x > 1 and abs(x) == 7 and x < 10)
```

```
False
```

- Since `abs(x)` is equal to `8` not `7`, the entire expression returns `False`!

# Condition types

- Conditions vary in many ways, but they absolutely need to result in a **true** or **false** output
- Conditional expressions (e.g. `2 + 2 == 4`) can also be assigned to variables
- Long conditional statements are often more readable that way within `if-else` blocks

```
condition = 2 + 2 == 4
print(condition)
```

```
True
```

```
another_condition = "this string" == "that string"
print(another_condition)
```

```
False
```

```
yet_another_condition = 5 + 10 > 10 + 5
print(yet_another_condition)
```

```
False
```

# Condition types (cont'd)

- Conditional statements can be compound (i.e. consist of multiple conditions that result in a single **true** or **false** output)

```
compound_condition1 = (5 + 10 > 10 + 5) and ("this string" == "this string")
print(compound_condition1)
```

```
False
```

```
compound_condition2 = (5 + 10 >= 10 + 5) and ("this string" == "this string")
print(compound_condition2)
```

```
True
```

```
# Here, it's helpful to look at each condition individually.
# If one of them is true, then the whole statement is true.
compound_condition3 = (compound_condition1 and compound_condition2) or (100/2 > 100 % 2)
print(compound_condition3)
```

```
True
```

# Putting it all together

- This is an `if-else` statement with an example of compound condition directly written in it

```
if (5 + 10 > 10 + 5) and ("this string" == "this string"):
    print('Compound condition 1 is true, do something!')
else:
    print('Compound condition 1 is false, do something else!')
```

```
Compound condition 1 is false, do something else!
```

- This is the same `if-else` statement with the compound condition saved to a variable beforehand

```
compound_condition1 = (5 + 10 > 10 + 5) and ("this string" == "this string")

if compound_condition1:
    print('Compound condition 1 is true, do something!')
else:
    print('Compound condition 1 is false, do something else!')
```

```
Compound condition 1 is false, do something else!
```

- Both ways are correct

# Special cases of conditional blocks

- Conditional blocks of code don't have to be in the `if-else` form
- Sometimes you want to take action if the condition is **true**, but don't need to do anything if it is **false**
- In that case, just a single `if` block will do the trick!

```
if compound_condition2:
    print("Ok, I guess I have to do something after all!")
```

```
Ok, I guess I have to do something after all!
```

```
if yet_another_condition:
    print("This means the `compound_condition3` is true, otherwise you will get nothing!")
```

- In case of the first example, the action is triggered, because our `compound_condition2` is **true**, but in the second one, we get nothing (i.e. the `print` statement is not triggered), because `yet_another_condition` is **false** and we didn't provide our program with an alternative action!

# Special cases of conditional blocks (cont'd)

- If we want to check multiple conditions, we can use `elif`
- As soon as one condition is met, the cascading down the `elif` tree will stop
- Suppose you want to purchase a car - you might make different decisions based on its price

```
price = 37000

if price > 40000:
    print("That's too expensive!")
elif price > 34000:
    print('A little pricey but maybe worth it...')
elif price > 26000:
    print("This seems like a fair price for the quality")
elif price > 22000:
    print("What a good deal! I'll get it")
else:
    print("Hmmm this is pretty cheap, maybe there's a problem with it.")
```

```
A little pricey but maybe worth it...
```

# Special cases of conditional blocks (cont'd)

- Sometimes we have a complex set of decisions to make based on the outcome of the other set of decisions
- In that case, **nested** conditionals are the way to go!

```
if condition:
    if another_condition:
        print("You got to a nested statement!")
    else:
        print("You still got to the nested statement!")
else:
    print("No luck printing a nested statement!")
```

```
You still got to the nested statement!
```

# Actions within conditional blocks

- Well, we went over the types of conditional blocks as well as conditions themselves
- What about the actions within those blocks?
- **What can we actually put within `if-else` blocks?**

# Actions within conditional blocks

- Anything!

