

PROGRAMMING LANGUAGES

```
each: function(e, t, n) {
  var r, i = 0,
      o = e.length,
      a = M(e);
  if (n) {
    if (a) {
      for (; o > i; i++)
        if (r = t.apply(e[i], n), r === !1) break;
    } else
      for (i in e)
        if (r = t.apply(e[i], n), r === !1) break;
  } else if (a) {
    for (; o > i; i++)
      if (r = t.call(e[i], i, e[i]), r === !1) break;
  } else
    for (i in e)
      if (r = t.call(e[i], i, e[i]), r === !1) break;
  return e
},
trim: b && !b.call("\uffeff\u00a0") ? function(e) {
  return null == e ? "" : b.call(e)
} : function(e) {
  return null == e ? "" : (e + "").replace(C, "")
},
makeArray: function(e, t) {
  var n = t || [];
  return null != e && (M(Object(e)) ? x.merge(n, "string" == typeof e ? [e] : e) : h.call(n, e)), n
},
isArray: function(e, t, n) {
  var r;
  if (t) {
    if (n) return m.call(t, e, n);
    for (r = t.length, n = n ? 0 > n ? Math.max(0, r + n) : n : 0; r > n; n++)
      if (n in t && t[n] === e) return n;
  }
}
```

WHAT WE WILL LEARN

INTRODUCTION TO PROGRAMMING

ALGORITHMS

COMMON BUILDING BLOCKS OF LANGUAGES

HANDLING ERRORS

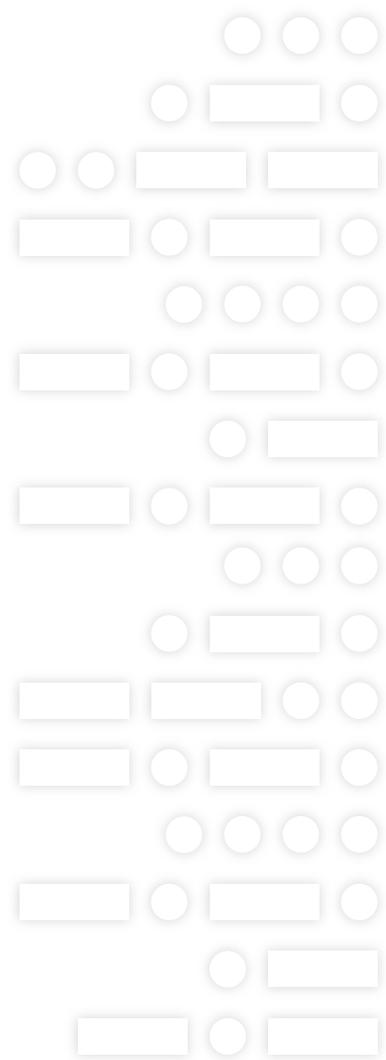
MEMORY MANAGEMENT

HOW PROGRAMS ARE EXECUTED (RUN)



PROGRAMMING: ?

Programming is the art of making hardware do things, by writing instructions for a device to act upon. While programming we often express these instructions in the form of **algorithms**. But it goes beyond algorithms, including control, input, output, data collection and distribution. In today's world Every digital device on the planet has some form of software embedded in it, i.e. someone wrote a program to enable that device to function.



PROGRAMMING: LANGUAGES

In the world of computing and specifically software development there are a multitude of different programming languages.

Whilst programming languages often share similar attributes in terms of how you write code many of them are designed for specific purposes.

To better understand the wide range of languages available visit:

<https://codedocs.org/what-is/list-of-programming-languages->

In this course we will concentrate on the fundamental structure of the kind of programming languages that we use for modern development purposes. All of these languages share common features.

Before we get into the languages themselves we should look at what constitutes writing a program.



PROGRAMMING: LANGUAGE PARADIGMS

There are numerous programming languages - most of which fit into one of or more of the following design paradigms.

- Imperative / procedural
 - Algorithms are as a hierarchy of tasks (procedures) that operate on data
 - Examples: Javascript, Fortran, Cobol, Basic, C, Pascal, Ada
- Functional
 - Computation is expressed in terms of the evaluation of functions
 - Lisp, Scheme, ML, CaML, Javascript, Scala
- Logic / declarative
 - A program consists of a set of facts and rules about objects, and a way to ask questions about objects and their relationships
 - Prolog
- Object-oriented
 - Computation is performed by a set of interacting objects
 - C++, Java, Python, Smalltalk, Simula, Ada, Javascript

These are general views and whilst languages like Prolog just doesn't fit into the object oriented category, Python is object-oriented, procedural and functional, i.e. Multi-paradigm. And Javascript is also Multi-paradigm: event-driven, functional, imperative, object-oriented. Above are just a few of the languages that fit into these paradigms.

ALGORITHMS

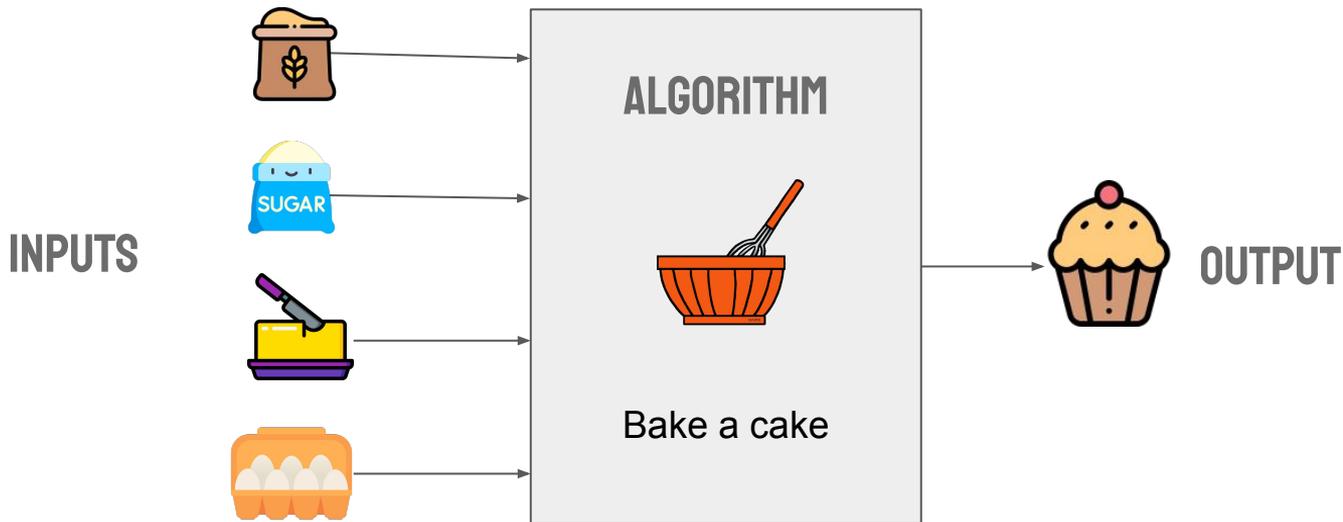
THE ART OF MAKING CODE MEANINGFUL



ALGORITHMS

The term algorithm comes from the name of Iranian mathematician Muḥammad ibn Mūsā al-Khwārizmī who is described as the father of algebra.

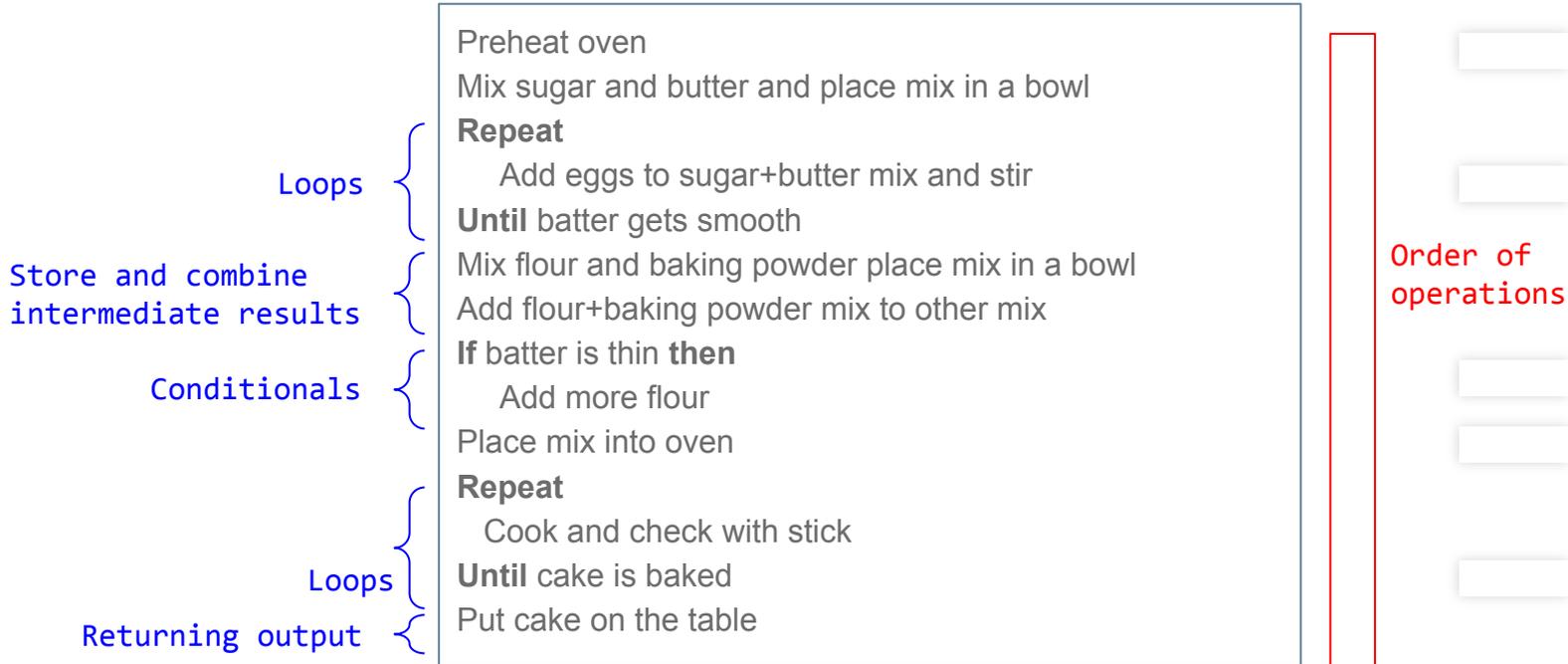
Algorithms are at the heart of software development and programming languages. An Algorithm is a detailed sequence of actions to accomplish a specific task. In short an algorithm is a way of producing some output from some input. The source of that input can be anything from a human, a data pipeline, even a sensor..



