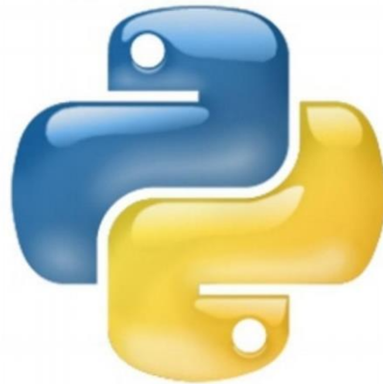


# Object-Oriented Programming In Python Workbook

**Damian Gordon**

**2020**

Feel free to use any of the content in the guide with my permission.

Any suggestions or comments are most welcome, email me Damian.X.Gordon@TUDublin

# Table of Contents

*1. Object-Oriented Design*

*2. Objects in Python*

*3. Modules and Packages*

*4. Inheritance and Polymorphism*

*5. Object-Oriented Programs*

*6. String Formatting*

*7. Regular Expressions*

*8. Object Serialization*

*9. Design Patterns*

*10. Comprehensions and Generators*

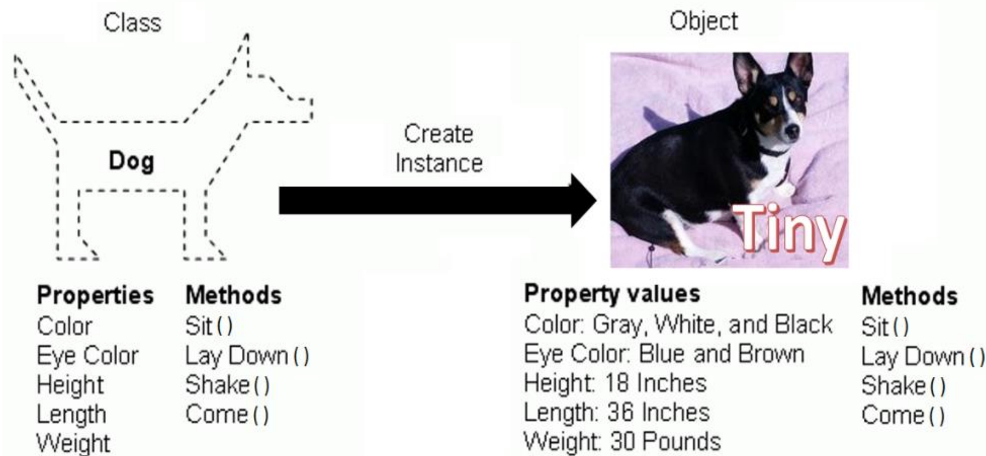
*11. Object-Oriented Testing*

*12. Development Models*

# Object-Oriented Design

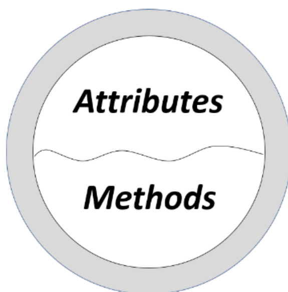
### What is a Class?

Creating a Class is like creating a variable, it's just a definition of a data type until it gets assigned a value, once you have an instance of a Class, we call it an Object. If our class is "Dog", we could have objects "Tiny", "Rover", "Rusty", and "Fido".




### What is an Object?

An object has:
Attributes and Methods




- *Attributes* (or *Features*) are a collection of variables.
- *Methods* (or *Behaviours*) are a collection of functions.
- The Attributes and Methods are *encapsulated* or contained in the object.
- The object can be configured so that some Attributes and Methods are private to the object, and others are visible to other objects, this is *Information Hiding*.
- The public elements (Attributes and Methods) of the class are referred to as the *Interface* (or *Public Interface*).

### Object-Oriented Concepts

| **Abstraction**<br>Abstraction means dealing with the level of detail that is most appropriate to a given task. For example, a driver and a mechanic interact with a car at different levels of abstraction. | **Inheritance**<br>Inheritance means that one class can inherit attributes and methods from another class. So we look for classes that have a lot in common, and create a single class with those commonalities. |
|---|---|
| **Composition**<br>Composition means collecting several objects together to create a new one. It is usually a good choice when one object is part of another object. | **Polymorphism**<br>Polymorphism is the ability to treat a class differently depending on which subclass is implemented. The appropriate subclass is determined based on parameters passed in. |

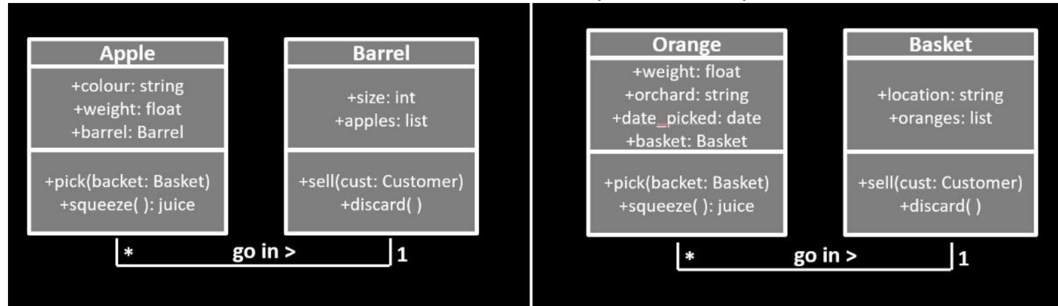
We'll look at these concepts in more detail later.

Object Oriented Programming © Damian Gordon

# Object-Oriented Design

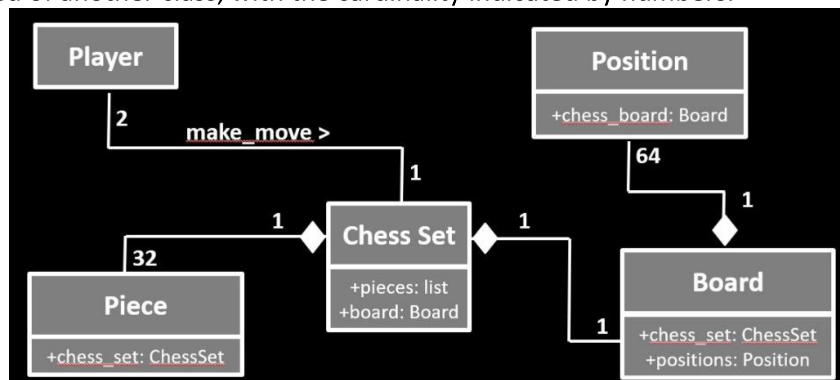
### Class Diagrams

A Class Diagram shows the design of a system showing the classes, and their attributes and methods. It also shows the relationship between classes, with a line (association) between classes, with a verb to describe the relationship, and cardinality is indicated by numbers, in this case 1…* indicates a one-to-many relationship.



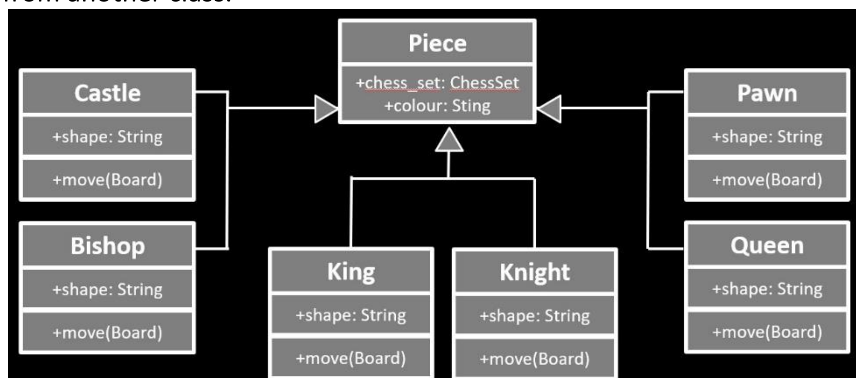

### Class Diagrams (composition)

A Class Diagram can show composition using a solid diamond to indicate that one class is composed of another class, with the cardinality indicated by numbers.




### Class Diagrams (Inheritance)

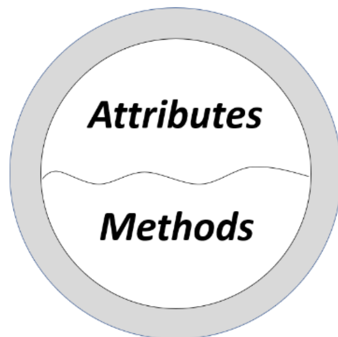
A Class Diagram can show inheritance using an arrowhead to indicate that one class inherits from another class.