```
price = 37000
account_balance = 45000

if price > 38000:
    action = "Leave the dealership immediately, this is a rip off!"
    account_balance = account_balance - price
elif price > 22000 and price <= 38000 :
    action = "Take the car and go celebrate, you can afford it!"
    account_balance = account_balance - price
else:
    action = "Leave the dealership immediately, this is a scam!"
    account_balance = account_balance - price

print(action)
```

```
Take the car and go celebrate, you can afford it!
```

```
print("Current account balance:", account_balance)
```

```
Current account balance: 8000
```

# Conditionals and actions based on them

- Any action can "live" within a conditional block
  - We can perform mathematical operations
  - Assign variables new values
  - Build other conditional blocks within them (a.k.a. *nested conditional blocks*)
  - Add *loops*
  - Call *functions* or even other *programs*

# Knowledge check 2

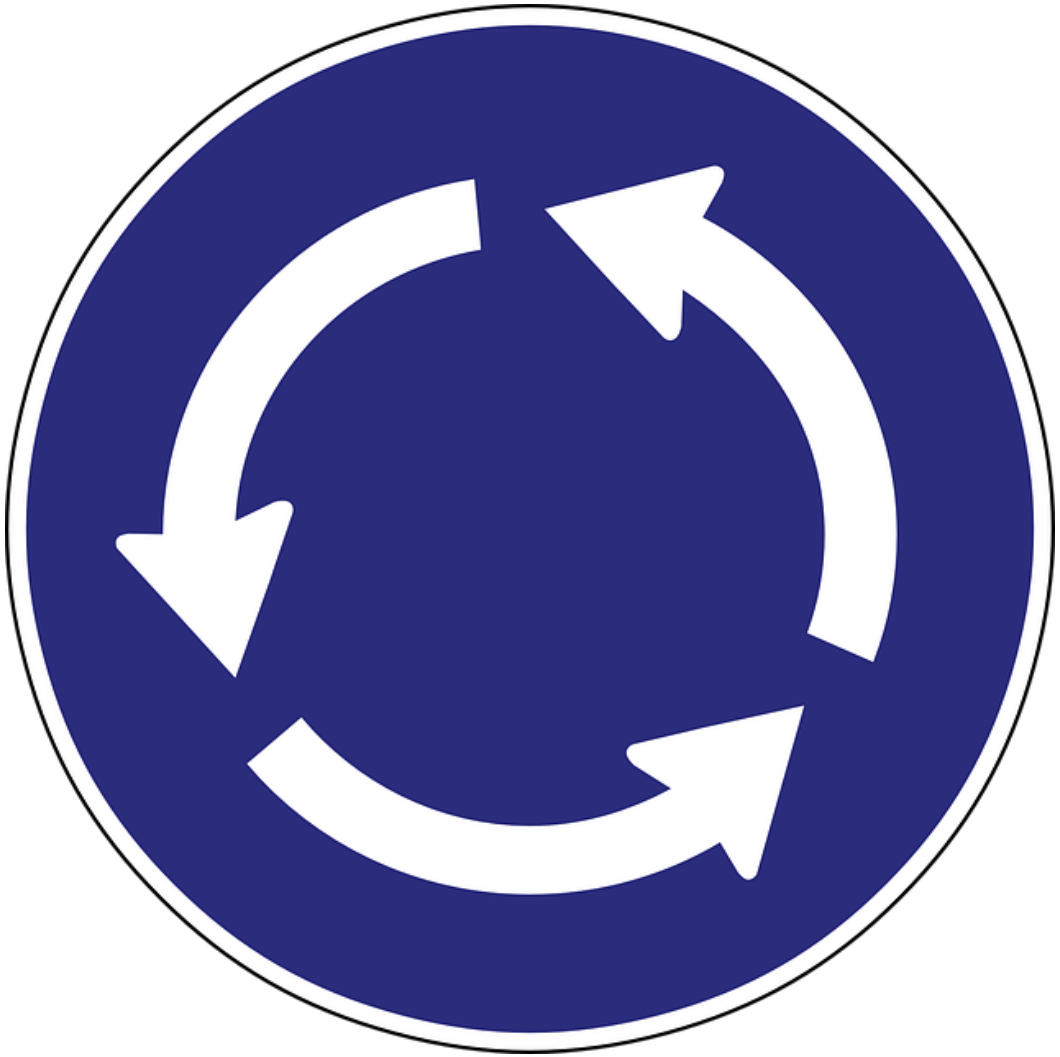
# Exercise 1

# Module completion checklist

| Objective | Complete |
| --- | --- |
| Discuss control flow structures and practice of writing of modular code | ✔ |
| Use conditional statements such as if / else | ✔ |
| Use for / while loops and list comprehensions | |
| Build functions | |

# Loops

- *Loops* allow our program to perform tasks over and over again given either:
  - A **counter** based on the number of actions we need to perform, or
  - A **condition**, based on which the loop will keep going until it no longer holds true
- In Python, we use `for` and `while` blocks to build *loops*
  - The loops using a **counter** are defined with `for` loops
  - The loops using a **condition** are defined with `while` loops



# Setting up a calendar invite for your party

- Imagine you have to organize a party
- You set up a calendar invite
- You need to send out invitations to each of your contacts
- For every contact in your contact list, you will need to perform the following actions:
  - copy the email address of the person
  - add it to a calendar invite
- **What kind of loop do you think you need for this?**

# For loops: when to use them?

**When should you use for loops?**

- If you have a **finite** set of items you need to go through
- If your items are organized in a *list*, *array*, *dictionary*, *sequence*, or any other **collection of elements**

- In our case of party planning, we need to add an email address of a person from a **list** of contacts to the calendar invite we have set up
  - A *list* is a **finite** set of elements
  - It is easy to *count its elements from start to end*, so our loop has a definitive *start and end*

# For loops in Python

To define a `for` loop in Python, we need 2 main components:

1. An **object** (e.g. a `list`) through which we would like to *iterate*
2. A **counter** variable that will increase based on the progress of the loop or an **element** variable that takes on the value of next element in the collection of elements through which we iterate

In the code to the right:

1. The **object** through which we *iterate* is a `contact_list`
2. The **element** of that object is a `name` variable that takes on a value from a `contact_list` at *each iteration* of the loop