ALGORITHMS: BAKE A CAKE

An algorithm consists of a sequence of actions.

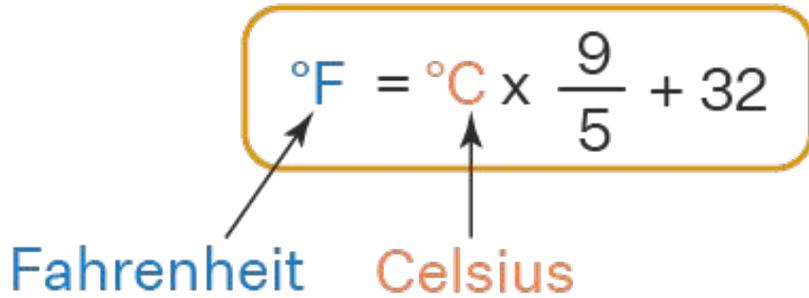
- Some actions are conditional: only happen if something happens.
- Some actions are performed multiple times in a loop, until something happens.
- Actions may produce and consume intermediate results of other actions.



ALGORITHMS: CONVERT TEMPERATURE

$$^{\circ}\text{F} = ^{\circ}\text{C} \times \frac{9}{5} + 32$$

Fahrenheit Celsius



1. Take Celcius degree as **Input**
2. Multiply input by 9
3. Divide the result by 5
4. Increase the result by 32
5. **Output** the result as Fahrenheit



ALGORITHM: PSEUDO CODE

Pseudo-code is a generic notation to express computations. It is not a real programming language, but helps with expressing the algorithm without following a specific syntax. Therefore, we can express the algorithm without code which provides a process of evaluation and iteration on the algorithmic concept. Once happy with the concept we can commit it to code.

```
ConvertToFahrenheit (inputValue) { # 1. Take Celsius degree as Input
    Var result
    result = inputValue * 9         # 2. Multiply input by 9
    result = result / 5            # 3. Divide the result by 5
    result = result + 32          # 4. Increase the result by 32
    return result                  # 5. Output the result as Fahrenheit
}

Var f = ConvertToFahrenheit(30)

Print(f) # Prints 86
```

PROGRAMMING LANGUAGES: SHARE SIMILAR CONSTRUCTS

In this class, we will first explain programming constructs in pseudo-code. Later, you will discover how to write these constructs in Javascript and Python.

```
ConvertToFahrenheit (celcius) {  
  Var result = celsius * 9  
  result = result / 5  
  result = result + 32  
  return result  
}
```

```
Var f = ConvertToFahrenheit(30)  
Print(f) # Prints 86
```

Pseudo-code

```
function convertToFahrenheit(celcius) {  
  var result = celcius * 9;  
  result = result / 5;  
  result = result + 32;  
  return result;  
}
```

```
var f = convertToFahrenheit(30)  
console.log(f) // Prints 86
```

JS

```
def convertToFahrenheit(celcius):  
  result = celcius * 9  
  result = result / 5  
  result = result + 32  
  return result
```

```
f = convertToFahrenheit(30)  
print(f) # Prints 86
```



PROGRAMMING LANGUAGES: SYNTAX AND SEMANTICS

Before we start we need to get some basic concepts in our heads.

Programming languages are defined by their syntax and semantics.

- **Syntax** is Grammar
 - Rules for writing grammatically correct programs.
 - For example: Variable and function definitions, assignment, operations, comments.
 - If the syntax is incorrect, your program will not run.
- **Semantics** is Meaning
 - Describes the behavior of syntactically correct programs.
 - For example: How conditionals and loops work, how functions are called, calculations are made.
 - If the semantics are incorrect, your program will run, BUT will provide incorrect result and/or errors

Example in English:

“I is reading a newspaper.” → Syntax is incorrect.

“I am reading a mouse.” → Syntax is correct but something is wrong with the semantics .

COMMON BUILDING BLOCKS



LITERALS AND VARIABLES: REPRESENTING VALUES

We write code to process data. Data appears in our code in various forms. We refer to data by literals and variables.

A literal points to a fixed data value, such as the number 1024. Whenever we put 1024 in code, it always points to the same number.

On the other hand, a variable (as it says “vary-able”) points to different values as the code is running. When we mention a variable in some expression at a point in code, we may be referring to a different value depending on what was assigned to that variable.

Let’s go over what literals and variables are.



LITERALS: FIXED VALUES

Literals are fixed values of varying types. A literal value does not change. In programming we often refer to these as constants. The value is constant it literally never changes.

Integer: 1, 2, 256, -100

Hexadecimal: 0x12F

Float: 4.32, 5.4, 4.0

String: "I love Fethiye"

Boolean: true, false

Object: {color: 'red'}



VARIABLES: MEMORY SLOTS TO CONTAIN VALUES

Variables are containers that store literal values. In some languages, a variable has a fixed type, i.e. a variable when assigned as an integer or a string can only contain a literal of the same type it is declared with.

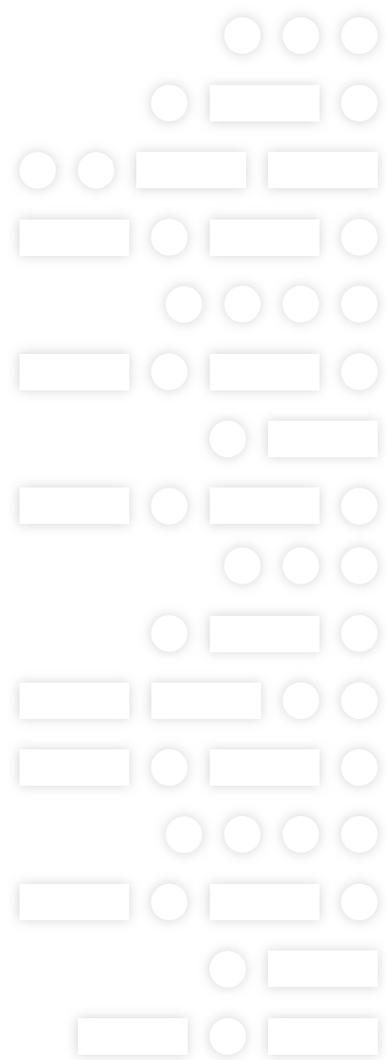
In other languages, variable types are dynamic, i.e. you can assign different types such as integers and strings to the same variable but not at the same time.

age
40

height
175.6

name
"Tayfun"

is_weather_cold
true



VARIABLES: STATIC AND DYNAMIC

The type of a variable is the type of the value stored in the variable. As previously mentioned, the type may or may not change depending on the language.

Static

```
int age; // age is of type int/integer
```

```
age = 30; // OK
```

```
age = "Tayfun"; // Error!
```



Dynamic

```
var age; // age is any type
```

```
age = 30; // OK
```

```
age = "Tayfun"; // OK
```



VARIABLES: DECLARATION AND ASSIGNMENT

In some languages to assign a value (put a value in the variable) the variable must first be declared. In other languages the variable is declared dynamically on assignment.

Declaration before assignment:

```
string var firstName  
firstName = "Richard"
```

Declaration with assignment:

```
string var firstName = "Richard"
```

Without formal declaration:

```
firstName = "Richard"
```

Notice that the variable in the example above is written in what we call “camel case”. This derives its name from the hump on a camel but in computer terms equates to declaring names of variables, with the first character of the first word with a small letter (lowercase) and the subsequent first character of all other words with a capital letter (Uppercase). Although this is not a fixed rule and won’t break the code, it is in line with what are called language coding standards or style guides. Other languages use underscores ‘_’ to separate variable words:

```
var first_name = "Richard"
```

See here for a Javascript style guide https://www.w3schools.com/js/js_conventions.asp . There are different style guides for different languages. For example, Pep8 for Python <https://pep8.org>

VARIABLES: EXPRESSIONS

A variable can be assigned a single value of a specific type or it may also be assigned the result of some form of expression.

An expression in programming is typically a line of code that resolves into a value. Expressions in their simplest form are assignments, i.e. 'x = 10'. Generally though you can think of them as code that uses some form of operator to deduce a value from a number of literal or variable values. Below are some example expressions.

Arithmetic:	<code>2 * 2 / 5, year + 10, 100 - age</code>
Comparison:	<code>age > 30, year >= 2000, name == "Tayfun"</code>
Logical:	<code>(age > 20) and (gender == "male"), stop or fail</code>
Strings:	<code>"I love" + birth_city</code>

VARIABLES: ASSIGNMENT EXAMPLES

In an assignment, there are two sides, the left side is a variable and the right side is either a literal or an expression that evaluate to a value. First the value is calculated and then it is stored in the variable.

