YEAR 13

PROGRAMMING WITH OBJECTS AND GRAPHICAL USER INTERFACES

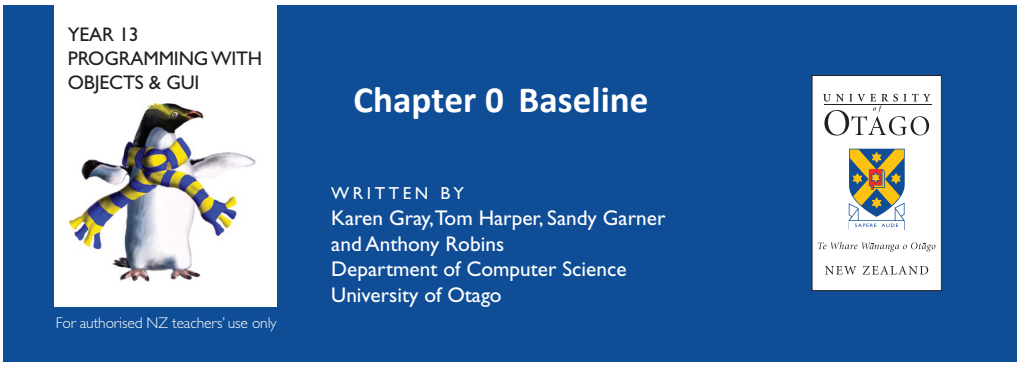
Student Workbook in Python

WRITTEN BY

Karen Gray, Tom Harper
Sandy Garner, Anthony Robins

Department of Computer Science
University of Otago

UNIVERSITY of OTAGO
SAPERE AUDE
Te Whare Wānanga o Otāgo
NEW ZEALAND



Assumed knowledge This workbook is designed to follow on from the Year 12 workbook by the same team in the same language. It is assumed that students using this book have completed a year 12 course in Python or a similar language. Transferring between languages should be possible with a little preparatory work. This chapter can be used as a refresher course.

Some terms which will be used with little further clarification:

assignment "gets the value of" the equals sign =

scope the portion of a program within which a variable can be used

local variable a variable declared within a function, scope limited to within that function

initialisation a variable is named for the first time and assigned an initial value

identifier the name given to a variable, function, method, class .

keyword word that has specific meaning in the language being used, unable to be used for other purposes

parameter value passed to a function

main routine the portion of your program from which execution begins

We assume familiarity with the use of Idle or some other IDE

Concepts which should already have been covered: lists, selection, iteration, use of `input`, using methods of the str class, functions including the use of return functions, formatting output.

A note about modules

When we discussed modularisation in year 12 we meant the "parcelling up" of certain tasks as functions. We stated, quite correctly, that "module" was a general (i.e. not Python specific) name for a function.

"Module" does have a specific meaning in Python. It means simply a Python file containing Python code. E.g.; a series of function definitions saved in a file called my_functions.py may be referred to as "the my_functions module". Such modules can be imported into a program so that its members are available for use in the program.

From now on we shall use "module" to refer to a Python file containing Python code. We will discuss import statements in a little more detail later in this workbook

0.1 Egg Order Example

We begin by presenting an example of a module (the Python sort!) which incorporates most of the key concepts listed as assumed knowledge. If you can understand how this program works, and answer the questions, you have the knowledge required for this workbook.

Look at the following code. Can you answer these questions about the eggOrder module?

1. Which are the functions which return a value?
2. Which function will be executed first?
3. If x is 2, what is the value of `x += 1` ?
4. Explain what is happening in the `get_dozens` function.
5. Explain the purpose of the `try..except` block in the `read_int` function.
6. Why is `choice` declared at the top of the `read_int` function rather than in the `try except` block where it is first used?
7. What condition is required to be true to stay in the `while` loop in the `get_orders` function?
8. How could you achieve the same purpose in the conditional statement without using `upper()`?
9. In which class is `upper()` defined?

Formatting note the following 3 statements will produce the same output at the end of the show_orders method:

```
print("{0:} ordered {1:} eggs. The price is ${2:.2f}".format(names[i], egg_order[i], price))
print("{0} ordered {1} eggs. The price is ${2:.2f}".format(names[i], egg_order[i], price))
print("{} ordered {} eggs. The price is ${:.2f}".format(names[i], egg_order[i], price))
```

They all format an output string by first defining a string with braces {} used in the spaces where variables will be inserted in the string. The braces might be numbered starting at 0 (like the first and second lines) or not numbered (third), but cannot be a mixture. The string is followed by **.format** and an ordered list of items to be formatted into the string to replace the braces.

If no formatting is required, an empty brace will insert the value. If a float value is to be formatted the brace will contain formatting instructions: the number of decimal places required are specified after a colon and a dot, and followed by 'f' e.g. {:.2f} or {0:.2f}

docstring note

In year 12, we just used the docstring for doctests. Generally, the first line of the docstring of any method or class should be a brief description of the purpose of the module. In year 13 we will use this pattern.

```
def get_orders(names, egg_order):
    """
    Collects order information - name, number of eggs – in a loop
    """

    print ("Collecting Orders")
    name = ""
    while name.upper() != 'F':
        name = input("What is the customer's name? (\"F\" to finish)")
        if name.upper()!= "F":
            names.append(name)
            egg_order.append(read_int("How many eggs does " + name + " wish to order?"))

def show_orders(names, egg_order):
    """
    calculates price for each egg order, and displays order information - name, number of eggs, price
"""
    """
    PRICE_PER_DOZEN = 6.5
    print("Showing orders")
    for i in range(len(names)):
        price = PRICE_PER_DOZEN /12 * egg_order[i]
        print("{} ordered {} eggs. The price is ${:.2f}".format(names[i], egg_order[i],
price))

def show_report(egg_order):
    """
    displays summary - total eggs, number of dozens to be ordered, average eggs per
customer
     """

    if len(names) == 0:
        print ("No orders")
    else:
        total = 0
```

```
        for eggs in egg_order:
            total += eggs
        print ("Summary")
        print ("Total eggs:" + str(total))
        print ("Dozens required:" + str(get_dozens(total)));
        average = 0
        if len(egg_order) > 0:
            average = total / len(egg_order)
        print ("Average number of eggs per customer: {0:.1f}".format(average))

def get_dozens (num_eggs):
    """
    returns whole number of dozens required to meet required number of eggs
    """
    num_dozens = num_eggs//12
    if num_eggs%12 != 0:
        num_dozens += 1

    return num_dozens

def read_int(prompt):

    choice = -1;
    while choice == -1:
        try:
            choice = int(input(prompt))
        except ValueError:
            print ("Not a valid integer")
    return choice

#main routine
names = []
egg_order = []
get_orders(names, egg_order)
show_orders(names, egg_order)
show_report(egg_order)
```

0.2 Warm up exercise

Here's an exercise to remind you about Python. You should be able to refer to the `EggOrder` example for assistance.

Design and write a program which takes care of a silent auction.

The beginning: It should have a reserve price, which is requested from the auction manager before the auction starts.

The auction: It should (repeatedly) show the highest bid so far, take a name and a bid, check that the bid is higher than the previous bid. If it is a higher bid, store the name and the bid at the next position in parallel lists. (Hopefully the last parallel list you will ever be asked to write) Otherwise show a "need a higher bid" message.