```
contact_list = ['Christian Bale', 'Bradley
Cooper', 'Willem Dafoe', 'Rami Malek',
'Viggo Mortensen', 'Yalitza Aparicio',
'Glenn Close', 'Olivia Colman', 'Lady
Gaga', 'Melissa McCarthy']
```

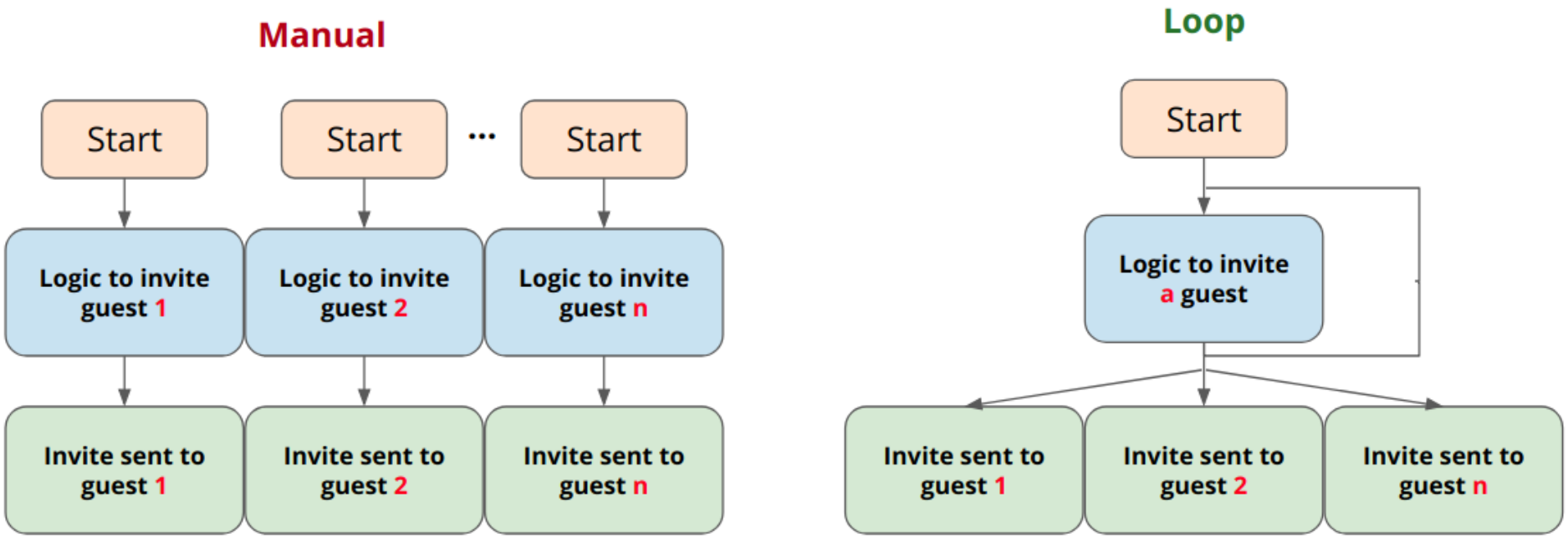
```
for name in contact_list:
    print('Invite ' + name + '!')
```