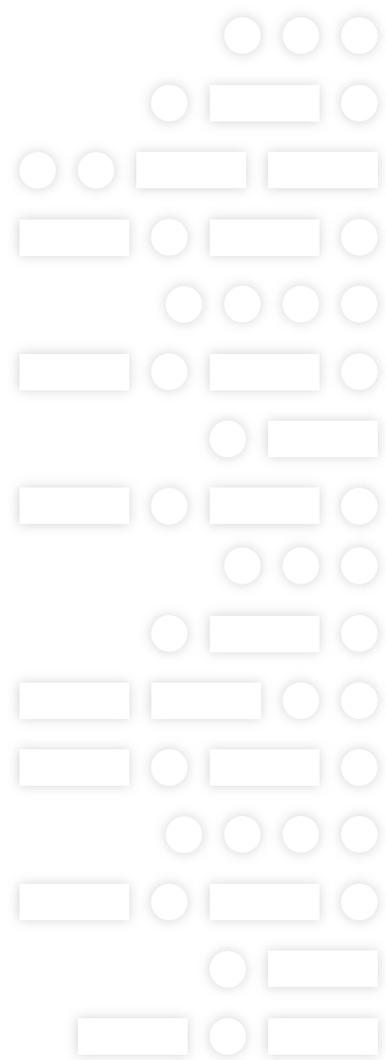
(LHS) = (RHS)

price = 100 + 50

tax = price * 0.1

price = price + tax

fullName = "Richard" + "Cheesmar"

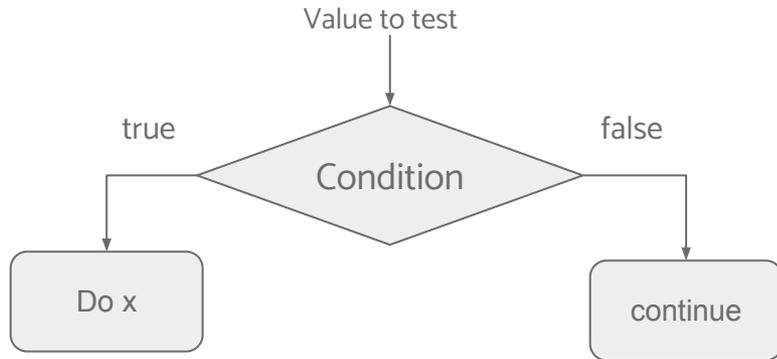


CONDITIONAL STATEMENTS



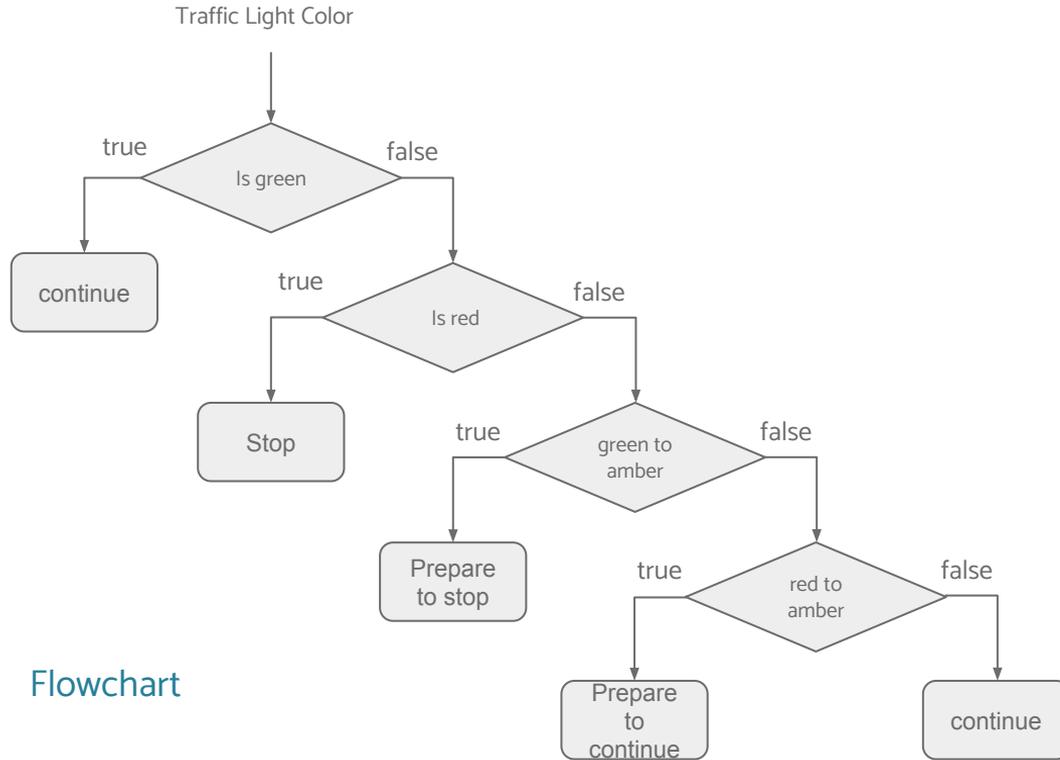
CONDITIONAL STATEMENTS: IF-THEN-ELSE

Conditional statements are tests on a specific value. The value may be a literal value or inside a variable. To test a condition of a value we predominantly, but not always, use the 'If-then' construct.



```
If speed > 90 Then  
    reduce speed  
End If
```

CONDITIONAL STATEMENTS: IF-THEN-ELSE



Flowchart

pseudo-code

```
If traffic-light is green Then  
    Continue  
Else If traffic-light is red Then  
    Stop  
Else if traffic-light is yellow from green Then  
    Prepare to stop  
Else if traffic-light is yellow from red Then  
    Prepare to continue  
Else  
    Continue driving carefully  
End If
```

CONDITIONAL STATEMENTS: SWITCH

A switch statement is a conditional statement that is often used instead of if then else statements depending on the test condition. Switch statements generally test against single values rather than expressions involving operators.

Larger value sets are more efficient when using a switch statement and switch statements are often more readable than multiple if-then-else statements. A general rule of thumb is if there are more than 5 conditions use a switch rather than if-then-else.

Switch traffic-light-color

Case "RED":

Stop car

Case "YELLOW":

Prepare to stop or move forward

Case "GREEN":

Continue driving

End Switch



PROCEDURES AND FUNCTIONS



PROCEDURES

A procedure is a way of running a list of sequential tasks. Expressions, should not generally occur in procedures, these are best placed in functions.

A procedure might call a collection of functions that perform calculation via expressions and return the values back to the procedure, which in turn passes the value to the next function in the sequence. A bit like a chain of function calls.

```
displayMaxTempProcedure
```

```
  londonTemp = getTemp("London")  
  parisTemp = getTemp("Paris")  
  istanbulTemp = getTemp("Istanbul")
```

```
  maxTemp = getMaxTemp([londonTemp, parisTemp, istanbulTemp])
```

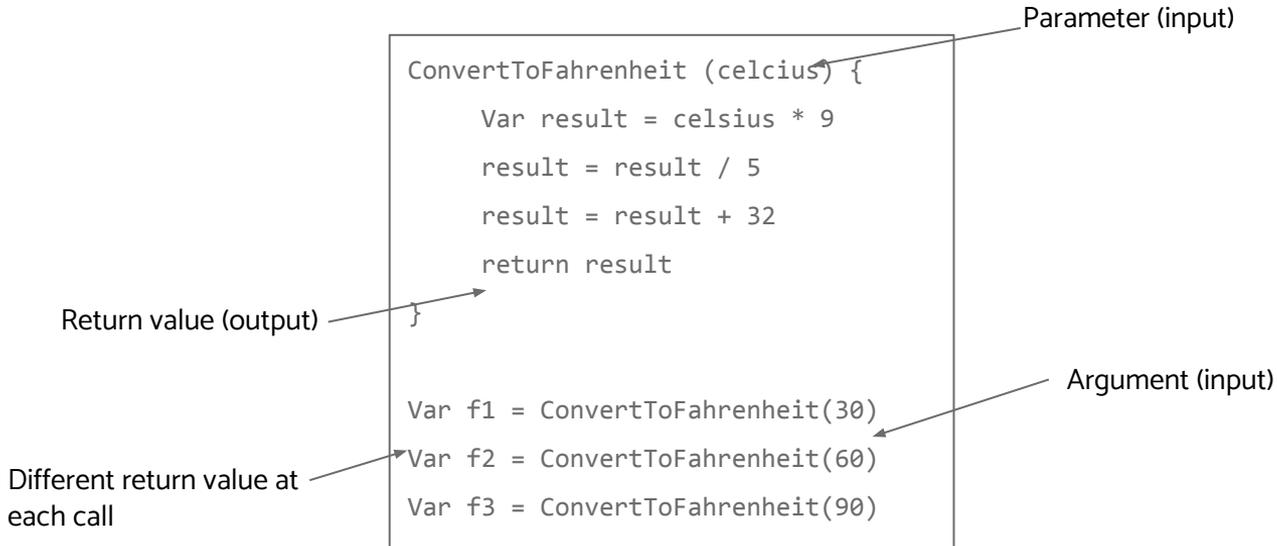
```
  print maxTemp
```

In the above example the `getTemp` is a function that gets the temperature.



FUNCTIONS

Unlike a procedure, a function is a block of code that is used to calculate something and commonly deal with expressions or some input output. It gets its name from mathematics. Functions can be called from procedures and other functions. Functions can even call themselves. This is known as recursion.



PROCEDURES AND FUNCTIONS: NOTES

While we have this distinction between procedures and functions, in some languages this distinction is implicit, i.e. down to the developer to create the appropriate structure. For example, in Javascript, both procedures and functions are written using the function construct and in Python we use def construct. And in some functional languages such as OCaml, there are no procedures only functions.