The program needs a way of being terminated by the user. Build in some prearranged signal value (e.g. a name of 'F' for FINISH) to end the loop on demand.

The end: The highest bid should be checked against the reserve, and an appropriate message produced. If the reserve price is met, the sale price and name of auction winner should be displayed. If not, a message to that effect should be displayed.

The program should then show the complete history of accepted bids with names of the bidder next to each. Some possible outputs are listed below.

What is the reserve price **30** Auction has started Highest bid so far is $0.00 What is your name? **Mr Able** What is your bid **25** Highest bid so far is $25.00 What is your name? **Mr Bear** What is your bid **35** Highest bid so far is $35.00 What is your name? **Mrs Clark** What is your bid **55.75** Highest bid so far is $55.75 What is your name? **F** Auction won by Mrs Clark with a bid of $55.75 Mr Able bid $25.00 Mr Bear bid $35.00 Mrs Clark bid $55.75	What is the reserve price **30** Auction has started Highest bid so far is $0.00 What is your name? **Mr Darby** What is your bid **15** Highest bid so far is $15.00 What is your name? **Mrs Jones** What is your bid **13** Sorry Mrs Jones You'll need to make another, higher, bid. Highest bid so far is $15.00 What is your name? **Mrs Jones** What is your bid **25** Highest bid so far is $25.00 What is your name? **f** Auction did not meet reserve price Mr Darby bid $15.00 Mrs Jones bid $25.00

0.3 More Revision from year 12

(a.k.a. Objects we already knew about even if we didn't know we did!)

From the introduction to Year 12, chapter 4

In Python the standard libraries contain useful classes, methods and other structures.

To explain what these are we need to define some key terms. In Python everything that can be given a name is an object. For example when we say `x = 3` we give the value 3 a name. Thus an integer is an object. So is a string e.g. `s = "Hello"`.

In Python every object is made from a class. So far we have talked about values (like 3) having a type (like int), but types and classes are the same thing. To say that 3 is of type `int` means that it is an object made from the class `int`.

A method is a function, which is part of a class. We call a method by specifying the value / object to call it on, followed by a ".", followed by the method name, for example:

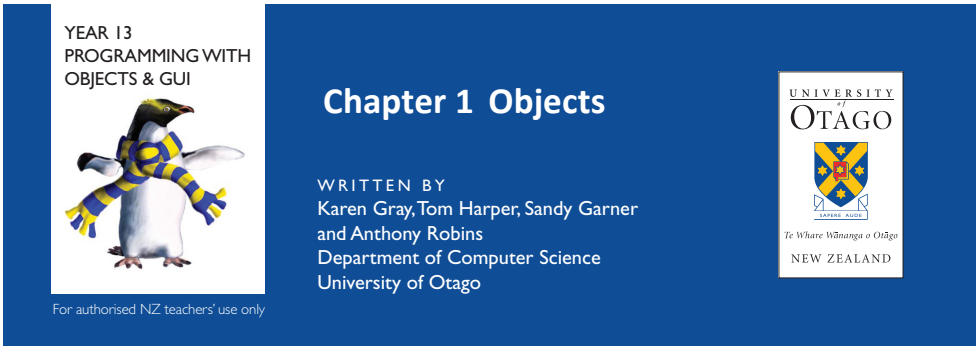
```
"My String".lower()
```

"My String" is one way of specifying an object made from the class str, so this statement is calling the `lower()` method on the "My String" object, and will return 'my string'. This makes a copy of the string in lower case.

Example:

```
n.__add__(4)
```

Given n = 6 (an object made from the class int) then the statement above will call the `__add__()` method on the object n, which will return 10. This is another way of doing addition. (The __ symbol is 2 underscore characters.)



1.1. Introduction to Objects

At the beginning of Year 12 we began by writing simple programs that just consisted of a series of statements, one after the other. These programs weren't very flexible or reusable. Then we moved to placing sections of code inside functions. Functions could be called repeatedly and passed different parameter values to make them more flexible. This made for a massive leap in the sophistication of our programming.

We are about to make another such leap and explore a different way of designing our programs – Object Oriented Programming (OOP).

Object Oriented Programs consist of a series of interacting objects. Objects are made from classes.

Remember the great realisation that we could define a function once and then call it many times. In a similar vein we can define a class and then from it create multiple objects (also known as instances or instance objects)

Classes can consist of multiple **methods** (which is what we call functions when they are in a class) as well as multiple **instance variables** (pieces of data). You can think of a class as a blueprint or a recipe from which many actual things (the instances) can be made.

Consider some examples. The first column in the table below describes a `Dog` class. It lists some properties a dog might have. It then describes some possible behaviours a dog might perform. The columns to the right look at a couple of actual `Dog` instances (`rover` and `fifi`).

Dog Class	Instance of Dog - rover	Instance of Dog - fifi
Properties		
name	Rover	Fifi
breed	German Shepherd	Poodle
age	3	6
Behaviours		
greet(a dog)	greet(a dog)	greet(a dog)
greet(a person)	greet(a person)	greet(a person)
flees(a dog)	flees(a dog)	flees(a dog)
wags tail(own)	wags tail(own)	wags tail(own)
chews slippers(a person's)	chews slippers(a person's)	chews slippers(a person's)

Notice that all the `Dog` instances have the same behaviours. Each `Dog` instance also has properties of the same name and type, but these have different values.

The table below describes a `Person` class.

Person Class	Instance of Person – susan	Instance of Person - john
Properties name gender age	 Susan Smith female 17	 John Jones male 45
Behaviours pats(a dog) flees(a dog) greets(a person)	 pats(a dog) flees(a dog) greets(a person)	 pats(a dog) flees(a dog) greets(a person)

Once again, each `Person` instance has the same behaviours, and the properties hold different values.

Even inanimate objects can have behaviours. Here is a table which describes a `DogKennel` class.

`DogKennel` Class	Instance of `DogKennel` - rover_kennel	Instance of `DogKennel` - fifi_kennel
Properties width length tenant	 1.2 1 rover	 .5 .3 fifi
Behaviours show_area() is_occupied(a dog) cleaning_routine()	 show_area() is_occupied(a dog) cleaning_routine()	 show_area() is_occupied(a dog) cleaning_routine()

Chapter 1 is going to show you how to apply these concepts when coding solutions to problems. If we created the instances shown from the classes above in Python, properties become **instance variables** and behaviours become **methods** (these are called the **attributes** of the class). **Glimpsing ahead**, we could then produce the following interactions between our objects:

```
rover.greets(john)
john.pats(rover)
rover.wags_tail()
susan.greets(john)
fifi.greets(rover)
rover.flees(fifi)
fifi.chews_slippers(john)
```

Another Glimpse ahead

A nursery Rhyme in code

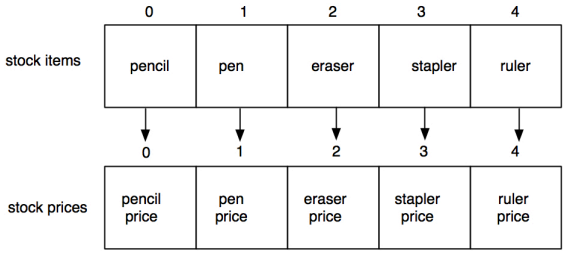
```
b1 = Bucket("water")
h1 = Hill("Mt Cook")
jack = Person("Jack", "male", 12)
jill = Person("Jill", "female", 11)
jack.climbs_hill(h1)
jill.climbs_hill(h1)
jack.fetch(b1)
jill.fetch(b1)
jack.fallsDown(h1)
jill.fallsDown(h1)
```

Why use Object Oriented Programming?