```
Invite Christian Bale!
Invite Bradley Cooper!
Invite Willem Dafoe!
Invite Rami Malek!
Invite Viggo Mortensen!
Invite Yalitza Aparicio!
Invite Glenn Close!
Invite Olivia Colman!
Invite Lady Gaga!
Invite Melissa McCarthy!
```

- Where do you think the party is going to happen? ☻

# Yes, you got it!

# For loop logic summary



# Making a sequence of numbers in Python

- When working with loops, you will be looking at lots of sequential elements and numbers
- To quickly create a sequence of numbers in Python, we will use `range()` function
- When given a single number, `x`, this function will generate consecutive numbers from `0` to `x` not including `x`
- For example, to generate a sequence from `0` to `9` inclusive you need to do the following
  - `range(10)`

```
sequence = range(10)
print(sequence)
```

```
range(0, 10)
```

```
for number in sequence:
    print(number)
```

```
0
1
2
3
4
5
6
7
8
9
```

# Making a sequence of numbers in Python (cont'd)

- To generate a sequence of numbers that starts with another number other than `0`, add a number as the first argument in `range()` method
  - `range(start, end + 1)`
- To generate a sequence of numbers from `1` to `10` inclusive
  - `range(1, 11)`

```
sequence = range(1, 11)
print(sequence)
```

```
range(1, 11)
```

```
for number in sequence:
    print(number)
```

```
1
2
3
4
5
6
7
8
9
10
```

# Making a sequence of numbers in Python (cont'd)

- To generate a sequence of numbers with a step size different than `1`, you can add the third argument to `range()` function
  - `range(start, end + 1, step_size)`
- To generate a sequence of even numbers from `2` to `20` inclusive
  - `range(2, 21, 2)`

```
sequence = range(2, 21, 2)
print(sequence)
```

```
range(2, 21, 2)
```

```
for number in sequence:
    print(number)
```

```
2
4
6
8
10
12
14
16
18
20
```

# For loops in Python (cont'd)

- An alternative and a more general way to perform the same task of looping through a contact list is by using a **counter**
- In this case, we will use a counter as an **index** to access the elements of the `contact_list`

```
print(len(contact_list))
```

```
10
```

- To go through the entire list, we need to start at index `0` and end with index `9`, since there are a total `10` elements in the `contact_list`

```
# Save length of the list as number of contacts
# for convenience.
num_contacts = len(contact_list)
```

```
# Go through indices in a range between 0 and 9.
for i in range(num_contacts):
    # Invite a person in the list at index i.
    print('Invite ' + contact_list[i] + '!')
```

```
Invite Christian Bale!
Invite Bradley Cooper!
Invite Willem Dafoe!
Invite Rami Malek!
Invite Viggo Mortensen!
Invite Yalitza Aparicio!
Invite Glenn Close!
Invite Olivia Colman!
Invite Lady Gaga!
Invite Melissa McCarthy!
```

# Using a for loop to build an object

- What if we need to build an object with squares of numbers?
- We don't know what those elements should be, but we know 2 things
  - The **base for our squares**: numbers from 5 through 15
  - The **formula**: $x^2$
- We can create an empty list of `squares`, and a **range of numbers** from `5` through `15`
- **Apply the formula** to each number `x`, getting an `x_squared` value
- Append the `x_squared` value to the list of `squares`

```
squares = []

for x in range(5, 16):
    x_squared = x**2
    squares.append(x_squared)
    print("Square of", x, "is", x_squared)
```

```
Square of 5 is 25
Square of 6 is 36
Square of 7 is 49
Square of 8 is 64
Square of 9 is 81
Square of 10 is 100
Square of 11 is 121
Square of 12 is 144
Square of 13 is 169
Square of 14 is 196
Square of 15 is 225
```

```
print(squares)
```

```
[25, 36, 49, 64, 81, 100, 121, 144, 169, 196,
225]
```

# List comprehension

- Since `for` loops are tightly connected to *finite collections of elements* like **lists**, there exists a shorthand for creating lists using `for` loops called **list comprehension**
- It was made to simplify the code and make it more concise and clean and it is specific to Python!
- It is more computationally efficient than the example on the previous slide