We'll look at these concepts in more detail later.

# Objects in Python

We remember, that an object has
Methods and Attributes:




- To add METHODS, you must add them to the CLASS definition.
- To add ATTRIBUTES, you typically add them to the CLASS definition, but it is possible to add them to the OBJECT definition also. You must add the values of the attributes to the OBJECT.

To declare a class in Python:

```
class <ClassName>:
    <Do stuff>
# END Class
```

For example:

```
class Point:
    pass
# END Class

p1 = Point()
p2 = Point()
```

To add attributes:

```
p1.x = 5
p1.y = 4
p2.x = 3
p2.y = 6
```

In Python the general form of declaring an attribute is as follows:

```
OBJECT.ATTRIBUTE = VALUE
```

We call this *dot notation*.

To add a method, we can simply put it in the class:

```
class Point:

    def reset(self):
        self.x = 0
        self.y = 0
    # END Reset

# END Class
```

In this case the use of the term `self` refers to the current instance of `Point`, so whichever object of `Point` you call the `reset` method with, is the `self`.

We can instantiate the class as follows:

```
p = Point()
p.x = 5
p.y = 4
```

We can call reset in two ways:

```
>> p.reset()
>> Point.reset(p)
```

Object Oriented Programming © Damian Gordon

# Objects in Python

**The Initialization Method**

Sometimes when a developer creates a new class they forget to declare all of the attributes required, or forget to give those attributes a starting value ("initial value"). So Python has a special method called "`__init__( )`" which forces the developer to make sure they declare and initialize all of the attributes that are required by each program. It can also be declared in such a way as to set default values for all attributes.

| This is an Initialization method without any default values: | This is an Initialization method out default values pre-set: |
|---|---|
| `def __init__(self,x,y):`<br>`    self.move(x,y)`<br>`# END Init` | `def __init__(self, x=0, y=0):`<br>`    self.move(x,y)`<br>`# END Init` |

The Initialization method should be used in every class unless there is a very good reason.

**Docstrings**

One of the key motivations behind the object-oriented paradigm is the idea of "code reuse", so if you (or someone else) have already written a method or class, if it is at all possible you should not rewrite the code again. Good documentation and labelling are an important element of making the purpose and intent of programs clear. One feature that Python provides to make this easier is the *Doscstring* feature, where you can add descriptive sentence to each class and method, as the first line of that class or method.

```
class Point:
    "Represents a point in 2D space"

    def move(self,a,b):
    'Move the point to a new location'
        self.x = a
        self.y = b
    # END Move

# END Class
```

In the Python shell, all you need to do is type "help" with the name of the class in brackets, and you will see all the *Docstrings*.

```
>>> help (Point)

Help on class Point in module __main__:
class Point(builtins.object)
 |  Represents a point in 2D space
 |
 |  move(self, a, b)
 |      Move the point to a new location
 |  ---------------------------------------
```

So the purpose of all of the methods is made clear using the HELP command.

# Modules and Packages

**MODULES**

- If we have two python programs in the same folder, and we want one to use methods and attributes from the other, all we need to do is say:

```
import <Filename>
```

**Modules are files, Packages are folders**

- To use specific class from a program (module) in the same folder, we can say:
    - `from <Filename> import <Classname>`
- If there already is a class in the calling program as the name of the class we want to import, we can import it with an alias:
    - `from <Filename> import <Classname> as <Alias>`
- If we want to import two classes from the same program (module):
    - `from <Filename> import <Classname1>, <Classname2>`
- If we want to import all the classes (which isn't really a good idea), we can say:
    - `from <Filename> import *`

**PACKAGES**

- If we feel there are too many programs (modules) in the current folder, we can create a sub-folder (package) to put some in that. Then to make it possible to access the programs in that subfolder we need to add a blank textfile into the subfolder called the following:

```
_ _init_ _.py
```

- To import a full file from that subfolder:
    - `import <SubFolder>.<Filename>`
- To import a specific class from a file in that subfolder:
    - `from <SubFolder>.<Filename> import <Classname>`
- We can also say:
    - `from <SubFolder> import <Filename>`

Object Oriented Programming © Damian Gordon

# Modules and Packages

**Tips for Using Modules and Packages**

| |
|---|
| We can use the full stop "." to refer to the current directory. |
| We can put text in the `__init__.py` file to import code directly from a package as opposed to a module. |
| The "global" command allows us to create a global variable, and we can make changes to that variable in a local context. |
| If we want to check if a program is being called from another program, or is being run as a stand-alone program, we can say: `if __name__ == "__main__":` If this is true, it means the program is being called as a stand-alone. |

These allow us to get the most out of modules and packages.

**Access Control**

Many object-oriented programming languages allow the programmer to specific the level of access to the methods and attributes of an object, using *Public*, *Protected*, and *Private*. Python does not have an equivant access control mechnaism, but by convention a method or attribute with no underscores at the start of its name is consdiered to be public (any object can access it), a method or attribute with a single underscore (_) at the start of its name indcates it is protected (the object itself and subclasses can access it), and a method or attribute with a double underscore (_ _) at the start of its name indcates it is private (only the object can access it), but remember there is nothing in the interpreter to stop external objects from access these methods or attributes.

We can also add a comment in the *docstrings* at the start of the class indicating which methods or attributes are for internal use only.

**Third-Party Libraries**

Python comes with a very big Standard Library with lots of features, but we may be looking for a feature that it doesn't have, if so we have two options; we can write a new package ourselves, or we can use somebody else's code.

If we want to find packages that might be of use, we can use the Python Package Index (PyPI) website: `http://pypi.python.org`.

Once we've found a package that we want to install, we can use the `pip` tool to install it. Pip doesn't come with the Python download, but we can download it from here: `https://pypi.python.org/pypi/pip`

And once pip has been installed, we can install the software into our library using:

- `pip install <package>`

# Inheritance and Polymorphism

**Basic Inheritance**

We have mentioned before that software re-use is considered one of the golden rules of object-orientated programming, and that generally speaking we will prefer to eliminate duplicated code whenever possible. One mechanism to achieve this is *inheritance*, which means that the attributes and methods of one class are available to another one in such a way the it appears those attributes and methods were declared within the second class. We call the first class, the *parent class* or the *superclass*, and the second class is called the *child class*, or *subclass*. To declare that one clas is a subclass of another, when you are declaring that class, just put the name of the superclass as a parameter in the class declaration of the subclass:

```
class Subclass(Superclass):
      <Class Declaration>
# END Class.
```

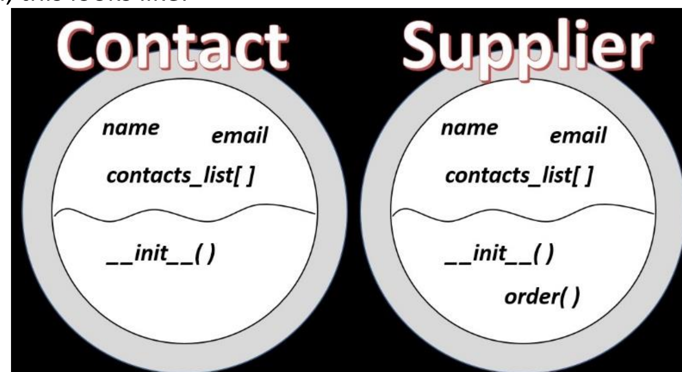
Let's look at an example in practice. Let's say we have a simple address book that keeps track of names and e-mail addresses. We'll call the class Contact, and this class keeps the list of all contacts, and initialises the names and addresses for new contacts:

```
class Contact:
    contacts_list = []
    def _ _init_ _(self, name, email):
        self.name = name
        self.email = email
        Contact.contacts_list.append(self)
    # END Init.
# END Class.
```

Now let's say that the contact list is for our company, and some of our contacts are suppliers and some others are other staff members in the company. If we wanted to add an order method, so that we can order supplies off our suppliers, we need to do it in such a way as we cannot try to accidently order supplies off other staff members. So we do:

```
class Supplier(Contact):
   def order(self, order):
       print("The order is for"
          "'{}' to '{}'".format(order,self.name))
   # END order.
# END Class.
```

So as a diagram, this looks like:



## Inheritance and Polymorphism

So we can declare objects of the classes:

```
c1 = Contact("Tom StaffMember", "TomStaff@MyCompany.com")
s1 = Supplier("Joe Supplier", "JoeSupplier@Supplies.com")
```

And if order something from our supplier by saying `s1.order("Bag of sweets")` we get the following message back:

```
The order is for 'Bag of sweets' to 'Joe Supplier'
```

But if we tried to order from our contacts by saying `c1.order("Bag of sweets")` we get an error message back:

```
Traceback (most recent call last):
  File "C:/Inheritance.py", line 64, in <module>
    c1.order("Bag of sweets")
AttributeError: 'Contact' object has no attribute 'order'
```

Because the method `order` exists only in the class `Supplier`, not in `Contact`.