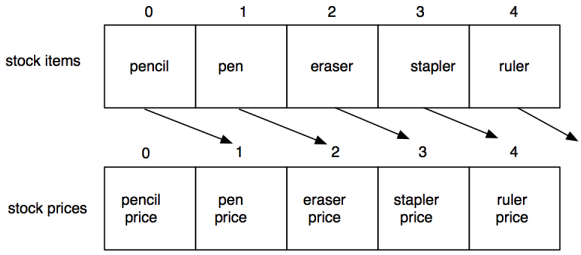
The underlying principle of object-oriented programming (OOP) is that data that belongs together should be stored together in an object. The object should have well-designed methods for manipulating the data. Because data can be protected from poorly considered changes, there is reduced likelihood of error when the program is running.

Up to now if we had to maintain relationships between pieces of data we had to employ parallel lists.

Imagine you had two parallel lists, one representing the name of a shop's stock items, and the other representing their prices.

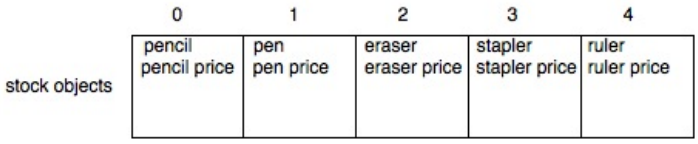


An accidental mismatch of the data, putting the wrong price on an item, is a very real possibility. Perhaps the programmer forgets to begin at 0 when processing the price list.



Or perhaps the data for the lists has been entered in a different sort order. Now imagine if we needed to create a third list to store, oh I don't know, shelf location. What about another to store cost prices? Matters can rapidly get out of hand!

If the price and name of a stock item are stored together in a stock item object, the two (or potentially more) related pieces of data cannot become detached from each other.



This was exactly the dilemma that faced us in our egg order module. There we painstakingly maintained a list of customer names and another parallel list of ints representing the number of eggs they each ordered. We did some slightly awkward checking but mostly we hoped and trusted that these lists would stay matched up and the same length! We will revisit this program towards the end of the chapter and see if we can shine a more Object Oriented light on it but first let's crack a few eggs and make a cake.

Developing an OOP mindset

Look at the picture below and pick a couple of likely objects. Brainstorm some possible data and behaviours for each one. How might objects be connected to each other?

1.2. Defining a class

Now we are hopefully ready to start defining our own classes. A class definition begins with a header:

```
class ClassName:
```

Over the course of this chapter we are going to develop examples of various Cake classes. So our header will be:

```
class Cake:
```

Note: By convention user-defined classes have names that start with a capital letter. Any further words in the class name also start with a capital letter with no underscores

The body of the class is then indented. The body of the class may consist of instance variables and methods. Methods are really just functions that are defined inside a class.

The __ init__ constructor method

In OOP we make objects which are instances of classes, when this happenes a special method in the class called a **constructor** runs to instantiate / initialise the new object. In Python the constructor for a class is the `__init__` method. It is called automatically whenever we instantiate (create an instance of) our class. It allows us to initialise our instance variables easily and sensibly. (Two underscore characters together __ make the underline before and after `init`, this indicates that we are dealing with a name that is a defined part of the language rather than a name invented by the programmer).

The `__init__` method's first parameter (`self`) is required and is a reference to the object being created. By convention this reference is **always** called `self`. This reference must prefix all variables that will belong to instances of the class. To do this we use dot notation. So inside any method of a class `self.variable_name` gives us access to that instance variable.

This reference to the instance object must also be the first parameter for any instance method we may wish to define. It is only by doing this that a method may be called on a particular instance. Ummm ok, let's just look at the example.

Code Listing 1.1 shows our `Cake` class with an `__init__` method, a `bake` method and a `print_ingredients` method.

```
class Cake:

    def __init__(self):
        """ Initialise instance variables """
        self.flour = 2
        self.sugar = 1
        self.bSoda = 1
        self.egg = 1
        self.butter = 200
        self.baking_time = 30

    def bake(self):
        """
        This instance method doesn't do much but could be
        developed to describe the behaviour of a cake being baked
        """
        print("baked")

    def print_ingredients(self):
        """  This instance method displays the cake ingredients  """
        print("*****************")
        print("Cake Ingredients")
        print("*****************")
        print()
        print(self.flour, "cups flour")
        print(self.sugar, "cups Sugar")
        print(self.bSoda, "tsp baking soda")
        print(self.egg, "eggs")
        print(self.butter, "grams butter")
```

Code Listing 1.1

Notice how the `print_ingredients` method is able to access `self.flour` even though it was declared inside `__init__`. It is the reference `self` that allows this to happen. Leaving `self` off would cause an error.

1.3. Creating instances of a class

Remember no building ever exists just because the plans have been drawn up. No instance object of Cake exists just because the Cake class has been defined.

Once we have a class definition, (our blueprint/recipe/template if you like) we can make instances from the class. Instances may be created in the main routine (i.e. below the class definition but not indented) or from inside another class or module. If the class definition is in another file then the file (i.e. Python module) containing the definition of the class must be imported into the file in which the instance is being created.

We can now sensibly create an instance of `Cake` in our `main` routine:

```
c1 = Cake()
```

The variable `c1` is the name of the instance, and provides the means to access it. The right hand side of the assignment `Cake()` calls the **constructor** for the class, i.e. the `__init__` method shown above.

From outside the class, we can call the `print_ingredients` method on `c1`. (We cannot use `c1` from inside the class as not all instances of the class will be called `c1`.)

```
c1.print_ingredients()
```

This will produce the output:

```
****************
Cake Ingredients
****************

2 cups flour
1 cups Sugar
1 tsp baking soda
1 eggs
200 grams butter
```

This syntax takes a little getting used to. After all, we are used to seeing the number of parameters in function definitions match the number of parameters passed in when they are called! So what is happening? Well the `self` from our class definition becomes, in this case, a reference to `c1`. We can go on to create as many instances of cake as we like in the main routine.

```
c2 = Cake()  # self in this Cake will become a reference to c2
c3 = Cake()  # self in this Cake will become a reference to c3
```

(In each case `Cake()` is read as calling the `__init__` constructor method for the class Cake.) Of course if we call the `print_ingredients` method on `c2` and `c3` we will get exactly the same output as we did for `c1` since the instance variables are storing the same values for each instance. This is not necessarily, or usually, the case.