```
squares = [x**2 for x in range(5, 16)]
print(squares)
```

```
[25, 36, 49, 64, 81, 100, 121, 144, 169, 196,
225]
```

# Looping over a dictionary

- When you iterate through dictionaries, keep in mind that a dictionary is a list of *tuples*
- Take this dictionary of fare prices for different means of transportation, for instance

```
prices = {'bus': 1.75, 'metro': 3.50, 'uber': 8.75, 'lyft': 7.50}
```

- We cannot use a simple index to access an element of the `prices` like this: `prices[i]`
- We will get a `KeyError` because we must access dictionary elements using a `key`
- So we use the `items()` method, which extracts the `keys` and `values` from each tuple

```
for key, value in prices.items():
    print('The price for', key, 'is', value)
```

```
The price for bus is 1.75
The price for metro is 3.5
The price for uber is 8.75
The price for lyft is 7.5
```

# While loops

- Sometimes you know you have to perform a repeating action, but you are not sure exactly how many times
- This is where `while` loops enter the scene
- They are typically used when we iterate through an object based on some **condition**, instead of a concrete set of elements
- When condition is no longer holds, the program will **exit** the loop

# While loops in Python

- While loops are defined in Python using the `while` block
- To build a loop, you just need to have your **condition** defined at top of your loop structure
- Consider our party invitation loop - we can write it as a `while` loop with minor modifications

```
# Set the index to starting point.
i = 0
```

```
# While  the index is less than the
# total number of contacts.
while i < num_contacts:

    print('Invite ' + contact_list[i] + '!')

    # Increase your index to advance
    # to the next name on list.
    i = i + 1
```

```
Invite Christian Bale!
Invite Bradley Cooper!
Invite Willem Dafoe!
Invite Rami Malek!
Invite Viggo Mortensen!
Invite Yalitza Aparicio!
Invite Glenn Close!
Invite Olivia Colman!
Invite Lady Gaga!
Invite Melissa McCarthy!
```

# Will these loops work?

- Consider the following code

```
k = 0

while k < 10:
    print(k)
```

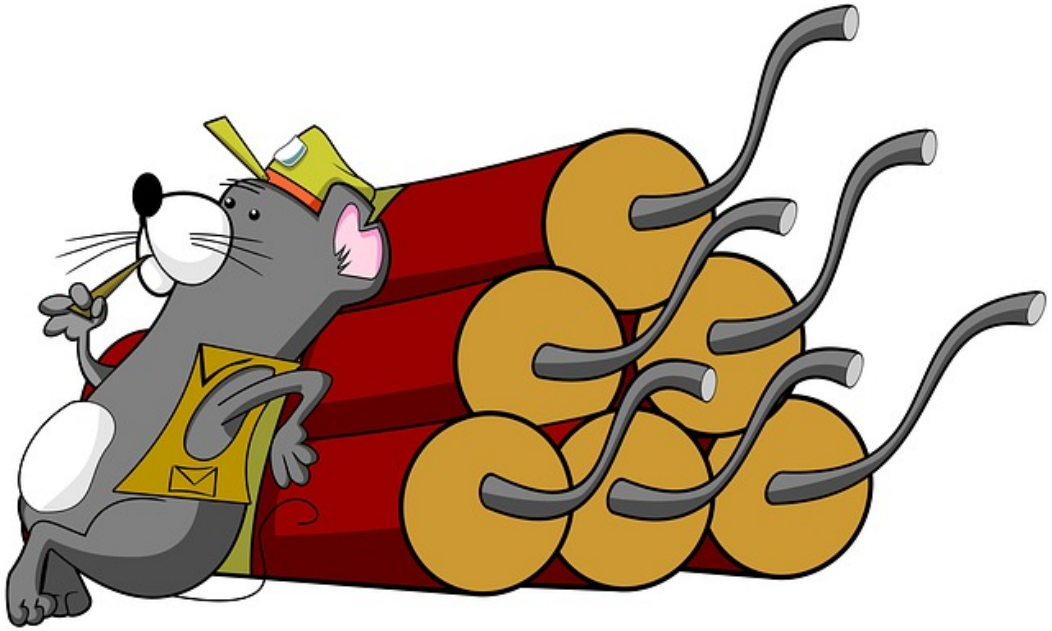
- Consider another example