**Overriding and Super**

Overriding means that Python allows a superclass and a subclass to have methods of the same name, and objects of each particular class can use the method associated with that class, by calling it in the normal way.

Super is a function, typically called as `super()`, that allows a subclass to call a method in a superclass, by saying `super().SuperclassMethod`.

**Multiple Inheritance**

A subclass can inherit for more than one superclass in a very simple way:

```
class Subclass(Superclass1, Superclass2):
      <Class Declaration>
# END Class.
```

In this scenario, when a specific attribute or method is mentioned, the Python interpreter first looks for it in the current class, and if it isn't there the interpreter will check the first superclass for the attribute or method, and if it isn't found there the interpreter will check if that superclass itself has a superclass, if so it will check that one, if not it will move onto the second superclass (this is called a depth-first search).

**Polymorphism**

Polymorphism means that we can call the same method name with different parameters, and depending on the parameters, it will do different things. For example:

| | |
|---|---|
| >>> print(6 * 5) | [*30*] |
| >>> print("Hello" * 5) | [*HelloHelloHelloHelloHello*] |

# Object-Oriented Programs

**Moving from Procedural to Object-Oriented Programs**
One of the key goals of object-oriented programming is software reuse. To achieve this we wrap methods and attributes in a class, and that makes it easier for other programs to use those classes. If we are just modelling data, maybe an array is all we need, and if we are just modelling behaviours, maybe some methods are all we need; but if we are modelling both data and behaviours, an object-oriented approach makes sense.

In this example we are calculating the perimeter of a shape. The code in **black** is the procedural version of the program, and the code in **red** is what we need to add in to make it object-oriented. We need to add in an `init` method for each class, and we also need to add in a method to take point values into the class, because with object-oriented design we prefer to encapsulate the values, and change them through a method.

```
# PROGRAM CalculatePerimeter:
import math

class Point:
    def _ _init_ _(self, x, y):
        self.x = x
        self.y = y
    # END init

    def distance(self, p2):
        return math.sqrt(
            (self.x – p2.x)**2 + (self.y – p2.y)**2)
    # END distance

# END class point

class Polygon:
    def _ _init_ _(self):
        self.vertices = []
    # END init

    def add_point(self, point):
        self.vertices.append((point))
    # END add_point

    def perimeter(self):
       perimeter = 0
       points = self.vertices + [self.vertices[0]]
       for i in range(len(self.vertices)):
           perimeter += points[i].distance(points[i+1])
       # ENDFOR
       return perimeter
    # END perimeter

# END class perimeter

# END.
```

The object-oriented code in **red**, doubles the length of the program.

/

# Object-Oriented Programs

**Running the Program**

Below is how we would run the program procedurally, and how we would run it in an object-oriented way. As we can see in the procedural version we input the perimeter points directly, whereas in the object-oriented version we input them via a method. The object-oriented version certainly isn't more compact than the procedural version, but it is much clearer in terms of what is happening, and makes reuse far easier.

**Procedural Version**

```
>>> square =
[(1,1),(1,2),(2,2),(2,1)]

>>> perimeter(square)
```

**Object-Oriented Version**

```
>>> square = Polygon()
>>> square.add_point(Point(1,1))
>>> square.add_point(Point(1,2))
>>> square.add_point(Point(2,2))
>>> square.add_point(Point(2,1))
>>> print(square.perimeter())
```

**Getters and Setters**

As we mentioned, we prefer to access attributes through methods instead of accessing them directly.There are times when that applies to the internal code as well as to other classes. For example, if we want to assign a variable called `name` to a particular value, we would say something like `Colour.name = "Red"`, but, we could also write a method as follows, and to assign a value we would say `Colour.set_name("Red")`.

```
def set_name(self, name):
    self._name = name
# END set_name.
```

We can do the same for the command `print(Colour.name)` which we can change to become `print(Colour.get_name()` [We can do the same for a `return`]:

```
def get_name(self):
    return self._name
# END get_name.
```

The real benefit of getters and setters is that we can add conditions and checking into the getters and setters to make the code more robust and powerful:

```
def get_name(self):
    if self._name == "":
        return "There is a blank value"
    else:
        return self._name
    # ENDIF;
# END get_name.
```

If we have code that already does assignments and prints, we can force them to run as getters and setters using the property function as follows:

```
>>> name = property(_get_name, _set_name)
```

And now without having to change any code, the assignments, prints, and returns are upgraded to become getters (aka *accessor methods*) and setters (aka *mutator methods*).

**Manager Objects**

Manager Objects are like managers in offices, they tell other people what to do. The manager class and objects don't really do much activity themselves, and instead they call other methods, and pass messages between those methods.

Object Oriented Programming © Damian Gordon

# String Formatting

**String Declarations**

There are a lot of different ways to declare a string in Python, you can use a set of double quotes (“) or a set of single quotes (‘). You can also create a string be enclosing a number of strings in round brackets (generators). To declare a sting over multiple lines all you have to do is enclose the strings in three double quotes (“) or three single quotes (‘).

| Code | Type | Value |
|---|---|---|
| `a = "Hello"` | String | Hello |
| `b = 'World'` | String | World |
| `c = ("Three " "Strings " "Together")` | String | Three Strings Together |
| `d = '''a multiple line string'''` | String | a multiple line string |
| `e = """More Multiple Strings"""` | String | More Multiple Strings |

**Counting and Searching for a Character**

Python strings have a number of built-in methods, including `count` which counts how often a particular chatacter or substring appears in a string. The `find` method locates the first occurance of a particular chatacter or substring (starting at location zero). The `rfind` (reverse find) method locates the last occurance of a character or substring.

| `s = "Hello World"` | |
|---|---|
| `s.count('o')` # How often does ‘o’ appear in s? | 2 (‘o’ appears twice) |
| `s.count('l')` # How often does ‘l’ appear in s? | 3 (‘l’ appears three times) |
| `s.find('o')` # What position is the first ‘o’ at? | 4 (starting at location zero) |
| `s.find('l')` # What position is the first ‘l’ at? | 2 (starting at location zero) |
| `s.rfind('o')` # What position is the last ‘o’ at? | 7 (starting at location zero) |
| `s.rfind('l')` # What position is the last ‘l’ at? | 9 (starting at location zero) |

**String Manipulation**

Other built-in methods include the `split` method to split a string based on a specificed parameter, the `join` method join substrings based on a specificed parameter, the `replace` method replaces characters in a string with others, the `partition` method divides the string into three parts based on the first occurance of a specificed parameter, and finally the `rpartition` method splits on the last occurance of a parameter.

| `s = "Hello World, how are you?"` | |
|---|---|
| `s2 = s.split(' ')` | ['Hello', 'World,', 'how', 'are', 'you?'] |
| `s3 = '#'.join(s2)` | Hello#World,#how#are#you? |
| `s4 = s.replace(' ', '**')` | Hello**World,**how**are**you? |
| `s5 = s.partition(' ')` | ('Hello', ' ', 'World, how are you?') |
| `s6 = s.rpartition(' ')` | ('Hello World, how are', ' ', 'you?') |

Python has many other built-in methods, including `center()`, `endswith()`, `isalpha()`, `isdigit()`, `isspace()`, `lower()`, and `upper()`.

Object Oriented Programming © Damian Gordon

# String Formatting

**The `format` Method**
Another built-in method that Python 3 provides is the `format` method. This allows you to add text into a string using curly braces. There are a number of different ways of persenting the arguments to the string.