INPUT AND OUTPUT

Programming languages provide various methods for input and output. Depending on where the input is from and where it goes. At the basic level a language will provide constructs for getting input from the user and printing information to the screen.

The most basic form of user input and output can be done via what is called a console. A console is a program where you can write code line by line.

Below is are two examples of getting and displaying user input, using Python and Javascript.

```
>>> x = input("give me a number")
give me a number>> 10
>>> print(10)
10
>>>
```

```
JS
1 let x = prompt("Give me a number");
2 alert(x);
```

Try opening your browser console and running the javascript line by line and see what happens

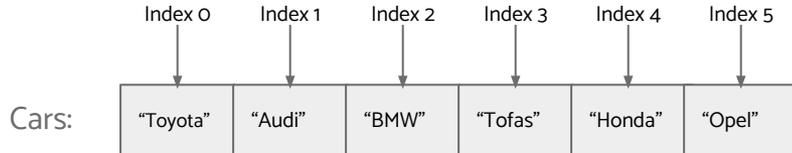
DATA STRUCTURES



DATA STRUCTURES: ARRAYS AND LISTS

Arrays and lists are containers that can contain any number of items. These items can be single values or more complex objects. Each item in the list has an index. An Index is the position of the item in that list. Arrays and lists are zero indexed, this means that the first item is always at position 0, the next at 1 etc. etc. So with a 3 item list the index positions would be [0, 1, 2] and NOT [1, 2, 3].

Arrays and lists can be declared as empty by using '[]' opening and closing square brackets. Most programming languages have some form of length function built in that can tell you the number of items in the list.



Array length: 5

```
#Declaring an empty array  
cars = []
```

```
# Creating a new array  
cars = ["Toyota", "Audi", "BMW", "Tofas", "Honda", "Opel"]
```

```
# Reading from array from the second index in the list  
mycar = cars[1]
```

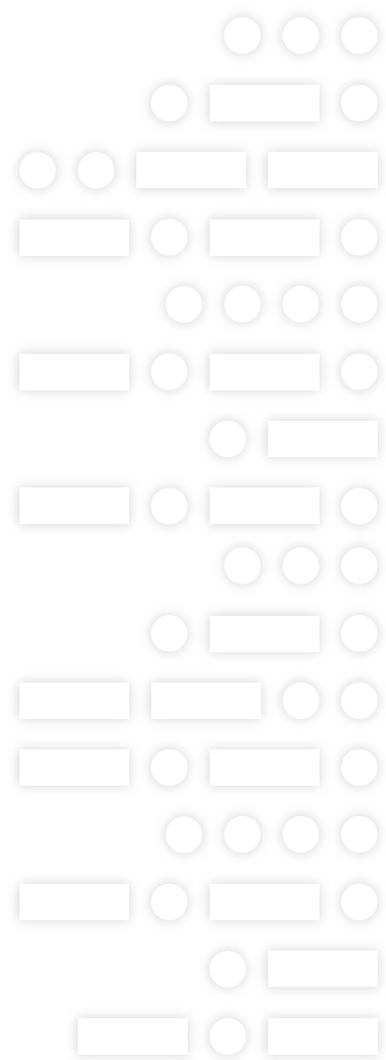
```
# Writing to array at the third position  
cars[2] = "Mercedes"
```

DATA STRUCTURES: ARRAYS AND LISTS

Programming languages use various methods for working with arrays and lists including adding, deleting and overwriting. Below you can see the methods for python

Method	Description
<code>append()</code>	Adds an element at the end of the list
<code>clear()</code>	Removes all the elements from the list
<code>copy()</code>	Returns a copy of the list
<code>count()</code>	Returns the number of elements with the specified value
<code>extend()</code>	Add the elements of a list (or any iterable), to the end of the current list
<code>index()</code>	Returns the index of the first element with the specified value
<code>insert()</code>	Adds an element at the specified position
<code>pop()</code>	Removes the element at the specified position
<code>remove()</code>	Removes the first item with the specified value
<code>reverse()</code>	Reverses the order of the list
<code>sort()</code>	Sorts the list

You can explore these methods at https://www.w3schools.com/python/python_ref_list.asp
And methods for javascript at https://www.w3schools.com/js/js_array_methods.asp



DATA STRUCTURES: SETS

Sets are **unordered iterable** data structures that can hold multiple types of **unique** and **immutable** items in a single variable. Once an item is in a set it cannot be changed, it is immutable.

```
set1 = set([1,4,7,9,0,3,6,5,2,8, "Richard", "Tayfun", "Richard", "Tayfun"])
```

Produces the data structure WITH NO DUPLICATES

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'Richard', 'Tayfun'}
```

Defining that same set again may produce the following order

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'Tayfun', 'Richard'}
```

Natively, Sets do not maintain an index related to where each item is, so you cannot ask for a Set item at index 3 as you can with an array or list. However, they allow you to find out if an item is in the set quite efficiently. This is because sets are hashable.

```
"Richard" in set1
```

For an object to be hashable it has to be immutable. A hash is a unique id that is created by applying a hashing algorithm to an object, be that object a data structure, an individual data item or even a function. As long as it keeps the same value for as long as the program executes it can be hashed. Hashing allows items in sets to be compared and referenced quickly. In fact it is the hashing that makes sets very fast when asking if they contain a specific data item.



DATA STRUCTURES: SETS

Sets are also iterable (can be looped over). It's important to remember that sets are unordered. Why, is a little complicated at this point but it is because the values are hashed. Small Sets may maintain an order but adding to the Set may well disrupt the order of the items. Never rely on a set to have a fixed order.

```
for item in set1:  
    print(item)
```

What are sets used for?

Sets are often used to create a unique set of data from a data structure such as an array or list.

Thus, **Sets are useful to create sets of unique data that will not change during the execution of a program.**

Another big advantage of using Sets is speed. If you have large settled data sets, then using Sets to access that data will definitely speed up your program. Sets are often used in data science specifically because the data sets are large and speed is imperative.



DATA STRUCTURES: TUPLES

Tuples are immutable (cannot be changed) like lists with fixed elements. Not every language has the tuple data type, but you can simulate it in most languages using lists and objects. However, the data in those are mutable (changeable).

Tuples are indexed just like lists. The first item in a tuple is always 0.
Tuples can store a mix of different objects from single literals to lists and other objects.
Once a tuple value is set you cannot change the value in the tuple.

Python has tuples as one of its four main object types, Lists, tuples, Sets and Dictionaries. Don't worry about Sets and Dictionaries yet, we'll get to those later.

Python uses parentheses '(') brackets to represent tuples:

```
Developers = ("richard", "Tayfun") is a valid python tuple declaration as is nums = (1,2,3) as is position = (1, "Richard")
```

Cool! But why use a tuple instead of a list? The main advantage is that they are indeed immutable. So if you know that a set of values will not and should not change during the execution of the code then tuples are a great way to safeguard this, especially when working in a team, where if you used a list some other programmer might well change the data inadvertently.

Also in python, using named tuples you can use the values in tuples as attributes see the code below

```
from collections import namedtuple

person = namedtuple('Person', 'first name, last name')
me = person(first_name="Richard", last_name="Cheesmar")
me.first_name
'Richard'
me.last_name
'Cheesmar'
```



DATA STRUCTURES: MAPS / DICTIONARIES

The data structures map and dictionary in computer science terms are distinct but very similar. In order not to confuse the situation we are going to describe them in their simplest definitions as containers for unordered key-value pairs. Where a 'key' is some literal identifier associated with another object, simple or complex, which is the value. Maps and Dictionaries are unordered and mutable. That is they do not maintain an order of items and every item is changeable.

Vehicle Prices:

Keys:	"Toyota"	"Audi"	"BMW"	"Tofas"	"Honda"	"Opel"
Values:	250	300	500	100	320	400

**To access elements of a map/dictionary you generally use '[']' square brackets and in some languages '.' dot notation - see examples below

Creating a new map

```
prices = { "Toyota": 250, "Audi": 300, "BMW": 500, "Tofas": 100, "Honda": 320, "Opel": 400 }
```

Reading from a map -

```
my_cars_price = prices["Toyota"]
```

Writing to array

```
prices["BMW"] = 600
```

A common data exchange format called JSON (Javascript Object Notation), which you'll use a lot uses the same structure to send and receive data compatible across any language that incorporates it, of which there are many.

DATA STRUCTURES: MAPS / DICTIONARIES

Maps and Dictionaries are often used to group related variables into structured objects. Which can be copied to other objects.

```
var car = {  
  brand = "Toyota",  
  model = "Corolla",  
  mileage = 100000,  
  color = "RED",  
  release_date = { day = 10, month = 8, year = 1025 }  
}
```

There are two common methods for accessing values depending on the language.: For example, to get the car brand we could use:

`car['brand']` or `car.brand`

Javascript allows both whilst Python uses the '[' notation, Although you can import some extra code to enable it.

Bear in mind that there are many helper methods for each language to access and manage map and dictionary keys and values..