The vars function

The `vars` function, when given an object identifier as an argument, returns a description of the instance variables of the class. We can use this function to inspect the instance variable at various points in the program. The statement `print(vars(c2))` will produce the useful but not very user friendly output:

```
{'butter': 200, 'flour': 2, 'sugar': 1, 'bSoda': 1, 'egg': 1, 'baking_time':
30}
```

Accessing and changing the values of instance variables

When we are inside a class we refer to instance variables using `self` and dot notation e.g. `self.flour`. From outside the class definition e.g. in the `main` routine, these variables can be accessed using the instance name and dot notation:

```
instance_name.variable_name
```

e.g. for the `Cake` example, `print(c1.flour)` will currently output 2.

The values stored in instance variables can be modified from outside the class by simply assigning them a new value: `instance_name.variable_name = new_value` e.g:

```
c2.flour = 3.5
c2.bSoda = 2
```

A call to `c2.print_ingredients()` will now reflect these changes.
In contrast, a call to `c3.print_ingredients()` will not reflect these changes.

This direct modification of instance variables from "outside" the class (the example of `c2` above) is often considered bad OOP practice and should be used carefully. It violates the idea that a class / object "encapsulates" data and manages access to it. More of this as the chapter progresses (see e.g. Section 1.12).

The constructor with parameters

As well as `self` the `__init__` constructor method takes further parameters in the normal fashion and so instance variables can be initialised to the correct values from the beginning. Code Listing 1.2 shows the complete class followed by a `main` routine. The class now has another instance variable representing the number of layers in the cake. The `__init__` method takes a parameter, which initialises this instance variable and then scales the default ingredient quantities by multiplying each of them by `num_layers`. (In our fictional cake making world ingredients are mixed in the same bowl but layers are probably baked in separate tins.)

```
class Cake:
    """initialise instance variables"""
    def __init__(self, num_layers):
        """Scales the default parameters by multiplying by num_layers """
        self.num_layers = num_layers
        self.flour = 2 * num_layers
        self.sugar = 1 * num_layers
        self.bSoda = 1 * num_layers
        self.egg = 1 * num_layers
        self.butter = 200 * num_layers

    def bake(self):
        """ This instance method doesn't do much but could be
        developed to describe the behaviour of a cake being baked
        """
        print("baked")

    def print_ingredients(self):
        """ This instance method displays the cake ingredients """
        print("****************")
        print("Cake Ingredients for", self.num_layers, "layers")
        print("****************")
        print()
        print(self.flour, "cups flour")
        print(self.sugar, "cups sugar")
        print(self.bSoda, "tsp baking soda")
        print(self.egg, "eggs")
        print(self.butter, "grams butter")
```

```
#main routine - note that the class definition has finished with the end of the indented code
c1 = Cake(2)
c1.print_ingredients()
c2 = Cake(4)
c2.print_ingredients()
print("Done")
```

Code Listing 1.2

What would happen if you tried to create a third cake but didn't pass in the number of layers?

Note on main routines

For reasons described in the Appendices, as our programs grow in complexity, it is a good idea for the code of the main routine to run only if a special "if" statement is true, see the first line of the code below. Thus we would usually write the main routine above as follows:

```
if __name__ == "__main__":
    c1 = Cake(2)
    c1.print_ingredients()
    c2 = Cake(4)
    c2.print_ingredients()
    print("Done")
```

This is how we will write our main routines from now on.

Reminder about doctests

The `doctest` module is used to find texts (which look like IDLE interpreter / shell sessions) enclosed by `"""`. It executes the code following `>>>` and compares the text output with the given output from the program.

We need to import `doctest` with the statement:

```
import doctest
```

In the main routine, we can run the `doctests` by calling `testmod`.

```
doctest.testmod()
```

Or equivalently (by using the default parameter `verbose --` see Code Listing 1.3):

```
doctest.testmod(verbose = False)
```

If the program's output matches the required output in `"""` then the program runs as expected and no doctest messages are displayed. But only if there are failures (mismatches) between expected and program output, then messages are displayed. For example:

```
File "/home/cshome/t/tharper/Desktop/PythonText/ChapterOneUnitTesting.py", line 19, in
__main__.Basketball.display_average
Failed example:
    b1.display_average()
Expected:
    Player: Karen, Average goals per game: 4.0
Got:
    Player: Karen, Average goals per game: 0.4
**********************************************************************
1 items had failures:
   1 of   2 in __main__.Basketball.display_average
***Test Failed*** 1 failures.
```

We can set verbose to `True` to display all successes and failures.

```
doctest.testmod(verbose = True)
```

Be careful of extra spaces or new lines in the expected text (enclosed by """). In these cases the expected and program output may appear the same but it still counts as a fail. This is a common cause of errors.

Car Example

Task description

Write a class definition for a Car class that contains information about the average kilometres per litre that it can achieve and the volume of its petrol tank.

Write an instance method that returns the typical/standard distance the car can be driven on a full tank. Include a doctest that tests a sample input.

In the main routine create two instances of the class and print the typical range for each.

> The first car should typically average 20 kpl (kilometres per litre) and have a tank volume of 100 litres.
>
> The second car should average 24 kpl and have a tank volume of 80 litres

Coded Solution

```
class Car:
    def __init__(self, kpl, tank_vol):
        self.standard_kpl = kpl
        self.tank_vol = tank_vol

    def get_std_range(self):
        """
        >>> c = Car(100, 20)
        >>> c.get_std_range()
        2000
        """
        return self.standard_kpl * self.tank_vol

# main routine
if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose = True)
    car1 = Car(20, 100)
    car2 = Car(24, 80)

    print("The typical range of car1 is", car1.get_std_range()," km")
    print("The typical range of car2 is", car2.get_std_range()," km")
```

Code Listing 1.3

In the solution of Code Listing 1.3, the parameters bringing the data in to the `__init__` method (`kpl` and `vol`) are local variables to the `__init__` method. The data needs to be stored in instance variables so that other methods in the class can access it. The statement `self.standard_kpl = kpl` does this for the `kpl` variable.

The same name could have been used for the instance and the local variables, e.g. `self.kpl = kpl` would assign the value of the local variable `kpl` to the instance variable `kpl`.

Objects and self

The underlying idea behind Object Oriented programming is that one class definition can be used repeatedly to create many instance objects, all of which may have different data stored in them (instance variables) but can perform the same operations on it (methods).

The data and methods need to be accessible both from outside and inside the object. In Code listing 1.3, `car1` and `car2` are the names by which we refer to the objects from outside themselves. We could for example write `print(car1.tank_vol)`. But from inside the class we can't use `car1`, because that wouldn't work inside the object called `car2`. Instead we use `self` to refer to variables and methods of the object from within. It effectively means "belonging to this object". Exactly the same instruction will work for each object.

An analogy might be helpful. Think about issuing an instruction to go to the kitchen of a house. From the street you would need to say something like "go to the kitchen of house number 46". You need to know which house's kitchen you are referring to. From inside the house, you

don't see or need to know the street number. You can just say "go to the kitchen of this house". Exactly the same instruction can be used inside every house.

1.4. Exercises

Task 1

Write a class definition `BasketballPlayer` for a basketball player that stores a player's name, their number of games played and their total number of goals scored. Define a method that calculates and returns their average number goals per game. Define a method `display_average` that displays this average within a descriptive string. This method, should satisfy the following doctest:

```
def display_average(self):
        """
        >>>b1 = Basketball("Karen", 20, 80)
        >>> b1.display_average()
        Player: Karen, Average goals per game: 4
        """
```

In the `main` routine create several instances of `BasketballPlayer` and display each player's game average.