**Unindexed Arguments**
If you put empty curly braces into the string with and have an equivalent number of strings in the `format` command, Python will use positional substitution to repleace the first set of braces with the first string in the `format` command, the second braces with with the second string in the `format` command, etc.

```
MyText = "{}, you are currently {}"
print(MyText.format('Damian', 'teaching'))
```

Gives you the following output:

```
Damian, you are currently teaching
```

**Indexed Arguments**
If you put numbers in the curly braces and have a number of strings in the `format` command, the Python intrepreter will substitute the arguments on the basis of the numbers (starting at zero).

```
MyText = "{0}, you are currently {1}, thanks {0}"
print(MyText.format('Damian', 'teaching'))
```

Gives you the following output:

```
Damian, you are currently teaching, thanks Damian
```

**Keywords Arguments**
If you put labels in the curly braces and have a number of strings, each associated with one of the labels, in the `format` command, the Python intrepreter will substitute the arguments on the basis of the labels.

```
MyText = "{name}, you are currently {activity}"
print(MyText.format(name=”Damian”, activity=”teaching”))
```

Gives you the following output:

```
Damian, you are currently teaching
```

**Mixing Argument Types**
You are also allowed to mix together different types of arguments, so in the example below the two unindex arguments are mixed with a keyword arguement. The two unindexed arguments match with the strings “x” and “x+1” in the `format` command which are also unlabelled. And the argument labelled as “Label” matches the string “=”:

```
print("{} {Label} {}".format("x", "x + 1", Label = "="))
```

Gives you the following output:

```
x = x + 1
```

This is a sampling of the range of ways you can use string arguments.

Object Oriented Programming © Damian Gordon

# Regular Expressions

**Regular Expressions**

A regular expression is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or *string matching*. Regular expressions originated in 1956, when mathematician Stephen Cole Kleene described regular languages using his mathematical notation called *regular sets*. Python has a library called `re` to help:

```
# PROGRAM MatchingPatterns:
import re
SearchString = "hello world"
pattern      = "hello world"
IsMatch = re.match(pattern, SearchString)
if IsMatch == True:
    print("regex matches")
# ENDIF;
# END.
```

This program compares `SearchString` to `pattern`, and if it matchs, it prints out the phrase "`regex matches`".

**Basic Patterns**

**Logical OR**: A vertical bar separates alternatives. For example, gray|grey can match "gray" or "grey".

**Grouping**: Parentheses are used to define the scope and precedence of the operators. For example, gr(a|e)y

**?**: indicates zero or one occurrences of the preceding element. For example, colou?r matches both "color" and "colour".

*****: indicates zero or more occurrences of the preceding element. For example, ab*c matches "ac", "abc", "abbc", "abbbc", and so on.

**+**: indicates one or more occurrences of the preceding element. For example, ab+c matches "abc", "abbc", "abbbc", and so on, but not "ac".

**{n}**: The preceding item is matched exactly n times.

**{min,}**: The preceding item is matched min or more times.

**{min,max}**: The preceding item is matched at least min times, but not more than max times.

**Basic Pattern Matching**

'hello world' matches 'hello world'
'hello world' matches 'hello worl'
'hello world' does not matche 'ello world'

**Matching Single Characters**

'hello world' matches 'hel.o world'
'helpo world' matches 'hel.o world'
'hel o world' matches 'hel.o world'
'helo world' does not match 'hel.o world'

'hello world' matches 'hel[lp]o world'
'helpo world' matches 'hel[lp]o world'
'helPo world' does not match 'hel[lp]o world'

'hello world' does not match 'hello [a-z] world'
'hello b world' matches 'hello [a-z] world'
'hello B world' matches 'hello [a-zA-Z] world'
'hello 2 world' matches 'hello [a-zA-Z0-9] world'

**Special Characters**

'.' matches pattern '\.'
'[' matches pattern '\['
']' matches pattern '\]'
'(' matches pattern '\('
')' matches pattern '\)'

**Example Matches**

'(abc]' matches '\(abc\]'
' 1a' matches '\s\d\w'
'\t5n' does not match '\s\d\w'
' 5n' matches '\s\d\w'

# Regular Expressions

### Matching Multiple Characters

The asterisk (*) character says that the previous character can be matched zero or more times.

```
'hello' matches 'hel*o'
'heo' matches 'hel*o'
'hellllo' matches 'hel*o'
```

The pattern `[a-z]*` matches any collection of lowercase words, including the empty string:

```
'A string.' matches '[A-Z][a-z]* [a-z]*\.'
'No .' matches '[A-Z][a-z]* [a-z]*\.'
'' matches '[a-z]*.*'
```

The plus (+) sign in a pattern behaves similarly to an asterisk; it states that the previous character can be repeated one or more times, but, unlike the asterisk is not optional:

```
'0.4' matches '\d+\.\d+'
'1.002' matches '\d+\.\d+'
'1.' does not match '\d+\.\d+'
```

The question mark (?) ensures a character shows up exactly zero or one times, but not more.

```
'1%' matches '\d?\d%'
'99%' matches '\d?\d%'
'999%' does not match '\d?\d%'
```

If we want to check for a repeating sequence of characters, by enclosing any set of characters in parenthesis, we can treat them as a single pattern:

```
'abccc' matches 'abc{3}'
'abccc' does not match '(abc){3}'
'abcabcabc' matches '(abc){3}'
```

### Two Further Patterns

**^**: The start of a string.
**$**: The end of a string.

### More Complex Patterns

Combining the patterns together allows us to expand our pattern-matching repertoire:

```
'Eat.' matches
'[A-Z][a-z]*( [a-z]+)*\.$'
'Eat more good food.' matches
'[A-Z][a-z]*( [a-z]+)*\.$'
'A good meal.' matches
'[A-Z][a-z]*( [a-z]+)*\.$'
```

### RegEx for a Valid e-mail Format

The regular expression that can be used to represent a valid e-mail is as follows:

```
pattern = "^[a-zA-Z.]+@([a-z.]*\.[a-z]+)$"
```

### More `re` Methods

In addition to the match function, the re module provides a couple other useful functions, `search()`, and `findall()`.

- The `search()` function finds the first instance of a matching pattern, relaxing the restriction that the pattern start at the first letter.
- The `findall()` function behaves similarly to `search`, except that it finds all non-overlapping instances of the matching pattern, not just the first one.

```
>>> import re

>>> re.findall('a.', 'abacadefagah')
['ab', 'ac', 'ad', 'ag', 'ah']

>>> re.findall('a(.)', 'abacadefagah')
['b', 'c', 'd', 'g', 'h']

>>> re.findall('(a)(.)', 'abacadefagah')
[('a', 'b'), ('a', 'c'), ('a', 'd'), ('a', 'g'), ('a', 'h')]

>>> re.findall('((a)(.))', 'abacadefagah')
[('ab', 'a', 'b'), ('ac', 'a', 'c'), ('ad', 'a', 'd'),
('ag', 'a', 'g'), ('ah', 'a', 'h')]
```

Object Oriented Programming © Damian Gordon

# Object Serialization

**Serializing and Deserializing (Pickling andUnpickling)**
We can store an object into a file, and retrieving it later from storage. Storing an object is called *serializing* it, and retrieving it is called *deserializing* it. Python uses a function called `pickle` to do this. So sometimes instead of saying we are serializing an object, we can say we are *pickling an object*; and instead of deserializing an object, we can say we are *unpickling an object*.

**The Dump Method**
To save an object to a byte file, `pickle` provides a `dump` method:

```
pickle.dump(object, file)
```

For example:

```
open("pickled_list.p", "wb") as MyFile:
pickle.dump(MyObject, MyFile)
```

So we open a byte file ("pickled_list.p") for writing, as MyFile, and serialize that object into MyFile.

**The Load Method**
To take a byte file and load it into an object, we have the `load` method:

```
object = pickle.load(file)
```

For example:

```
open("pickled_list.p", "rb") as MyFile:
MyNewObject = pickle.load(MyFile)
```

So open the byte file ("pickled_list.p") for reading, as MyFile, and load it into the object.

**Example Program**

```
import pickle
MyObject = ["a list", "containing", 5, "values including
another list", ["inner", "list"]]

with open("pickled_list.p", "wb") as MyFile:
    pickle.dump(MyObject, MyFile)
with open("pickled_list.p", "rb") as MyFile:
    MyNewObject = pickle.load(MyFile)
```

Using the `with` statement means that the file is automatically closed when finished.

**The Dumps and Loads Methods**
The `dumps` and `loads` behave much like their file-like counterparts, except they return or accept byte strings instead of file-like objects. The `dumps` method requires only one argument, the object to be stored, and it returns a serialized byte string object. The `loads` method requires a byte string object and returns the restored object.

```
import pickle
MyString = ["a list", "containing", 5, "values"]
DumpedString = pickle.dumps(MyString)
LoadedString = pickle.loads(DumpedString)
```

The variable `LoadedString` is uploaded as a byte string.