// Written in Javascript - try it out in codepen

```
let car = {  
  brand: "Toyota",  
  model: "Corrola",  
  mileage: 10000,  
  color: "red",  
  release_date: { day: 10, month: 8, year: 2020 }  
};
```

```
let carMileage = {};
```

```
carMileage[car.brand] = car.mileage;
```

```
alert(carMileage['Toyota']);
```

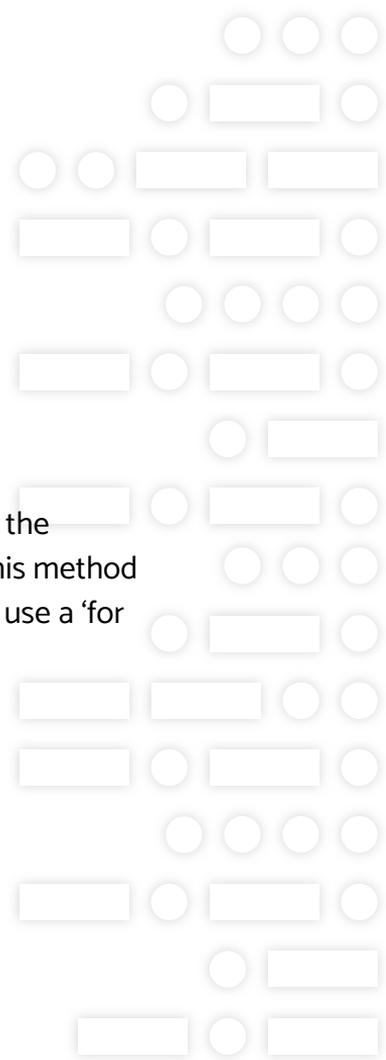
DATA STRUCTURES: ITERATING OVER STRUCTURES

When we use data structures we often need to iterate (run through all the values) and do some processing or calculations...

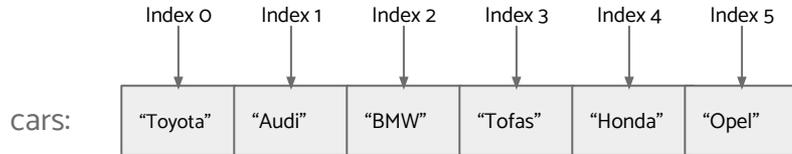
Programming languages offer iteration constructs that allow us to do just that. These are known as 'Loops', because we generally, but not always, loop over an object one item at a time.

There s one common way of looping over objects. The 'For Loop'

For Loops often use an explicit variable counter that gets incremented on each loop to a maximum (generally the number of items in the object) to loop over an object starting normally at the start (0) but not always. Using this method you can iterate over an object starting at any position you want, even in reverse if necessary. Some languages use a 'for each' or 'For in' construct which loops over each item in the object until the end is reached.



DATA STRUCTURES: ITERATING OVER STRUCTURES: FOR LOOP



```
For each car in cars do
    Print "I love my " + car
End
```

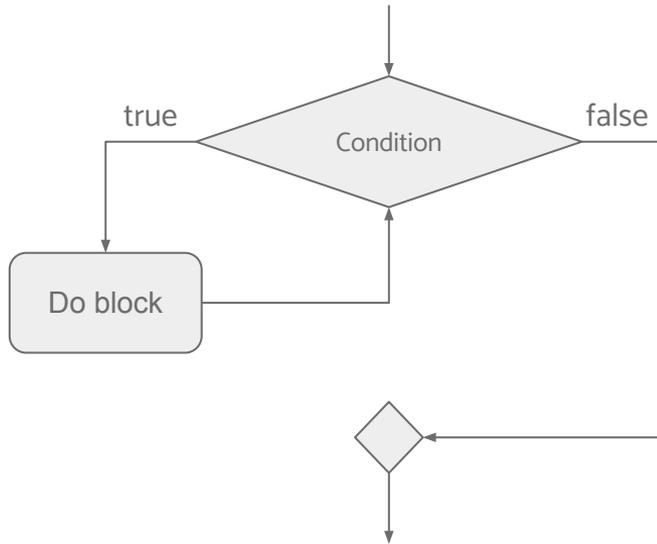
prices:

Keys:	"Toyota"	"Audi"	"BMW"	"Tofas"	"Honda"	"Opel"
Values:	250	300	500	100	320	400

```
For each (car, price) in prices do
    print "Price of " + car + " is " + price
End
```

ITERATING USING A WHILE LOOP

While loops are not suitable for object iteration but are very suitable to repeat some process or calculation until a certain condition is met. Below is a typical while loop.



```
number = 0
percentage = 10
while number < 20 Do
    print number + percentage
    number = number + 1
End
```



LIBRARIES AND MODULES

Libraries of code often called modules or packages are separate code bases (file or sets of files) that are designed to perform a specific task or set of tasks. For example there maybe a library for manipulating images, calculating statistics, networking, databases, file operations, math, sending email, anything... that has already been written by a third party, but we would like to use in our application, in order not to rewrite what has already been written (especially when it's complex), and speed up our development time.

Many libraries are included with the language itself as standard, whilst others are external and others we write ourselves for our applications.

Languages have their own internal frameworks for importing and using libraries. Specific library management code is used to find and install external libraries.

Generally an application is made up of semantically grouped code (code that handles specific aspects of an application). We normally separate these in specifically named directories, for example if we develop a shopping application we might separate out our code into the following modules:

‘customers’, ‘products’, ‘orders’, ‘sales’, ‘shipping’, ‘returns’, ‘accounts’ ‘utility functions’ etc. etc.

Thus, internal modules provide a coherent structure for our application development and maintenance.



OBJECT ORIENTED PROGRAMMING

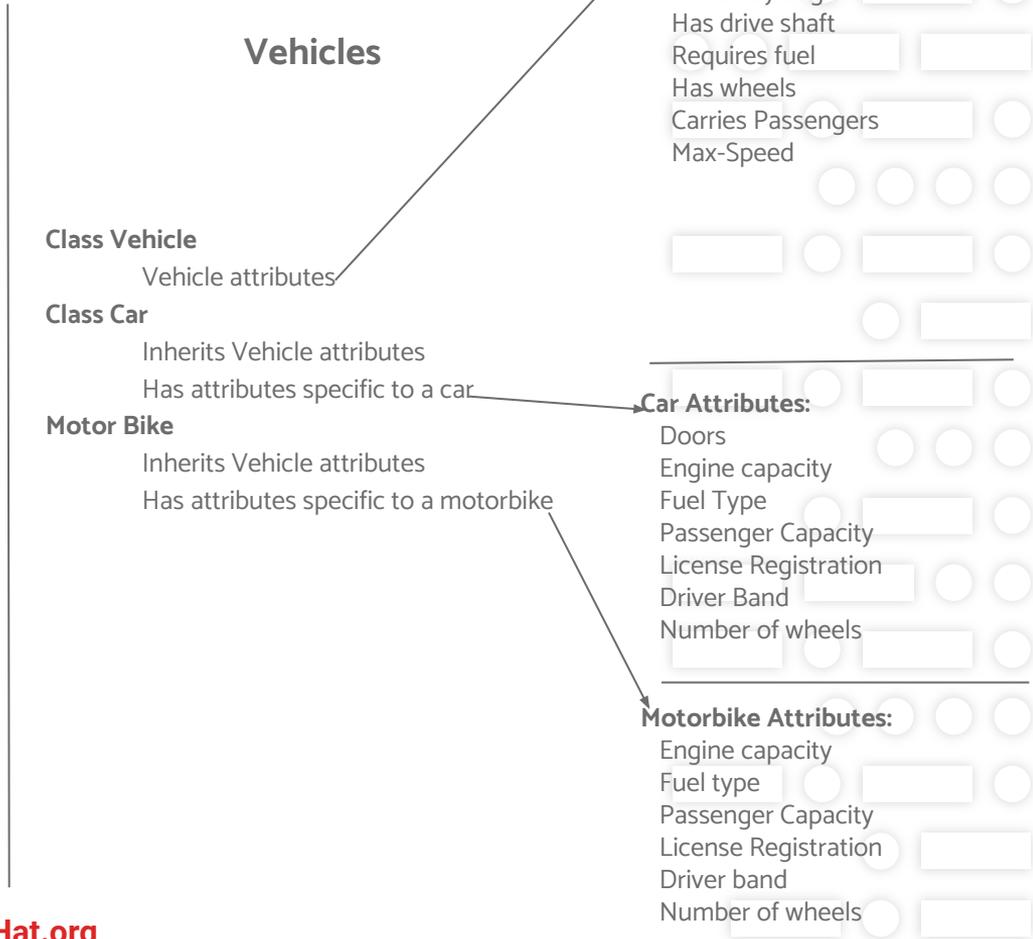


OBJECTS: INTRODUCTION

An object is a representation of some entity. It could be anything we know, a car, an apple a house, a computer, a tree, literally anything. All of these objects have certain attributes that make them what they are. For example a car has wheels, it requires fuel, it moves, it can be driven, it has a color, a brand, and a top speed plus many other attributes not least an engine. We could of course have a car without an engine, which would mean the car is not functional, but it would still be a car, just without an engine.

In object oriented programming paradigms we use objects to represent major components of our application. Each object is individual but belongs to a **class** of objects. The class defines the attributes of an object that are indisputable, in other words the basic attributes that make the object what it is. A class can also **inherit** from another class and place the object in some form of taxonomy. For example:

Both cars and motorbikes are vehicles because they inherit from Vehicles.