```
while i <= num_contacts:

    print('Invite ' + contact_list[i] + '!')
    i = i + 1
```

- What do you think will happen if you try to run these chunks of code?
- Why do you think these loops are not going to work?
- What do you need to do to fix them?

# While loops: danger zone!



- The biggest problem with `while` loops is a **poorly defined stopping condition**
- It can happen in case when
  - The condition itself is faulty
  - The counter (like `k` in previous slide) is not being increased
- This can lead to
  - *Infinite loops* like in the previous slide
  - Letting the loop go *out of bounds* of a `list` or `array`, which will produce an error
- If you can **substitute** a `while` loop with a `for` loop, do it… it may save you a lot of trouble!

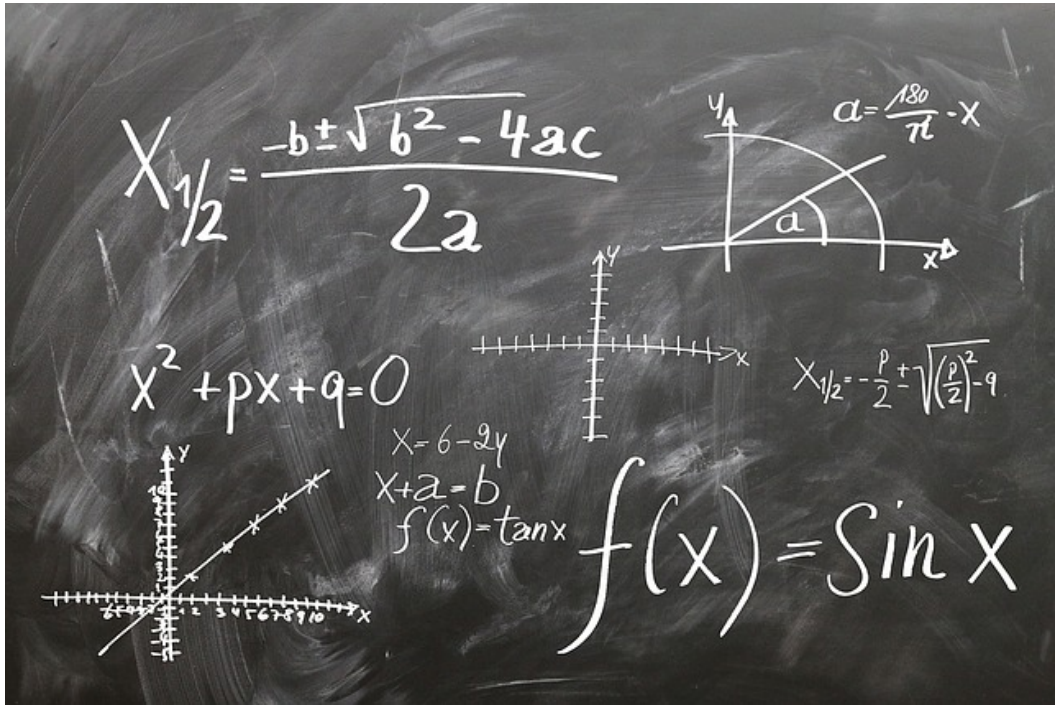
# Knowledge check 3

# Exercise 2

# Module completion checklist

| Objective | Complete |
|---|---|
| Discuss control flow structures and practice of writing of modular code | ✔ |
| Use conditional statements such as if / else | ✔ |
| Use for / while loops and list comprehensions | ✔ |
| Build functions | |

# Functions are fun!




- When you hear a word **function**, you might imagine a blackboard full of math symbols
- **Why do you need to deal with them here?**

# How functions apply to programming

- In programming, functions serve a very similar purpose as in mathematics
- They help us *abstract from operations on actual numbers and let us define those operations through a set of variables*
- The **logic of operations stays stationary**, but the **value of variables changes**
- This makes it **easy to re-use such blocks** and only pass new values to those variables

# Functions in Python

- Functions can
  - Have `0` or more arguments
  - Perform `1` or more actions
  - Return some value(s) or not return anything
- The simplest possible function will have no arguments and will not return any value

- We need to **define** a function, for which we use a `def` construct that includes:
  - a `name` for function
  - an `action` or `actions` to perform when the function is triggered

```
# Define a function that prints the value of `Pi`.
def PrintPi():     #<- function name
    print(3.14)    #<- action to perform
```

- To "trigger" the function, we can **call** it by its name