# Object Serialization

**Serializing Web Objects**

To transmit object data over the web you need to use a recognised standard so that the sending and receiving classes will know what is being transmitted. There are many common standards, but the most common one is JSON ("jason"). JSON stands for JavaScript Object Notation, and is an open-standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. JSON is a language-independent data format, and the JSON filename extension is .json:

```
{
   "firstName": "John",
   "lastName": "Smith",
   "age": 25,
   "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York"
    }
}
```

**The `Import` Statement**

To deal with JSON web objects, Python provides a json library, so we say:

```
import json
```

**The `Dump` and `Load` Methods**

The `Dump` and `Load` methods work almost exactly the same way as their pickle counterparts, except that we are creating text files with valid JSON notation rather than byte files. So to save an object to a JSON file, `json` provides a `dump` method:

```
json.dump(object, file)
```

For example:

```
   open("MyFile.json", "w") as WriteFile:
   json.dump(MyObject, WriteFile)
```

So we open a text file ("MyFile.json") for writing, as WriteFile, and serialize that object into MyFile. And to take a text file and load it into a JSON object, we `load`:

```
object = json.load(file)
```

For example:

```
open("MyFile.json ", "r") as WriteFile:
MyNewObject = json.load(WriteFile)
```

So open the text file ("MyFile.json") for reading, as MyFile, and load it into the object.

**The `Dumps` and `Loads` Methods**

The `dumps` and `loads` behave much like their `pickle` counterparts:

```
import json
MyString = ["a list", "containing", 5, "values"]
DumpedString = json.dumps(MyString)
LoadedString = json.loads(DumpedString)
```

The variable `LoadedString` is uploaded as a byte string.

# Design Patterns

**What are Design Patterns?**
Computer scientists got together and discussed the common types of problems they are usually asked to solve, and realised that a lot ofcreated a set of generic solutions to those problem. These are not full programs, and sometimes not even pseudocode, but represent good ideas or good approaches to solving common problems. Types of Design Patterns include: **Algorithm strategy patterns**, **Computational design patterns**, **Execution patterns**, **Implementation strategy patterns**, and **Structural design patterns**

**Some Common Design Patterns**

**The Iterator Pattern:** A design pattern in which an iterator is used to traverse a container and access the container's elements.

**The Decorator Pattern:** A design pattern which wraps an existing class and can alter the functionality of the methods.

**The Observer Pattern:** A design pattern which monitors a core class and different observers react different to changes in the core class.

**The Strategy Pattern:** A design pattern which presents different potential solutions to the same problem, and allows the program to choose the most suitable one.

**The State Pattern:** A design pattern which represents a system that goes through different states, and records the current state and the transitions between states.

**The Singleton Pattern:** A design pattern which allows only one object based on a certain class to exist.

**The Template Pattern:** A design pattern which creates a common base class the can be inherited by multiple class that share common states, these can override the base class methods.

**The Adapter Pattern:** A design pattern which allows two pre-existing objects to interact with each other, even if their interfaces are not compatible.

**The Façade Pattern:** A design pattern which presents a simple interface to complex system but encapsulating typical usage into a new object.

**The Flyweight Pattern:** A design pattern which helps objects that share the same state to use the same memory location.

**The Command Pattern:** A design pattern which creates an object (an execute object) that can execute another object at a later time.

**The Abstract Factory Pattern:** A design pattern which returns a different class (or implementation) of the same system, depending on the platform, local settings, or current locale.

**The Composite Pattern:** A design pattern which allows complex tree-like structures to be built easily from simple components.

These are some of the 23 classic design patterns.

Object Oriented Programming © Damian Gordon

# Design Patterns

**The Singleton Pattern**

The Singleton Pattern is a design pattern which allows only one object based on a certain class to exist. The general Singleton design pattern is as follows:

```
class Singleton:
  IsSingleton = None

  Method NewObject:
    If IsSingleton == None
      Then IsSingleton = super(Singleton).new_object
    EndIf;
    return IsSingleton
  # END NewObject

# END Singleton.
```

In Python the Singleton design pattern can be implented as follows:

```
class OneOnly:
    _singleton = None

    def __new__(cls, *args, **kwargs):
        if not cls._singleton:
        # THEN
            cls._singleton = super(OneOnly, cls
                ).__new__(cls, *args, **kwargs)
        # ENDIF;
        return cls._singleton
    # END __new__

# END OneOnly.
```

In Python we use the _ _new_ _ class to help ensure there's only one instance of a class. When the _ _new_ _ class is called, it typically creates a new instance of that class. When we override it, we first check if our singleton instance has been created; if not, we create it using a super() call.

A sample output would be as follows:

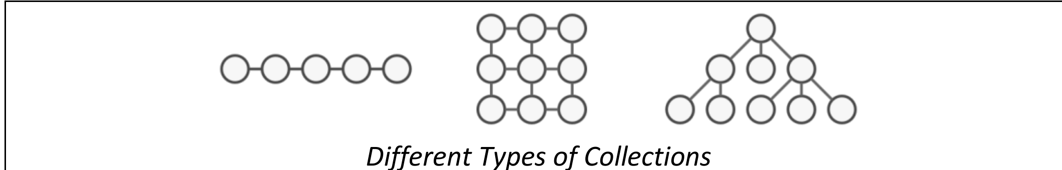
```
>>> o1 = OneOnly()
>>> o2 = OneOnly()
>>> o1 == o2
True
>>> o1
<__main__.OneOnly object at 0xb71c008c>
>>> o2
<__main__.OneOnly object at 0xb71c008c>
```

So even though it looks like two objects are created, in reality they are both at the same address in memory.

# Design Patterns

**The Iterator Pattern**

The iterator pattern is a design pattern in which an iterator is used to traverse a container and access the container's elements. Contaier types include lists, tuples, dictionaries, and sets, and they can be structured in multiple ways:



*Different Types of Collections*

The general Iterator design pattern provides a way to access the elements of a collection object sequentially without exposing its underlying representation. The Iterator design pattern allows us to separate out all the logic for iterating over a collection. It allows an object to traverse through a container (collection of objects) without having the container to reveal how the data is structured internally. To achieve this the iterator pattern is designed so that the container object provides a public interface in the form of an interator object for different client objects to access its contents. It consists of two main classes:

- Iterable is a class that provides a way to expose its data to the public.
- Iterator is a class that contains a pointer to the next element in the iteration.

**Generic Iterator Pattern**

```
class ITERABLE:
        def __init__(self, VALUE):
                self.VALUE = VALUE
        # END Init

        def __iter__(self):
                return ITERATOR(self.VALUE)
        # END Iter
# END ITERABLE.
```

```
class ITERATOR:
        def __init__(self, VALUE):
                        self.VALUE = VALUE
                        self.index = 0
        # END Init
        def __iter__(self):
                        return self
        # END Iter
        def __next__(self):
                if CONDITION:
                        VALUE = SELF.VALUE
                        self.index = self.index + 1
                        return VALUE
                else:
                        raise StopIteration()
                # ENDIF;
        # END Next
# END ITERATOR.
```

This is the general design pattern, not the specific implementation.

# Design Patterns

**Python Iterator Pattern**

To implement the Iterator Pattern, Python provides you with a pair of built-ins:

- `iter()` takes in a container object and builds and returns a new iterator object.
- `next()` takes in the iterator and, each time it is called, returns the next item from the container.When there are no more objects to return, the exception `StopIteration` is raised.

```
class MyCountIterable:
    def __init__(self, Value):
        self.Value = Value
    # END Init

    def __iter__(self):
        return MyCountIteration(self.Value)
    # END Iter
# END MyCountIterable.
```

```
class MyCountIteration:
    def __init__(self, Value):
        self.Index = 0
        self.Value = Value
    # END Init

    def __iter__(self):
    # Iterators are iterables too.
        return self
    # END Iter

    def __next__(self):
       if self.Index < self.Value:
            # THEN
            Index = self.Index
            self.Index += 1
            return Index
       else:
            raise StopIteration()
       # ENDIF;
    # END Next
# END MyCountIteration.
```

Here is the Python code to run the iterator program:

```
FirstCount = MyCountIterable(5)
list(FirstCount)
FirstCountIter = iter(FirstCount)
while True:
    try:
        print(next(FirstCountIter))
    except StopIteration:
          break
# ENDWHILE
```

This is clearly not as easy as a simple FOR loop counting 0 to 4, but it is a standard, and well-known pattern, and is readily recognisable by other programmers.