Extension: How might you determine who has the best average?

Task 2

Design and write a class representing a movie. It should have:

an `__init__` method for the class.

a method which displays the data.

Design and write a `main` routine which creates some movie objects and displays their data.

Consider adding a method to the `Movie` class to calculate and return the net profit. Include a call to this in your display method.

Possible instance variables:
Title
Producer
Director
Date
Length
Rating
Genre
Cost
Revenue

Task 3

Design and write a class representing a country. It should have:

an `__init__` method for the class.

a method which displays the data.

Design and write a `main` routine which creates some country objects and displays their data.

Possible instance variables:
Name
Continent
Leader
Population
Capital
Land area

Let your display method fulfill the following doctest (or something similar):

```
def display_data(self):
        """
        >>> c1 = Country("Ireland", "Europe", "Dublin", 6399152, 84421)
        >>> c1.display_data()
        Country name: Ireland, continent: Europe, capital: Dublin, population:
6399152, land area: 84421 km^2, population density: 75/km^2
        """
```

Consider adding a method to the `Country` class to calculate and return the population density.

Task 4

Design and write a class representing a cellphone. It should have:

an `__init__` method for the class.

a method which displays the data.

Possible instance variables:
Make
Model
Year
Camera?
Internet?
Price

Design and write a `main` routine which creates some cellphone objects and displays their data.

Consider adding a method to the `CellPhone` class to calculate and return the depreciated value of the cellphone after 2 years, if it depreciates at 67% per year (actually the correct depreciation rate at July 2011 according to the IRD – fun fact!).

Let the `display_data` (or appropriately named) method satisfy the following doctest:

```
def display_data(self):
    """
    >>> p1 = CellPhone("Nokia", "6610", 2002, 400)
    >>> p1.display_data()
    Make: Nokia, Model: 6610, Year: 2002, Price: $400.00
    """
```

Let the `depreciated` method satisfy:

```
def depreciated(self, years):
    """
    >>> p1 = CellPhone("Nokia", "6610", 2002, 400)
    >>> p1.depreciated(9)
    Depreciated value after 9 years: $10.88
    """
```

1.5. Data Abstraction

A key skill when you are designing classes is deciding what information about a situation or object you need to capture. This decision making process is called data abstraction. Think about the class you just wrote for `BasketballPlayer`. Basketball players are people with jobs, qualifications, addresses, parents etc. Your class didn't mention any of those things because they were simply not required for our program.

The `Cake` class that we have written seems like it would be at home in a bigger Recipes program. A cake class being used inside some kind of Nutritional Value program might be quite different e.g.; it might have a `calculate_total_calories` method or a method that displayed the disturbing amount of energy each cake contains.

Just to demonstrate the point further, the program in the next section uses a very different `Cake` class to the one we've seen so far. This one displays a graphical image. This class definition is concerned not with the ingredients of a cake but solely what it looks like.

1.6. Graphical Cakes

We are digressing a little here to prove a point. Let's make some objects that you can actually see. There is an alternative `Cake` class, which we have named `GraphicalCake`, a `BakeryWindow` class and a `main` routine in Code Listing 1.4.

Keep Calm (when looking at this code for the first time ;-)

`GraphicalCake` has some code to do with Graphical User Interfaces (`tkinter`) and Graphical drawing that will be explained later in this workbook. For now, you just need to know that it has a draw method that will create graphical output. Instances of `GraphicalCake` are being created and managed in the `Bakery` class. The bits for you to focus on (and that should hopefully be starting to look familiar) are in bold.

Run the module of Code Listing 1.4 . You should get output resembling the image below but with prettier colours (unless someone has splashed out on colour printing for you).

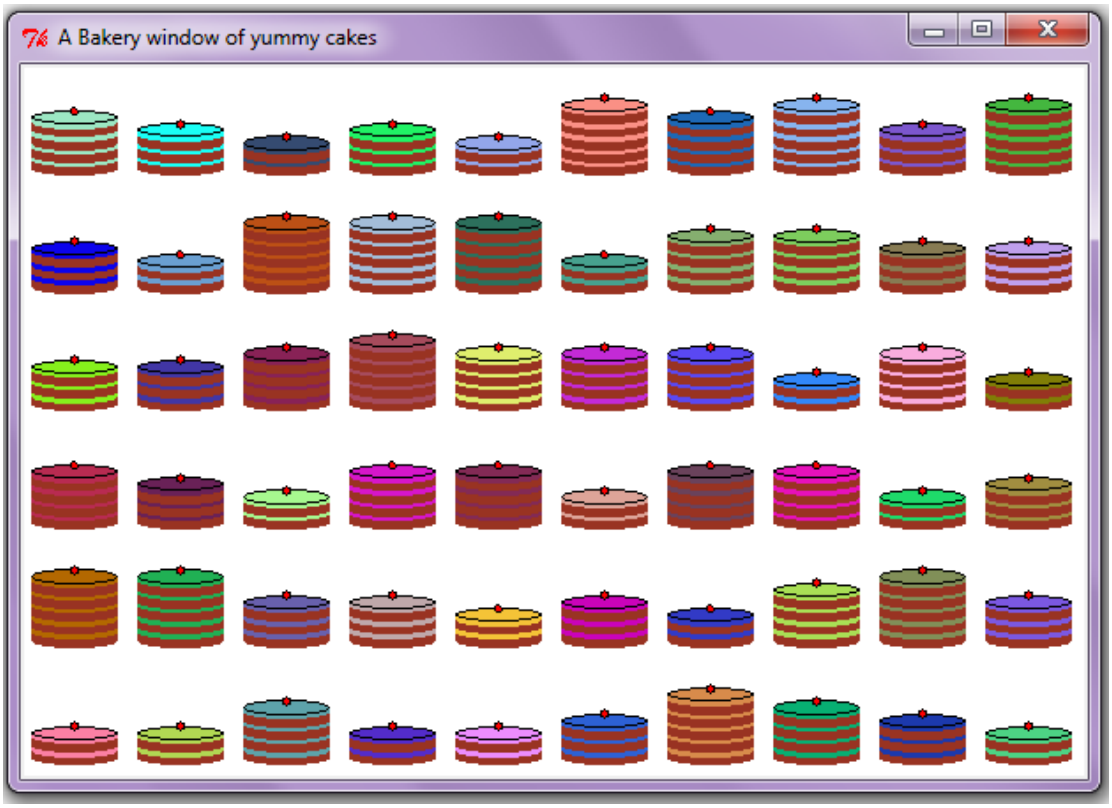


Figure 1.1

Each `GraphicalCake` object can clearly be seen. These are graphical representations of instance objects. (When run the window above may be hidden behind other windows at the top left of the screen.) Although there is just one `GraphicalCake` class, 60 instances of it have been created. These instances are not identical because they each have different data generated randomly when they are created.

Experiment with the `__init__` method of `Bakery` below to instantiate different numbers of cakes.

It is not expected that you understand this code at this point. Just how to run it.