```
PrintPi()
```

```
3.14
```

# Functions in Python with 1 or more arguments

- Most of the time, you will create or use a function that has at least `1` **argument**
- When we need to define a function with argument(s), we use a `def` construct that includes:
  - a `name` for function
  - at least `1` **argument** in parentheses
  - an `action` or `actions` to perform when the function is triggered

```
# Define a function that prints the value of `Pi`
# and takes a number of decimal points to which we want to round the number.
def PrintPi(num_decimals):                  #<- function name + argument(s)
    pi = 3.14159265359                      #<- action to perform
    rounded_pi = round(pi, num_decimals)    #<- action to perform
    print(rounded_pi)                       #<- action to perform
```

```
# Print value of pi rounded to 4 decimal points.
PrintPi(4)
```

```
3.1416
```

# Functions in Python that return a value

- Functions can also **return value(s)** in addition to some `actions` they perform
- When we need to define a function with or without argument(s) that **return value(s)**, we use a `def` construct that includes
  - A `name` for function
  - At least `0` or more `arguments` in parentheses
  - An `action` or `actions` to perform when the function is triggered
  - A `return` **value** or **values**

```
# Define a function that prints the value of `Pi`,
# takes a number of decimal points to which we want to round the number,
# and returns the value back to us.
def GetPi(num_decimals):                    #<- function name + argument(s)
    pi = 3.14159265359                      #<- action to perform
    rounded_pi = round(pi, num_decimals)    #<- action to perform
    return rounded_pi                       #<- value to return
```

# Functions in Python that return a value (cont'd)

- Let's print the value that `GetPi()` returns

```
# Return a value of pi rounded to 4 decimal points.
print(GetPi(4))
```

```
3.1416
```

- A function that returns a value **can also be assigned to a variable**
- Now the value returned and assigned to a variable can be re-used throughout the code!

```
# Return a value of pi rounded to 4 decimal points.
# Assign it to a variable.
pi_4 = GetPi(4)
print(pi_4)
```

```
3.1416
```

# Functions in Python: general structure

- Here is a general outline of a function that takes arguments and returns something in Python
- The arguments, actions, and returned values are all up to you to define based on what you are trying to achieve!

```
def FunctionName(argument1, argument2, ...):
    action1
    action2
    ...
    return something
```

# Functions in Python: MakeFullName

- Let's define a function that concatenates the *first name* and *last name* into a **full name**

```
# Define a function that concatenates
# first and last names.
def MakeFullName(first_name, last_name):
    full_name = first_name + ' ' + last_name
    return full_name
```

- Here are the components of the function definition broken down:
- `def` defines a function
- `MakeFullName` is the name of our function
- The two arguments it takes are specified in the parentheses: `first_name` and `last_name`
- The `return` statement controls what gets returned when someone uses the function (i.e. `full_name`)

# Functions in Python: calling a function

- After we have defined a function, we need to test it
- Running a function in programming is usually known as *calling* a function
- Let's call `MakeFullName()` - to do that, we need to call the function by its name and pass any necessary values we want for the arguments of the function

- Let's print out the function results

```
# Call the function.
print(MakeFullName("Harry", "Potter"))
```

```
Harry Potter
```

- If the function returns a value (like a string with full name in this case), we can assign the output of that function to a variable

```
# Call the function and save
# output to variable.
output_name = MakeFullName("Harry", "Potter")
print(output_name)
```

```
Harry Potter
```

# Functions in Python: EvaluateCarPrice

- Let's define another function that evaluates the price of a car and gives us the action to do based on that
- Here are the components of the function definition broken down:
  - `EvaluateCarPrice` is the name of our function
  - The argument it takes is specified in the parentheses: `price`
  - The `return` statement controls what gets returned when someone uses the function (i.e. `action`)

```
def EvaluateCarPrice(price):
    if price > 38000:
        action = "Leave the dealership immediately, this is a rip off!"
    elif price > 22000 and price <= 38000 :
        action = "Take the car and go celebrate, you can afford it!"
    else:
        action = "Leave the dealership immediately, this is a scam!"
    return action
```

# Functions in Python: calling a function

```
# Let's set the price of a car to $45,000.
price = 45000
action1 = EvaluateCarPrice(price)
print(action1)
```

```
Leave the dealership immediately, this is a rip off!
```

```
# Let's set the price of a car to $32,000.
price = 32000
action2 = EvaluateCarPrice(price)
print(action2)
```