# Comprehensions and Generators

**COMPREHENSIONS**

- Comprehensions are a quick way of generating or altering Lists, Sets or Dictionaries. They provide a powerful range of functionality using a single line of code.

```
List        [List of Values]
Set         {Unique Values}
Dictionary  {Label: Set}
```

**List Comprehensions**

Let's imagine we had a list (or array) of strings, as follows:

- `string_array = ["234", "75", "331", "73", "5"]`

If we wanted to take each element and covert them into integers, we could do:

```
Output1 = [int(num) for num in string_array]
```

Convert `num` to integer — for each number in the list

Which would give us:

```
[234, 75, 331, 73, 5]
```

If we wanted to only convert strings less than three characters long:

```
output2 = [int(num) for num in string_array  if len(num) < 3]
```

Convert `num` to integer — for each number in the list — if string length less than 3

Which would give us:

```
[75, 73, 5]
```

**Set Comprehensions**

A set is like a list but with no duplicate entries. We create a set is to use the `set()` constructor to convert a list into a set, but we can also use a set comprehension.

```
f-authors = {b.author for b in books if b.genre == 'fantasy'}
```

Add unique author — for each book in list — if genre attribute is "fantasy"

Which would give us the following output (removing any duplicate values):

```
{'Pratchett', 'Le Guin', 'Turner'}
```

**Dictionary Comprehensions**

A dictionary is a list that has a label at the start of it, we can use an existing dataset and convert it into a dictionary using a dictionary comprehension.:

```
f-titles = {b.title: b for b in books if b.genre == 'fantasy'}
```

Add label — set — for each book in list — if genre attribute is "fantasy"

Now we have a dictionary we can look up using title.

Object Oriented Programming © Damian Gordon

# Comprehensions and Generators

**GENERATORS**

- Generators have a similar synthax to Comprehensions, but work on Tuples instead of Lists, Sets and Dictionaries.

*Generator* **(List of Values)**

**Generator Experssions**

If we were processing a large log file that had lots of lines in it, some of which had the word "WARNING" in it. If the log file was very big (terabytes) and we were looking only for lines with the word “WARNING” in them, we shouldn’t use a List Comprehension in this case because they would temporarily create a list containing every line and then search for the appropriate messages.

Instead if we use a generator, we avoid that issue because generators don’t create a temporary list, they only write content out when they are instructed to, this is sometimes called *Lazy Evaluation*, and it allows you to start using the list immediately:

```
import sys
InName = "InputFile.txt"
OutName = "OutputFile.txt"

with open(InName) as infile:
  with open(OutName, "w") as outfile:
    warnings = (line for line in infile if 'WARNING' in line)
    for line in warnings:
        outfile.write(line)
    # ENDFOR;
# END.
```

So the most important line in the program is as follows:

```
warnings = (line for line in infile if 'WARNING' in line)
```

Add this line in | for each line in the file | if ‘WARNING’ is in line

If we wanted to remove the word “WARNING” out of each line, we could do:

```
W = (line.replace(' WARNING', '') for line in file if 'WARNING' in line)
```

Replace ‘WARNING’ with blank | for each line in the file | if ‘WARNING’ is in line

If we wanted to do the same thing in a more object-oriented manner, we could use the following code. Note that the `yield` command works exactly like the normal `return` command (when you return a value from a method), but it temporarily returns control to the calling method, and remembers where it was in a sequence in each new call.

```
def warnings_filter(insequence):
    for line in insequence:
        if 'WARNING' in line:
            yield line.replace(' WARNING', '')
```

This will return the same result as the previous code snippet.

Object Oriented Programming © Damian Gordon

# Unit Testing

**Python `UnitTest` Library**

Unit Testing is concerned with testing small chunks of a program, for example, testing a single class or a single method. Python has a library for unit testing called `unittest`. It provides several tools for creating and running unit tests.

One of the most important classes in `unittest` is called `TestCase` which provides a set of methods to compare values, set up tests and clean up after tests are finished. To write unit tests, we create a subclass of `TestCase` and write methods to do the actual testing. Typically we start all of these methods with the name test. Here's an example:

```
import unittest
class CheckNumbers(unittest.TestCase):
    def test_int_float(self):
        self.assertEqual(1, 1.0)
    # END test_int_float
# END CheckNumbers
```

If the assertion is found to be true, it returns ".", and if it fails, it returns "F".

**Assetion Methods**

| Methods | Description |
|---|---|
| `assertEqual`<br>`assertNotEqual` | Accept two comparable objects and ensure the named equality holds. |
| `assertTrue`<br>`assertFalse` | Accept a single expression, and fail if the expression does not pass an IF test. |
| `assertGreater`<br>`assertGreaterEqual`<br>`assertLess`<br>`assertLessEqual` | Accept two comparable objects and ensure the named inequality holds. |
| `asssertIn`<br>`assertNotIn` | Ensure an element is (or is not) an element in a container object. |
| `assertIsNone`<br>`assertIsNotNone` | Ensure an element is (or is not) the exact value None (but not another false value). |
| `assertSameElements` | Ensure two container objects have the same elements, ignoring the order. |
| `assertSequenceEqual`<br>`assertDictEqual`<br>`assertSetEqual`<br>`assertListEqual`<br>`assertTupleEqual` | Ensure two containers have the same elements in the same order. If there's a failure, show a code diff comparing the two lists to see where they differ. The last four methods also test the type of the list. |
| `assertRaises` | Ensure a specific function call raises a specific exception. |

To pass the `assertFalse` method the test should return False, None, 0, or an empty list, dictionary, string, set, or tuple. To pass the `assertTrue` method the test should return True, non-zero numbers, containers with values in.

# Unit Testing

**The `assertRaises` Method**

Let's look at an example of the `assertRaises` method:

```
import unittest

def MyAverage(seq):
  return sum(seq) / len(seq)
# END average

class TestAverage(unittest.TestCase):
    def setUp(self):
        self.stats = StatsList([1,2,2,3,3,4])
    # END setUp

    def test_mean(self):
        self.assertEqual(self.stats.mean(), 2.5)
    # END test_mean

    def test_MyAverage(self):
      self.assertRaises(ZeroDivisionError, MyAverage, [])
    # END test_zero
# END CLASS TestAverage
```

The `setUp` is called individually before each test, so each test starts with a clean slate.

**Built-in Exceptions**

Some of the Python built-in exceptions are as follows:

| Methods | Description |
|---|---|
| `AssertionError` | Raised when an `assert` statement fails. |
| `AttributeError` | Raised when an attribute reference or assignment fails. |
| `FloatingPointError` | Raised when a floating point operation fails. |
| `IOError` | Raised when an I/O operation (such as a `print` statement) fails for an I/O-related reason. |
| `IndexError` | Raised when a sequence subscript is out of range. |
| `MemoryError` | Raised when an operation runs out of memory but the situation may still be rescued by deleting some objects. |
| `OverflowError` | Raised when the result of an arithmetic operation is too large to be represented. |
| `RuntimeError` | Raised when an error is detected that doesn't fall in any of the other categories. |
| `SyntaxError` | Raised when the parser encounters a syntax error. |
| `ZeroDivisionError` | Raised when the second argument of a division or modulo operation is zero. |

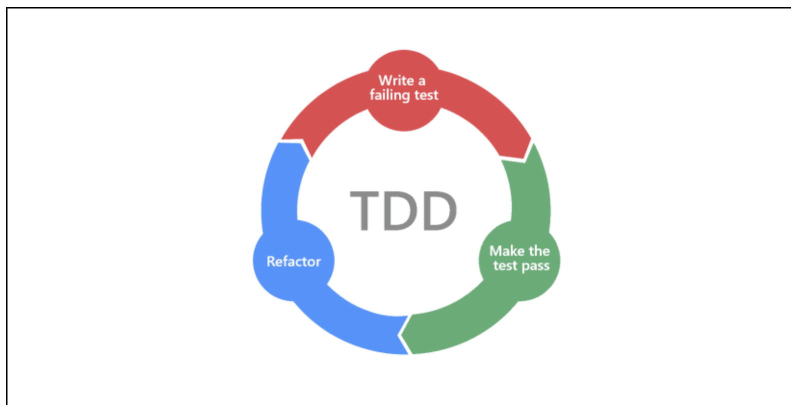
Test cases should never have side effects.