As written, the code should be in one file (which could be called bakery.py).
The classes could be in separate files, and the `Bakery` class imported by the `Cake` class e.g. if `Bakery` is in a file named `baker_boys.py`, import it in the `Cake` file using

```
from baker_boys import Bakery
```

```
from tkinter import *   # makes graphics possible, using the tkinter module – see Chapter 2
import random

class GraphicalCake:

   def __init__(self, canvas, x, y, w, h):
      """ Sets up coordinates of cake's bounding area """
      self.canvas = canvas
      self.x = x
      self.y = y
      self.w = w
      self.h = h
      self.num_layers = random.randrange(2,6)
      self.cake_colour = "#993322"
      self.fill_colour = self.random_colour()

   def draw(self):
      """ Sets up coordinates at which cake begins to be drawn. These coordinates are based
      on those of the cake's bounding area but allow for some margin around the edges  """

      x = self.x + self.w / 10
      y = self.y + 3 * self.h / 4
      w = 4 * self.w / 5

      thickness = w / 10
      filling_thickness = w / 20
      h = thickness + filling_thickness
      cherry_r = filling_thickness

      for i in range(self.num_layers):
         # lower oval - cake body
         self.canvas.create_oval(x, y + thickness, x + w, y + thickness
                   + h, fill = self.cake_colour, outline = self.cake_colour)
         # rectangle for thickness - cake body
         self.canvas.create_rectangle(x, y + h / 2, x + w, y + thickness
                  + h / 2, fill = self.cake_colour, outline = self.cake_colour)
         # top oval - cake body
         self.canvas.create_oval(x, y, x + w, y + h, fill = self.cake_colour)

         y -= filling_thickness
         # lower oval - filling
         self.canvas.create_oval(x, y + filling_thickness, x + w,
                     y + filling_thickness + h, fill = self.fill_colour,
                     outline = self.fill_colour)
         # rectangle for thickness - filling
         self.canvas.create_rectangle(x, y + h / 2, x + w,
                     y + filling_thickness + h / 2, fill = self.fill_colour,
                     outline = self.fill_colour)
         # top oval - filling
         self.canvas.create_oval(x, y, x + w, y + h, fill = self.fill_colour)
         y -= thickness

      # cherry on the top
      self.canvas.create_oval(x + w / 2 - cherry_r, y + cherry_r,
                        x + w / 2 + cherry_r, y + 3 * cherry_r, fill = "red")

   def random_colour(self):
      """Returns a random colour as a hexidecimal string"""
      hex_chars =['a','b','c','d','e','f','1','2','3','4','5','6','7',
         '8','9','0']
      s = "#"
      for i in range(6):
         s += hex_chars[random.randrange(len(hex_chars))]
      return s
```

```
class Bakery:

   def __init__(self, parent):
      """Dictates the size of the "bakery window" and how many "shelves" it has """
      NUM_ROWS =6       # the number of "shelves - too many may produce overlap"
      NUM_COLS = 10     # How many cakes per shelf
      CW = 600          # width of the canvas
      CHT = 400         # height of the canvas

      canvas = Canvas(parent, width = CW, height = CHT, bg = "white")
      canvas.grid(column = 0, row = 0)

      cakes = []
      # Loop below creates enough cakes to fill the window and appends to a list
      # Each cake is passed the x, y, width, and height of its bounding area
      # so it can calculate the coordinates at which to draw itself
      for i in range(NUM_ROWS):
         for j in range(NUM_COLS):
            cakes.append(GraphicalCake(canvas, j * CW / NUM_COLS,
                         i * CHT / NUM_ROWS, CW / NUM_COLS, CHT / NUM_ROWS ))

      # Loop below asks each cake to draw itself
      for i in range(len(cakes)):
         cakes[i].draw()

# main routine will run if we run this module (in IDLE, F5 this file)
if __name__ == '__main__':
   root = Tk()
   root.title("A Bakery window of yummy cakes")
   b = Bakery(root)
   root.mainloop()
```

Code Listing 1.4

1.7. UML class diagrams

It is very useful to have diagrammatic representations of the structure of a class. The Unified Modelling Language (UML) is one way of representing objects on paper. A UML class diagram shows a class in a box with separate sections for:

- the class identifier – Python convention suggests a class name should always start with an uppercase letter e.g. `Cake`. (Some of the Python built-in modules break this convention by containing classes beginning with lower case letters, e.g. `str`, `list`.)
- a list of the instance variables, in the pattern `identifier : data type`
- a list of the methods, in the pattern `identifier(parameter list):return type`

  if a Python method or function does not return another value, by default it returns a special value called None.

  Parameters (if any) are in the pattern `identifier : data type`
  If there is more than one parameter, they are separated by a comma.

Figure 1.2 shows two UML class diagrams. The first is the UML class diagram for the `Cake` class of Code Listing 1.1, and below for the `GraphicalCake` class of Code Listing 1.4. `Cake` has 6 instance variables while `GraphicalCake` has 4. Both classes use `num_layers`. `Cake` has 2 methods while `GraphicalCake` has 4 methods, some with parameter/s.