OBJECTS: INSTANTIATING

Everytime we create a new object from a class it is said to be instantiated (created). When an object is instantiated it becomes an instance of that class. In other words it is an individual object that has inherited some attributes from the class definition, just like we inherit some DNA from our parents but we are not our parents. We operate on an object as an individual not as a collective, although it shares attributes, the values are often different.

```
# Define our car and motorbike object classes
class Car(Vehicle):
    ...
class Motorbike(Vehicle)
    ...

# Create an Instance of the class car and bike
# Both Car and Bike are instances of Vehicle
my_car = new Car("Toyota", "Corolla");
my_bike = new MotorBike("BMW", "F800 GS");
```

Motor Vehicle Attributes:

- Moves
- Driven by engine
- Has drive shaft
- Requires fuel
- Has wheels
- Carries Passengers
- Max-Speed

Car Attributes:

- Brand = "Toyota"
- Model = "Corrola"
- Color = "red"
- Doors = 5
- Engine capacity = 1400cc
- Fuel Type = Benzine
- Passenger Capacity = 5
- License Registration = 48 12 ABC
- Driver Band = Passenger car
- Number of wheels = 4
- Max-Speed = 140
- Kilometers 25000

Motorbike Attributes:

- Brand "BMW"
- Model "F800 GS"
- Color = "blue"
- Engine capacity = 800cc
- Fuel type = Benzine
- Passenger Capacity = 2
- License Registration = 34 10 XYZ
- Driver band = Motorcycle
- Number of wheels = 2
- Max-Speed = 180
- Kilometers = 2000

OBJECTS: INHERITANCE

When we define an object that inherits we include the 'Parent' class, the class it inherits from in the definition. All of the attributes of the 'Parent' class will now be inherited by the new object. Below is a simple example from Python.

PARENT CLASS

```
class Vehicles:

    moves = True
    driven = True
    engine = True
    engine_capacity = None
    has_drive_shaft = True
    requires_fuel = True
    has_wheels = True
    carries_passengers = True
    max_speed = 100
```

CHILD CLASS

```
class Car(Vehicles):

    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
        self.number_of_wheels = 4
        self.working = False
        super().add_vehicle('car')
```

As you can see, the Car class inherits from Vehicles. When we create a new Car object we pass in the car brand and model. The `__init__` is what is called an object **constructor**. It initialises any default attribute values for the object. Different languages do object construction differently, but don't worry about that, it's the principle that is important here.



OBJECTS: INHERITANCE

Once a class object has inherited it is said to have a 'Parent'. In fact, an object can have more than one 'Parent' via what is called Multiple Inheritance, but we won't go into that here.

PARENT CLASS

```
class Vehicles:
```

```
    moves = True
    driven = True
    engine = True
    engine_capacity = None
    has_drive_shaft = True
    requires_fuel = True
    has_wheels = True
    carries_passengers = True
    max_speed = 100
```

```
@staticmethod
```

```
def add_vehicle(vehicle_type):
    if vehicle_type == 'car':
        Vehicles.num_cars += 1
    elif vehicle_type == 'bike':
        Vehicles.num_bikes += 1
```

```
def description(self):
```

```
    desc_str = "This vehicle is a %s %s %s - color %s with a top speed of %s km an hour."
    (self.brand, self.model, self.category, self.color, self.max_speed)
    return desc_str
```

CHILD CLASS

```
class Car(Vehicles):
```

```
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
        self.number_of_wheels = 4
        self.working = False
        super().add_vehicle('car')
```

Beyond inheriting attributes an object can inherit and use methods (functional blocks of code) that do stuff from the 'Parent' class. See the example to the left with a couple of methods included.

These methods will be available to the 'Child' objects.

```
car1 = Car('Toyota', "Corolla")
car1.max_speed = 160
car1.color = "red"
car1.category = "saloon-car"
print(car1.description())
```

The print statement will display the following

"This vehicle is a Toyota Corolla saloon-car - color red with a top speed of 160 km an hour."

OBJECTS: EXERCISE

Come up with at least two different examples of objects that have shared and individual attributes. Include one level of inheritance. Create a pseudo-code 'Parent' for each as well as the object itself.

Think about the attributes and how they are derived in terms of individuality or not as the case may be. Also think about what actions or common functionality might be included for objects to apply individually or via the 'Parent'.

Take your time and think about the details.

HANDLING ERRORS



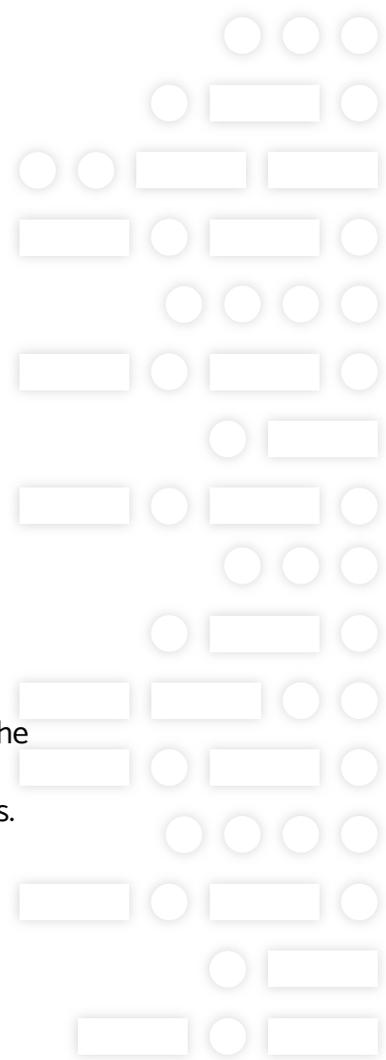
HANDLING ERRORS

A program will not always follow the “happy path”, things may fail in different ways:

- Input/output errors
 - User enters invalid values (incorrect email address, text instead of a number)
 - Fail to write to file, filesystem errors
- Programming errors
 - Invalue values for variables (null pointer)
 - Index out of bounds errors
- External factors
 - Network connection problems, server unreachable, slow network
 - Filesystem errors, no storage left
 - Power failure

Most all languages have a way of catching errors, but the programmer has to use them. If they are not used, the program will just crash with an error and users will be left wondering what happened. The art of dealing with errors is to deal with them gracefully. A lot of modern languages, use the ‘Try-except’ or ‘Try-catch’ paradigms.

In programming speak, errors are more often than not called exceptions.



HANDLING ERRORS: CATCHING ERRORS

Placing code under a 'Try' block and if an error occurs, catching it with the 'Catch' block. Catching the error allows you to understand exactly where the error occurred and what type of error it is.

Try do

```
x = input("Give me a number")
```

```
y = input("Give me a number")
```

```
z = x / y
```

```
print("Division result is " + z)
```

Catch error and do

```
If error is division by zero Then
```

```
    print("You entered 0 for divider!")
```

```
End if
```

```
End try-catch
```

Try do

```
message = input("What is your message?")
```

```
Send message to server
```

```
print("Message sent")
```

Catch error and do

```
If error is "connection lost" Then
```

```
    Resend message to server
```

```
Else If error is "cannot reach server" Then
```

```
    print("Cannot reach server")
```

```
    Stop operation
```

```
End if
```

```
End try-catch
```

HANDLING ERRORS: CATCHING PROGRAMMING ERRORS

Some errors occur because the logic of our code is wrong. This happens predominantly when things are rushed, i.e. code without design... Other occur because some data is incorrect... whilst still others occur through something that is not always within our control. However, we can prepare for these.

The code on the right represents a conditional block of code to send a message to an email address.

What this does is check an email address is in the correct format. If it is it attempts to send a message. If it sends it exits the block.

If there is a network error it will try again, a maximum of 3 times. If it sends it will exit the block and returns 'email sent'

If it does not send after 3 attempts it will return the appropriate message.

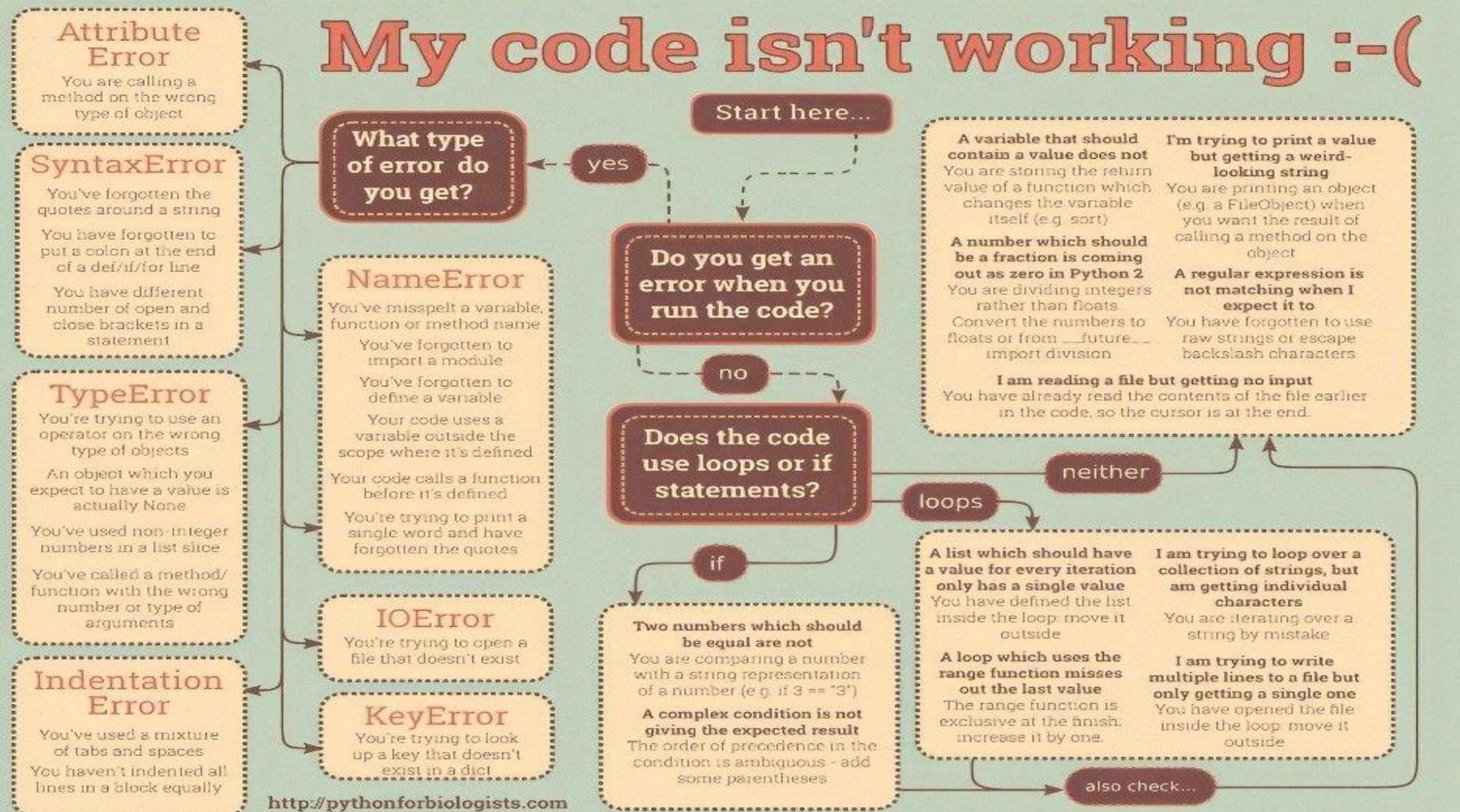
If the email is in an incorrect format it will return with appropriate message.

```
attempts = 0

If email_address is in correct format Then
    status = send_message(message, email_address)
    While attempts < 3 and status is network error Do
        attempts = attempts + 1
        Wait 3 seconds
        status = send_message(message, email_address)
    End while
Else
    Return email address is incorrect
End If

If status is ok:
    Return email sent
Else:
    Return could not send email - Network Error
```