# Development Models

**Test-Driven Development**

"*Write tests first*" is the key mantra of Test-Driven Development. The key concept is that a developer shouldn't write any code until they have written tests for that code first. Test-Driven Development has two goals:

1. To ensure that tests are actually written, and written well. Too often developers leave the design of tests until after the development process, and then don't bother because the code seems to work.
2. To help the developers envisage exactly what the code will do, and what processes and modules it will interact with, thus testing becomes part of the design process.




**Test-Driven Development Cycle**

A typical test-driven development cycle is as follows:

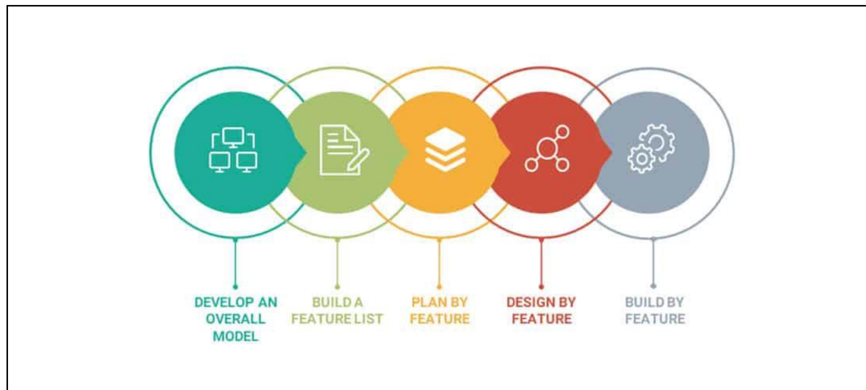
1. *Add a test*: Each new feature begins with writing a test.
2. *Run all tests and see if the new test fails*: This validates that the test harness is working correctly.
3. *Write the code*: The next step is to write some code that causes the test to pass.
4. *Run tests*: If all test cases now pass, the programmer can be confident that the new code meets the test requirements.
5. *Refactor code*: The growing code base must be cleaned up regularly during test-driven development.
6. *Repeat*: Starting with another new test, the cycle is then repeated to push forward the functionality.

Test-driven development offers more than just validation of correctness, it can also drive the design of a program. By focusing on the test cases first, we have to imagine how the functionality is used by end-users (in the first case, the test cases). So, we are concerned with the interface before the implementation.

Object Oriented Programming © Damian Gordon

# Development Models

**Feature Driven Development**
Feature Driven Development (FDD) is a framework that organizes the software development process around developing one feature at a time until a complete system is finished. It was originally developed for a large team, and as such is designed to compensate for the range of skills that could be found in a large team.




**Feature-Driven Development Cycle**
A typical feature-driven development cycle is as follows:

1. *Build a domain object model*, in an intense, highly iterative, collaborative and generally enjoyable activity involving "*domain and development members under the guidance of an experienced object modeller in the role of Chief Architect*". While not mandatory, the object model is typically built using Peter Coad's modelling in colour technique.
2. *Build feature list*, features are small, client-valued requirements. Feature typically take 1-3 days to implement, occasionally 5 days but never 10 or more days to implement.
3. *Plan by Feature*, the last initial phase involves constructing an initial schedule and assigning initial responsibilities. The development team sequence the feature sets based on activities that represent best relative business value.
4. *Design by Feature*, Each feature is tackled by a feature team (3-5 designers and developers usually working together for 1-3 days).
5. *Build by Feature*, This involves the team members coding up the features, testing them at both unit level and feature level, and holding a code inspection before promoting the completed features into the project's regular build process.

FDD mandates code inspections is that research has shown time and again that when done well, inspections find more defects and different kinds of defects than testing. Not only that but by examining the code of the more experienced, knowledgeable developers on the team and having them explain the idioms they use, less experienced developers learn better coding techniques.

## Python Object-Oriented Programming Quiz

1. *What is a method, and how do you declare it in Python?*
2. *What is an attribute, and how do you declare it in Python?*
3. *How do you declare a class, and how do you instantiate a class?*
4. *What is the purpose of the Initialization Method?*
5. *What are Docstrings?*
6. *What is the purpose of the __init__ method?*
7. *What are Docstrings?*
8. *What is a module and what is a package?*
9. *How does Access Controls work in Python?*
10. *Explain the purpose of the Python Package Index.*
11. *What is Inheritance, and how is it implemented in Python?*
12. *What is overriding?*
13. *What is super?*
14. *What is Multiple Inheritance, and how is it implemented in Python?*
15. *What is Polymorphism?*
16. *What is the purpose of Getters and Setters?*
17. *What are Manager Objects?*

# Python Object-Oriented Programming Quiz

1. *What do the following String functions do?*
   - *`count()`*
   - *`find()`*
   - *`rfind()`*

2. *What do the following String Manipulation functions do?*
   - *`split()`*
   - *`join()`*
   - *`replace()`*
   - *`partition()`*
   - *`rpartition()`*

3. *How are the following argument types used in the string Format method*
   - *Unindexed arguments*
   - *Indexed arguments*
   - *Keyword arguments*

4. *What are regular expressions?*

5. *In Python, how is an object serialized (pickled) and deserialized (unpickled)?*

6. *In Python, how is a Web object serialized (pickled) and deserialized (unpickled)?*

7. *What is Unit Testing?*

8. *In Python, what is the purpose of the TestCase class?*

9. *What is the purpose of the assertEqual() method?*

10. *Give 5 examples of assert methods.*

11. *Explain the purpose of the assertRaises() method?*

12. *Give 5 examples of built-in exceptions.*

13. *Explain the goals of Test-Driven Development, and outline the TDD Cycle.*

14. *Explain the goals of Feature-Driven Development, and outline the FDD Cycle.*

# Object-Oriented Programming In Java Workbook

**Damian Gordon**

**2020**

Feel free to use any of the content in the guide with my permission.

Any suggestions or comments are most welcome, email me Damian.X.Gordon@TUDublin

# Table of Contents

*1. Introduction to Java*

*2. Handling Data in Java*

*3. Object-Oriented Java*

# Introduction to Java

**Java Introduction**
Java was developed by James Gosling, Mike Sheridan, and Patrick Naughton, starting in 1991, with Java 1.0 released in 1996. There have been 15 version of Java up to 2020.

Java was designed as a general-purpose programming language intended to let developers *write once, run anywhere* (WORA) meaning that compiled Java code can run on all platforms without the need for recompilation

**The Java "Hello, World!" Program**
Here's how to we say hellow world, we have the `print` statement inside a method called `main`, and that method is inside a `class`:

```
public class HelloWorld {

   public static void main(String []args) {

      System.out.print("Hello World\n");
   }
}
```

Here's what each of those statements mean:

| The Class | The Method | The Print |
|---|---|---|
| `public`: Public is an access modifier for classes and methods, and means they are accessible by any other class.<br><br>`class`: Used to create a class.<br><br>`HelloWorld`: This can be whatever name you want (except for keywords and built-in function names. | `public`: Public is an access modifier for classes and methods, and means they are accessible by any other class.<br><br>`static`: There won't be an object created from the class that this method is in.<br><br>`void`: means that this method doesn't return anything.<br><br>`main`: This is the first method Java will visit, the main method.<br><br>`String []args`: Any command line arguments are put into the argument-string, like parameters that go into the program. | `System`: A class that contains several useful Input/Output attributes and methods. It cannot be instantiated.<br><br>`out`: An output class that helps write content to the screen.<br><br>`print`: Prints the string enclosed in double quotes. |

The class and the method will be the same for most Java programs we are going to write, so you can cut-and-paste it when you are writing new code (just change the classname).

# Introduction to Java

### Java Comments

```
// A Single Line Comment
/* This is a multi-line
Comment */
```

### Java Arithmetic Operators

| | |
|---|---|
| + | Addition, 7 + 3 |
| - | Subtraction, 7 - 3 |
| * | Multiplication, 7 * 3 |
| / | Integer Divsion, 7 / 3 |
| / | Real Divsion, 7.0 / 3.0 |
| % | Division Remainder, 7 % 3 |

### Java Variable Types

```
int x;
x = 15;
int x = 15;
```

```
double x;
x = 15.0;
double x = 15.0;
```

```
char x;
x = 's';
char x = 's';
```

```
String x;
x = "Hello, World!";
String x = "Hello World!";
```

```
boolean x;
x = false;
boolean x = false;
```

### Java Conditional Operators

| | |
|---|---|
| != | Is not equal to |
| == | Is equal to |
| > | Is greater than |
| < | Is less than |
| >= | Is greater than or equal to |
| <= | Is less than or equal to |

### Java Logical Operators

| | |
|---|---|
| && | Logical AND |
| \|\| | Logical OR |
| ! | Logical NOT |

These can be used in conditions.