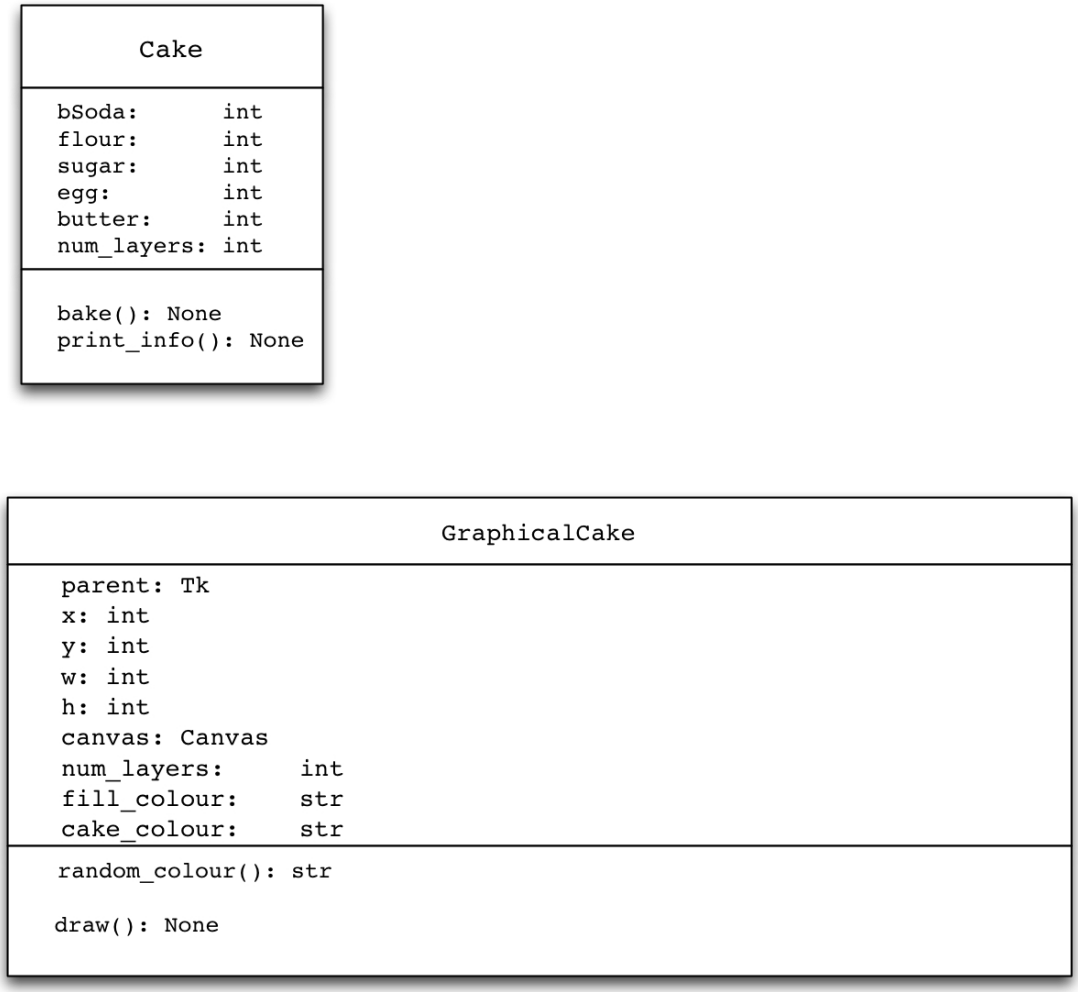


Figure 1.2

1.8. Object diagrams

While learning to use objects, it can be useful to use an object diagram, which includes a representation of the instance objects involved, in combination with an extended UML class diagram, which shows the values held by each object.

The object diagram in Figure 1.3 represents the `Cake` class from Code Listing 1.2 together with its `main` routine. It includes the values held in the instance variables of the resulting objects **after** the code has run. The values are the same in each object, but this is not usually the case. Each method is enclosed in an oval, offering the opportunity to list local variables within the method if desired. The `main` routine is showing its local variables, `cake1` and `cake2`, and their data type: `Cake`.

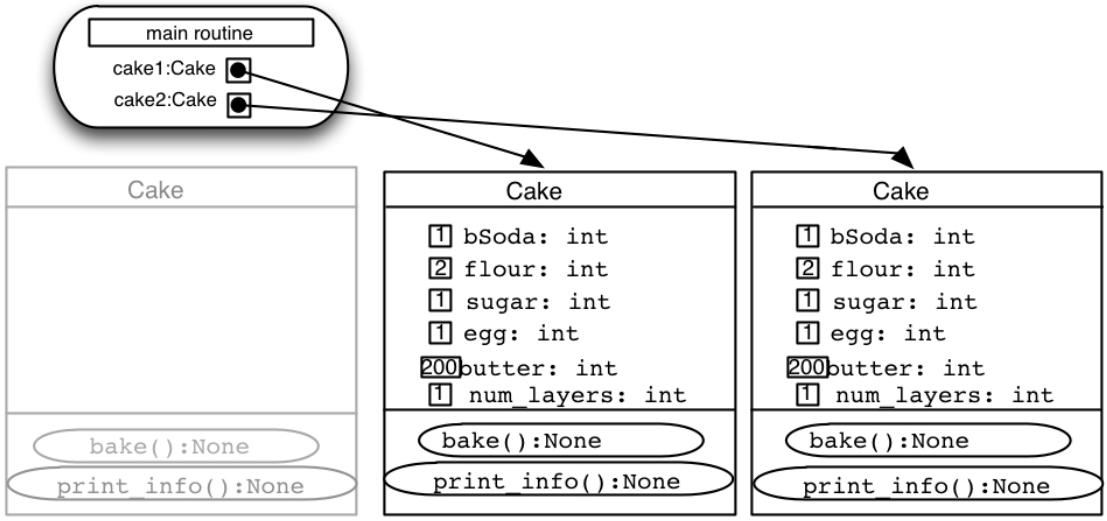


Figure 1.3

Figure 1.3 shows one box on the left in grey. This box represents the class object which exists automatically – without any instantiation required, after running (or importing) the file / module. The other two boxes on the right are instance objects. The `main` routine is pictured above.

There are two black versions of the `Cake` class to the right, one representing the `cake1` instance object and the other representing the `cake2` instance object. The variables `cake1` and `cake2` have arrows to these objects. The arrows depict the variable `cake1` referring to / pointing to an instance of `Cake` and `cake2` referring to another instance of `Cake`. These are instance objects, created from the "recipe" specified by the class object.

Notice that `self` does not appear in any diagram or parameter list - its presence is assumed, and must be remembered. Don't worry, the interpreter will remind you with a message about arguments.

An **argument** in programming is a value that is sent into a method or function, as distinct from the data that is received by the method or function in its formal definition. We have been accustomed to using the term **parameter** for both of these.

`def my_function(x, y):`	x and y are parameters, listed in the formal function definition
`my_function(3,6)`	3 and 6 are arguments in this function call

Task 5

Using the object diagrams shown below in Figure 1.4, write a `Person` class which stores the name of the person, the year of birth and the gender (whether or not this `Person` is male). Write two methods:

(1) `get_age` which takes the current year (this_year) as input and returns the Person's approximate age (we are not going to worry about which month it is). Satisfy this doctest:

```
    def get_age(self, this_year):
        """
        >>> p = Person("Timothy", 1967, True)
        >>> p.get_age(2013)
        46
        """
```

(2) `print_info` which displays the name, approximate age and gender of the person.

Note if you want Python to identify `this_year` automatically, rather than hard-code it:

```
import datetime # this statement goes at the top of the file, before the class header

    # these next 2 lines store the current year in this_year,
    # use them in the method which calls the get_age method
    now = datetime.datetime.now()
    this_year = now.year
```

Remember that both methods require `self` as the first parameter - this goes without saying!

In the `main` routine create two `Person` objects with appropriate names, ages and gender and print the information. For example, the expected output in response to Person objects created in 2012 using the statements:

```
person1 = Person("Joe",2000,True)
person2 = Person("Freda",1990,False)
```

would be:

```
Joe will be 12 this year. He is male.
Freda will be 22 this year. She is female.
```