My code isn't working :-)



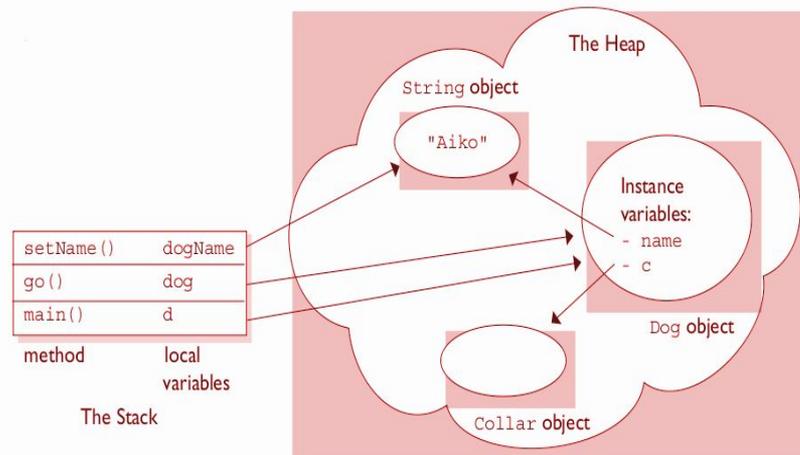
MEMORY MANAGEMENT BASICS



MEMORY MANAGEMENT BASICS: A BRIEF INTRODUCTION

The following brief introduction to this subject is not meant to be a deep dive into memory management, but to provide a basic understanding of the basics and to provide you, the reader, with a good enough level of understanding to dig deeper without hesitation.

Managing memory is the primary responsibility of the operating system, which allocates chunks of available memory to various running applications. However, programming languages also have memory management and liaise with the operating system to acquire chunks of memory for their code and data. Different languages do this in different ways. For example Python has an Python Memory manager under the hood that takes care of more or less everything to do with memory allocation and deallocation, whilst 'C' and it's variants leave a lot of memory management to the discretion of the developer. Either way, it is important to understand the basic memory types and what they are used for.



MEMORY MANAGEMENT: MEMORY TYPES

There are several types of memory when it comes to programming languages, Code, Heap, Stack, and Cached memory.

Code

Code memory is where the instructions for your code to run are stored. As Python is an interpreter, the memory for code is allocated when each line of code is taken from the compiled bytecode and translated into runnable machine code.

Heap

Heap memory is a non-static, dynamically allocated memory that is resizeable and non-contiguous. Heap memory is akin to global memory in as much as anything stored in heap is available from anywhere in your code. Everything stored in heap memory can change, i.e. when you add, modify or remove data. For this reason heap memory can become fragmented, chunks of related data may not be in contiguous order making access slower than it would be if they were. Heap memory may be stored in RAM or on hard disk if RAM becomes tight with lots of applications open simultaneously. The operating system tries to maximise the efficiency of RAM depending on what applications are running and their individual memory consumption.

Python has a private heap where it stores data structures and objects and uses a number of object-specific allocators that take care of allocating different types of objects and data structures to memory blocks. For example, an allocator for taking care of integers and another for dictionaries and lists etc. All these allocators are managed via the Python Memory Manager, and unlike when using a language like 'C', Python developers have no control over where data is stored and how much memory is allocated. 'C' on the other hand allows developers to get their hands dirty offering far greater control of how memory is allocated and deallocated.

MEMORY MANAGEMENT: MEMORY TYPES

Stack

Unlike heap memory, stack memory is both static and linear and used to store local data as the code is running, such as functions local variables. Anything stored on the stack is fixed in size and cannot be changed.

Stack memory cannot be reallocated and has a LIFO (last-in first out) paradigm. Imagine stacking some books on top of each other. When unstacking the books the last one stacked is removed first, freeing up space. That is how stack memory is managed.

Frequently, references to functions and procedure calls and local variables are stored on the stack. Once a function returns control, i.e. finishes processing, that stack memory is released from the stack. This ensures that the stack memory does not get bloated.

Cache

Cache memory is fast access memory that generally lives in RAM. RAM is extremely fast compared to disk memory. Frequently used data is often stored in cache memory. How much data stored in cache depends on the size of the RAM on any individual computer. Most applications try to take advantage of cache memory as much as possible, but it's the operating system that will manage the allocation of cache distribution amongst applications. A number of databases use cache for storing data, and automatically load data from files on disk into cache. This is particularly relevant for a lot of NOSQL databases.

Once an application is terminated and or a computer shutdown, anything in cache, unless saved to disk, is permanently lost. You cannot recover data from cache memory as you can from disk files.

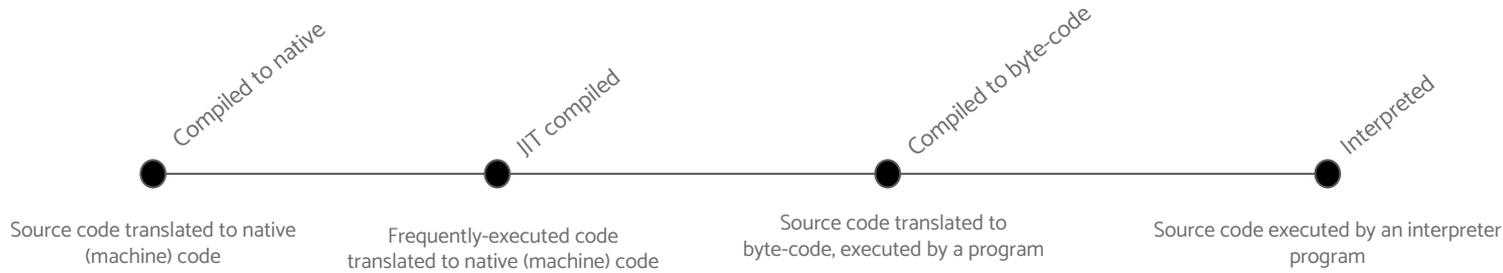


HOW PROGRAMS ARE EXECUTED (RUN)



EXECUTABLE CODE: HOW

Depending on the language used, the code you write follows different paths to executable form. Executable form is the form of the code that can be executed by the computer, i.e. machine code. You can't just write code in a text-file and tell it to run on its own. It requires an interpreter or a compiler.



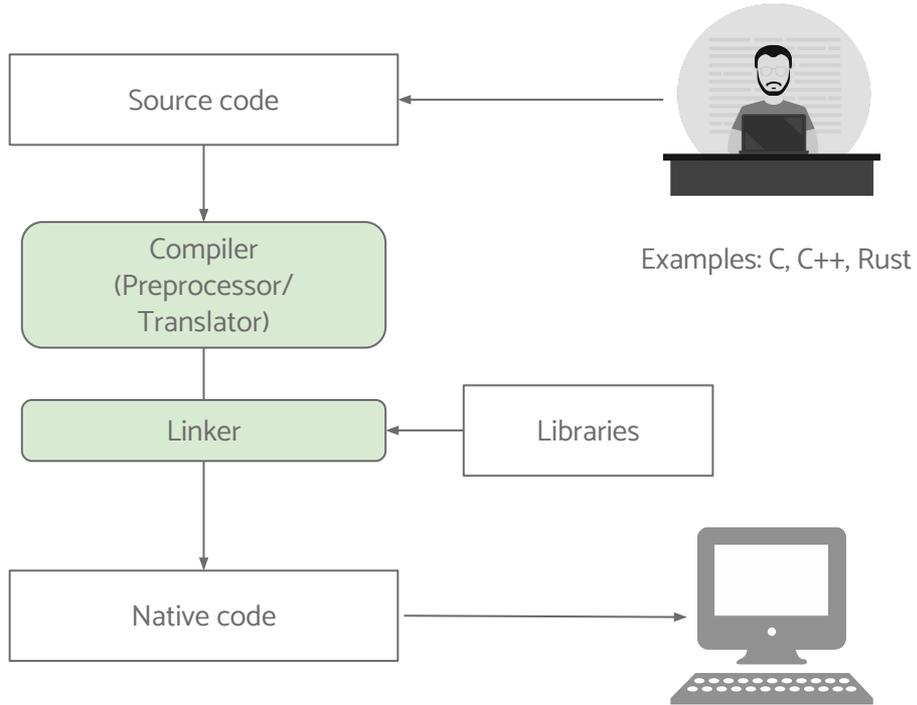
A compiler does what it says on the tin, it compiles the source code either directly to machine code or byte-code which is then run by a program, which in turn will execute the compiled code step by step. Byte-code is a low level set of instructions that can be run by a program which executes those instructions, turning them into machine code.