### Java String Formatting

| | |
|---|---|
| `length()` | Length of string |
| `toUpperCase()` | Make upper case |
| `toLowerCase()` | Make lower case |
| `indexOf(Str)` | Find `Str` |

### Java IF Statement

```
if (condition) {
   // if condition is true
} else {
   // if condition is false
}
```

### Java SWITCH Statement

```
switch(expression) {
   case x:
      // code block
      break;
   case y:
      // code block
      break;
   default:
      // code block
}
```

### Java WHILE Statement

```
while (condition) {
   // code to be executed
}
```

### Java FOR Statement

```
for (initial; cond; inc) {
   // code to be executed
}
```

### Java Methods

```
public class Main {
   static void METHOD () {
        // code to be executed
   }
}
```

All methods are generally enclosed within a class, and the method called `main()` is executed first.

# Handling Data in Java

**Reading in Data**

To get input from the user, Java has the Scanner class, so we need to do the following:

```
import java.util.Scanner;     // import the Scanner class
```

And from there we can read the system input (`System.in`) by creating an object:

```
Scanner myObj = new Scanner(System.in);
String userName;
System.out.println("Enter username: ");
userName = myObj.nextLine();
```

There are other methods as well as `nextLine()` which reads in the next string.

**Reading Methods**

| Method | Description |
|---|---|
| `nextBoolean()` | Reads a boolean value from the user. |
| `nextByte()` | Reads a byte value from the user |
| `nextDouble()` | Reads a double value from the user |
| `nextFloat()` | Reads a float value from the user |
| `nextInt()` | Reads a integer value from the user |
| `nextLong()` | Reads a long value from the user |
| `nextShort()` | Reads a short value from the user |

**File Handling**

To control files we need to import the `Scanner` class, as well as the `File` class:

```
import java.io.File;          // Import the File class
```

And then to read in a file, we do the following:

```
File myObj = new File("filename.txt");
Scanner myReader = new Scanner(myObj);
while (myReader.hasNextLine()) {
    String data = myReader.nextLine();
    System.out.println(data);
}
myReader.close();
```

**File Methods**

| Method | Description |
|---|---|
| `getName()` | Returns the name of the file |
| `getAbsolutePath()` | Returns the absolute pathname to the file |
| `canWrite()` | Returns whether you can write to the file |
| `canRead()` | Returns whether you can read from the file |
| `length()` | Returns the length of the file |
| `createNewFile()` | Creates a new file. |

To write to a file, we import the `FileWriter` class, instead of thr the `File` class:

```
import java.io.FileWriter;      // Import FileWriter class
```

And we can use the method `write(String)` to add to the file.

# Handling Data in Java

**Arrays in Java**

To declare an array in Java, get can do the following:

```
int[] Age;
```

To initialise the array:

```
int[] Age = {44, 23, 42, 33, 16};
```

To access the first element in the array:

```
System.out.println(Age[0]);
```

A program to print out all of the values in an array can be as follows:

```
int[] Age = {44, 23, 42, 33, 16};
for (int i = 0; i < Age.length; i++) {
  System.out.println(Age[i]);
}
```

For a String array it's almost exactly the same:

```
String[] cars = {"Volvo", "BMW", "Ford"};
for (int i = 0; i < cars.length; i++) {
  System.out.println(cars[i]);
}
```

**Linked Lists in Java**

Java has a linked list class to help in creating and accessing linked lists:

```
import java.util.LinkedList;  // Import LinkedList class
```

And we can create an linked list object as follows:

```
LinkedList<String> cars = new LinkedList<String>();
cars.add("Volvo");
cars.add("BMW");
```

There are also several methods to get, set, add and remove items from the linked list:

| Method | Description |
|---|---|
| `get(index)` | Returns the item at location *index*. |
| `set(index, value)` | Sets the item at location *index* to the value *value*. |
| `remove(index)` | Removes the item at location *index*. |
| `addFirst(value)` | Add an item to the start of the list with the value *value*. |
| `addLast(value)` | Add an item to the end of the list with the value *value*. |
| `removeFirst()` | Removes the first item from the list. |
| `removeLast()` | Removes the last item from the list. |
| `getFirst()` | Returns the first item of the list. |
| `getLast()` | Returns the last item of the list. |
| `clear()` | Clears the list. |

Together this gives us a lot of functionality to manpulate linked lists.

Object Oriented Programming © Damian Gordon

# Object-Oriented Java

**Basic Java Object-Oriented Programming**

Declaring a class in Java is done as follows:

```
public class Main {
  int x = 5;
}
```

And to declare an object, we do:

```
public class ExampleObj {
  int x = 5;

public static void main(String[] args) {
    ExampleObj myObj = new ExampleObj();
    System.out.println(myObj.x);
  }
}
```

As you can see we've also creted an attribute in the above program.

To modify an attribute we do the following:

```
public class Main {
  int x = 10;

  public static void main(String[] args) {
    Main myObj = new Main();
    myObj.x = 40;
    System.out.println(myObj.x);
  }
}
```

If we change the declaration of "`int x = 10;`" to "`final int x = 10;`", the code to change X to 40 will produce an error.

**Java Constructors**

A constructor in Java is a special method that is used to initialize objects. The constructor name must match the class name, and it cannot have a return type (like `void`). Also note that the constructor is called when the object is created. All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you.

```
public class ExampleObj {
  int x;
  public ExampleObj(int y) {
    x = y;
  }

  public static void main(String[] args) {
    ExampleObj myObj = new ExampleObj(5);
    System.out.println(myObj.x);
  }
}
```

The constructor sets x to y, and in this case we pass a parameter to the constructor (5).

# Object-Oriented Java

**Java Interitance**

Let's look an example, the Car class (subclass) inherits the attributes and methods from the Vehicle class (superclass):

```
class Vehicle {
  protected String brand = "Ford"; // Vehicle attribute
  public void honk() {             // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}
```

```
class Car extends Vehicle {
  private String modelName = "Mustang";     // Car attrib
  public static void main(String[] args) {

    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method on the myCar object
    myCar.honk();

/* Display the value of the brand attribute (from Vehicle
class) and the value of the modelName from the Car class*/
    System.out.println(myCar.brand + myCar.modelName);
  }
}
```

To inherit from a class, use the extends keyword.

**Java Polymorphism**

For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}
```

```
class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}
```

```
class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}
```

Instances of `Dog` that call the method `animalSound()` will give a different anwswer than instances of `Pig` that call the method `animalSound()`.

Object Oriented Programming © Damian Gordon

## Java Object-Oriented Programming Quiz

1. *When declaring a java class what is the meaning of "public"?*
2. *When declaring a java method what is the meaning of "static"?*
3. *When declaring a java method what is the meaning of "void"?*
4. *When declaring the main method what is the meaning of "String []args"?*
5. *How are comments declared in java (both forms)?*
6. *List the 5 java variable types.*
7. *List the 6 conditional operators.*
8. *List the 3 logical operators.*
9. *List the 4 string formatting methods.*
10. *How is the IF statement declared in java?*
11. *How is the SWITCH statement declared in java?*
12. *How is the WHILE statement declared in java?*
13. *How is the FOR statement declared in java?*
14. *How are methods declared in java?*

# Java Object-Oriented Programming Quiz

1. *What is the purpose of the `Scanner` class?*
2. *What is the meaning of `System.in`?*
3. *What does the `nextLine()` method do?*
4. *What is the purpose of the `File` class?*
5. *What do the following methods do?*
   - `getName()`
   - `getAbsolutePath()`
   - `canWrite()`
   - `canRead()`
   - `length()`
   - `createNewFile()`
6. *What does the `FileWriter` class do?*
7. *How do you declare and initialise an array in Java?*
8. *Write a program to print out all of the values in an array.*
9. *What does the `LinkedList` class do?*
10. *How do you declare a Class in Java? How do you add attributes and methods?*
11. *How do you instantiate an object from a class in Java?*
12. *How do you write a Constructor in Java?*
13. *What is the purpose of the `extends` keyword?*
14. *Write a program that shows how polymorphism works in Java.*