```
Take the car and go celebrate, you can afford it!
```

```
# Let's set the price of a car to $5,000.
price = 5000
action3 = EvaluateCarPrice(price)
print(action3)
```

```
Leave the dealership immediately, this is a scam!
```

# Functions that return two or more values

- Let's define another function that evaluates the sales data (i.e. a dictionary with `month: amount` key-value pairs)
  - `MostProfitableMonth` is the name of our function
  - The argument it takes is a dictionary `sales_data`
  - It **returns two values**: `biggest_month` and `biggest_amt` for the month with the most profit

```
def MostProfitableMonth(sales_data):

    biggest_amt = 0
    biggest_month = None

    for month, amt in sales_data.items():
        if amt >= biggest_amt:
            biggest_amt = amt
            biggest_month = month

    return (biggest_month, biggest_amt)
```

# Functions that return two or more values (cont'd)

```
# Let's define a dictionary with sales data.
year_sales = {'January': 1045, 'February': 1008, 'March': 1025,
                    'April': 1080, 'May': 1100, 'June': 1050, 'July': 1050,
                    'August': 950, 'September': 1010, 'October': 1500,
                    'November': 1450, 'December': 1380}
```

- To save the output of a function that returns 2 or more values, assign the function call output to the same number of variables as there are outputs in the function in the same order

```
# Assign output of function to variables in correct order.
best_sales_month, best_sales_amt = MostProfitableMonth(year_sales)
print("Best month:", best_sales_month)
```

```
Best month: October
```

```
print("Best amount:", best_sales_amt)
```

```
Best amount: 1500
```

# Functions that have default arguments

- Let's modify another `MostProfitableMonth` and add another argument
  - This argument will have a **default** value
  - We are not required to provide the other argument, if it has a default value
- Let's set the second argument to be `verbose`, which in programming usually signals to print some message along with returned value
- Let's set the **default value** to `True`

```
def MostProfitableMonth(sales_data, verbose = True):

    biggest_amt = 0
    biggest_month = None

    for month, amt in sales_data.items():
        if amt >= biggest_amt:
            biggest_amt = amt
            biggest_month = month

    if verbose:
        print('The best sales month was', biggest_month, 'with amount sold equal to', biggest_amt)

    return (biggest_month, biggest_amt)
```

# Functions that have default arguments (cont'd)

- Let's call the function `MostProfitableMonth()` and provide only the first (required) argument

```
# Assign output of function to variables in correct order.
best_sales_month, best_sales_amt = MostProfitableMonth(year_sales)
```

```
The best sales month was October with amount sold equal to 1500
```

- Let's call the function `MostProfitableMonth()` and set the `verbose` argument to `False`

```
# Assign output of function to variables in correct order.
best_sales_month, best_sales_amt = MostProfitableMonth(year_sales, False)
```

- The output is still the same, but the "verbose" message is no longer printed, so if we need to see it, we would have to do it manually

```
print("Best month:", best_sales_month, "Best amount:", best_sales_amt)
```

```
Best month: October Best amount: 1500
```

# Anonymous functions: lambda

- Anonymous functions are **small nameless functions**
- They work the same way as conventional functions, but are **used as shorthand within other pieces of code**
- They are often used as a part or as an argument in other functions

# Anonymous functions: lambda (cont'd)

- To write an anonymous function in Python, we use the `lambda` command followed by an **argument** and a `:`
- The expression that follows represents an **action** we would like to perform using those arguments

```
remember = lambda v: v + " the 5th of November!"
print(remember("Remember"))
```

```
Remember the 5th of November!
```

- Anonymous functions can take several arguments separated by a comma

```
y = lambda a, b: a + b
print(y(765, -987))
```

```
-222
```

# Knowledge check 4

# Exercise 3

# Module completion checklist

| Objective | Complete |
| --- | --- |
| Discuss control flow structures and practice of writing of modular code | ✔ |
| Use conditional statements such as if / else | ✔ |
| Use for / while loops and list comprehensions | ✔ |
| Build functions | ✔ |

# Workshop!

- Workshops are to be completed in the afternoon either with a dataset for a capstone project or with another dataset of your choosing
- Make sure to **annotate and comment your code** so that it is easy for others to understand what you are doing
- This is an exploratory exercise to **get you comfortable with the content** we discussed today
- **Today, you will:**
  - Use if/else statements
  - Run for loops and while loops
  - Run loops through list comprehension
  - Define functions and use lambda functions

# This completes our module

**Congratulations!**