An interpreter, takes code and interprets that code using a virtual machine (a program that can understand the source code, i.e. interpret it into a form of machine code to get it to execute. It does this one line at a time.

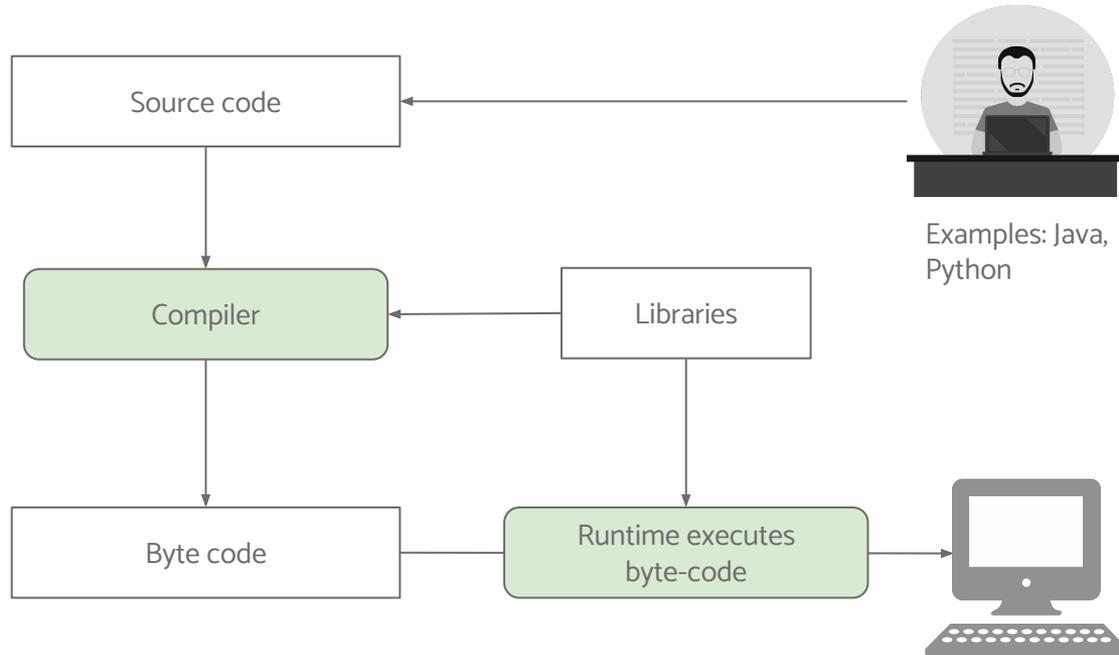
Sometimes code, i.e. Python code, is first compiled into byte-code and then interpreted by a python virtual machine.

Compilers store the machine code whereas Interpreters do not, it's run and forgotten. In the early stages of the IT revolution the distinction between Compilers and Interpreters was clear. Today there is an array of approaches that have to be understood individually, depending on the language in use.

EXECUTABLE CODE: SOURCE CODE COMPILED TO NATIVE (MACHINE) CODE



EXECUTABLE CODE: SOURCE CODE COMPILED TO BYTECODE



EXECUTABLE CODE: JAVA BYTECODE

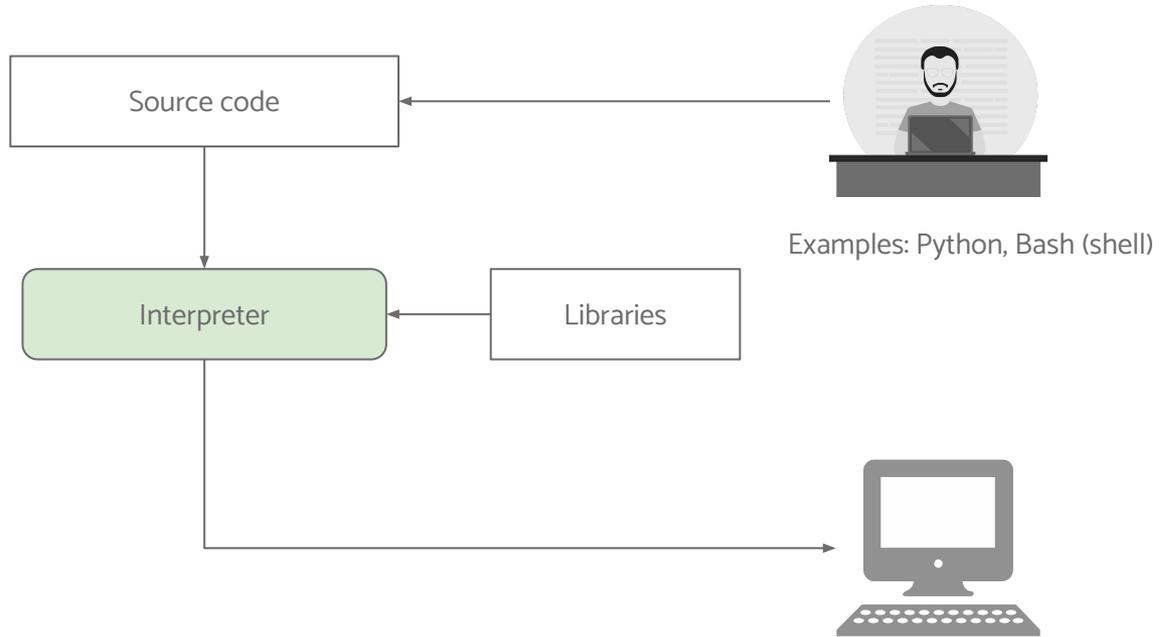
```
public class TemperatureConverter {  
    public TemperatureConverter() {  
    }  
  
    public double convertToFahrenheit(double celcius) {  
        double result = celcius;  
        result = result * 9;  
        result = result / 5;  
        result = result + 32;  
        return result;  
    }  
}
```



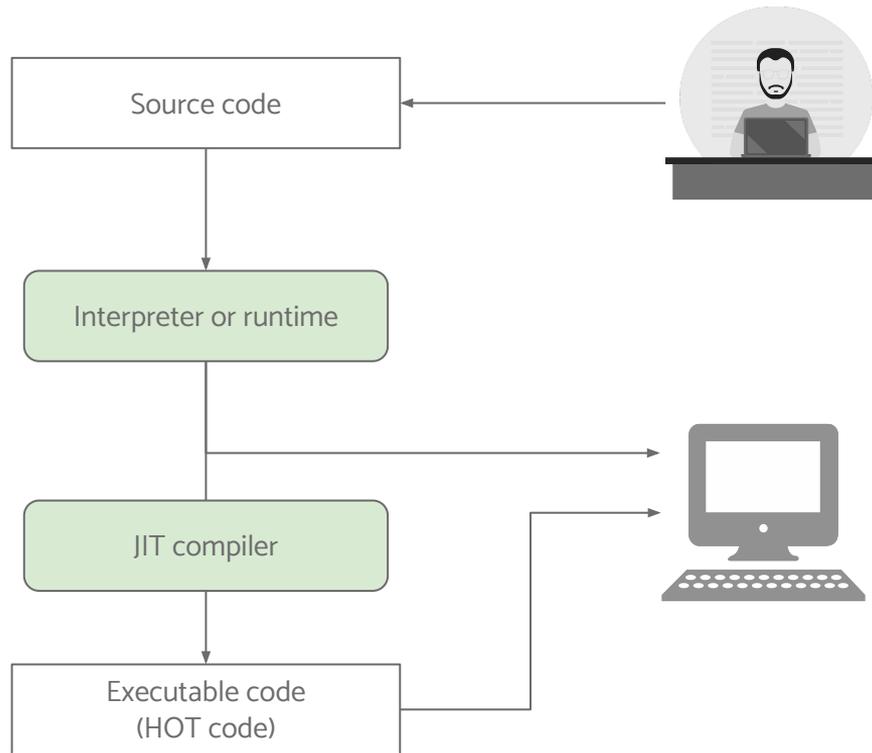
javac

```
public class TemperatureConverter {  
    public TemperatureConverter();  
    Code:  
    0: aload_0  
    1: invokespecial #1           // Method  
    java/lang/Object."<init>":()V  
    4: return  
  
    public double convertToFahrenheit(double);  
    Code:  
    0: dload_1  
    1: dstore_3  
    2: dload_3  
    3: ldc2_w           #7           // double 9.0d  
    6: dmul  
    7: dstore_3  
    8: dload_3  
    9: ldc2_w           #9           // double 5.0d  
    12: ddiv  
    13: dstore_3  
    14: dload_3  
    15: ldc2_w           #11          // double 32.0d  
    18: dadd  
    19: dstore_3  
    20: dload_3  
    21: dreturn  
}
```

EXECUTABLE CODE: SOURCE CODE INTERPRETED AT RUNTIME



EXECUTABLE CODE: FREQUENTLY-EXECUTED CODE COMPILED TO NATIVE CODE AT RUNTIME



CONCLUSION

We have learnt a lot about programming languages, their common constructs, and how they work.

The information given, really only touches the surface. Today there are so many programming languages out there, to understand them is a continuous process of learning.

If indeed you do pursue a career in software development, you will over time become more and more knowledgeable.

Good Luck!