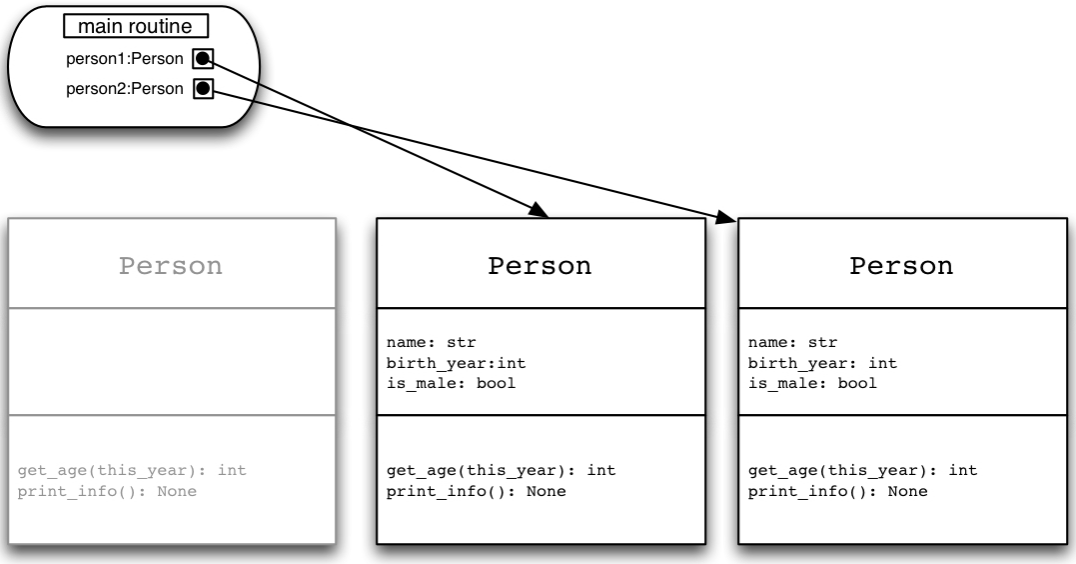


Figure 1.4

1.9. Flow Diagrams within Objects

UML class diagrams show the structure of a class. Object diagrams show the relationship between objects in the program. Neither of them describes the processes within the class. The Cake UML class diagram of Figure 1.2 is based on the `Cake` class of Code Listing 1.1. We can't see when the `bake` method should be called, or what it does. There is still a place for the traditional pseudocode and / or flowchart to describe processes within methods. The `Cake` class isn't a complete class as some of its methods aren't yet written, but Figure 1.5 shows some possible flowcharts for the program.

The left division describes the `main` routine. The object identifiers and their data types are listed as they are instantiated (e.g. `cake1:Cake`).

The right division describes discrete processes (methods) in the `Cake` class. *Most of the methods are not yet written - this is an example of top-down design in progress. The overview or top-level of the program has been defined, and it is clear what the purpose of the empty methods is. Classes could be written and tested to this design before going on to plan the processes within the next level of methods.*

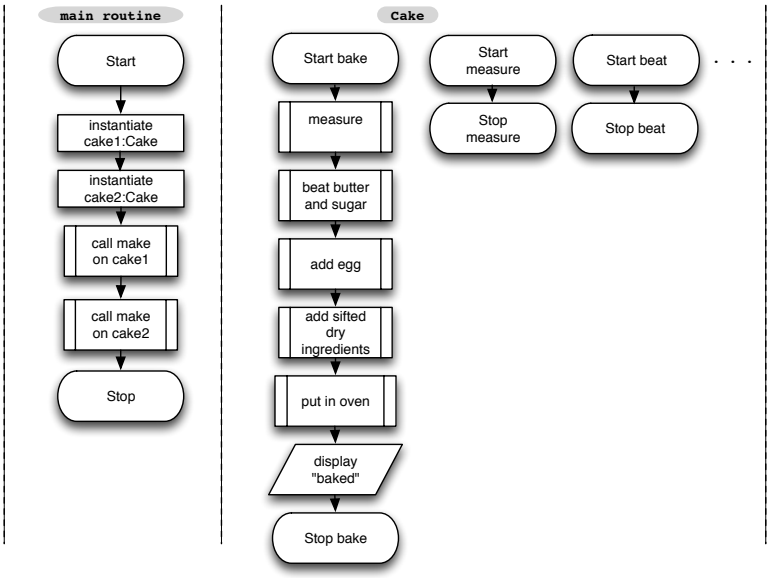


Figure 1.5

The choice we have made for diagrams in this workbook is to describe processes within each method using flowcharts or pseudocode, but generally not to draw arrows between classes or methods. The main routine is usually the location of the code which defines the main sequence of steps in the program in the case of text-based (non-graphical) programs.

Designs using objects are more complex to put on paper, as not only the object design needs to be described, but also the processes which happen both between and within the objects. There is no "one size fits all" approach, but a combination of UML class diagrams, object diagrams and flowchart or pseudocode is likely to be useful at this level.

Task 6

A design for a simple class named `OneString` is shown in Figure 1.6. It includes an extended UML class diagram, a flowchart for the `main` routine and a flowchart for the `show_string` method. Note that the class diagram doesn't show the `__init__` method. It is just assumed.

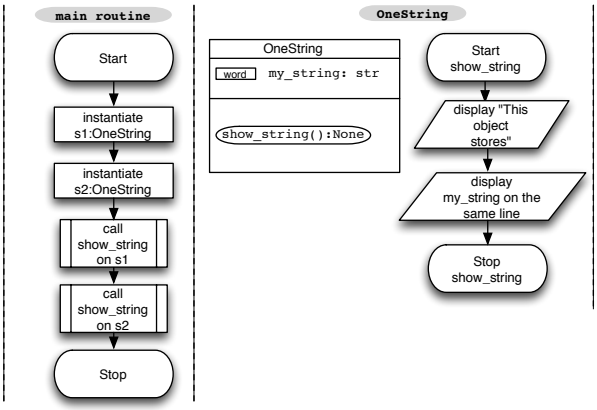


Figure 1.6

Code the class `OneString` to the design. Remember to initialise the instance variable to the value "word". Include the following doctest:

```
    def show_string(self):
        """
        >>> o = OneString()
        >>> o.show_string()
        This object stores word
        """
```

Write the `show_string` method by following the specifications in the `OneString` flowchart. Compile the class to make sure it has no syntax errors. You cannot run the program until it has a main routine.

Write a main routine below the class which makes two instances of `OneString`, then calls the `show_string` method on each. The application flowchart describes this process. Compile and run the application class.

Do the two objects have the same output i.e. do they display the same data?

Are the objects equal i.e. does s1 == s2? If not, why not?

How could you test this?

Complete the object diagram below adding the instance object references to the main routine and drawing arrows to the objects these represent.

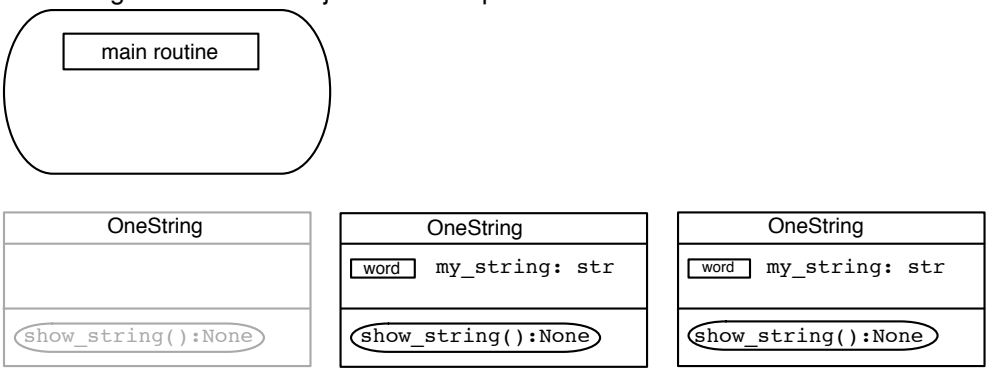


Figure 1.7

Another pattern to initialise instance variables

Any instance variable can be created in the `__init__` method, but it can also be created by any other method in the class. The ability to change the value stored in an instance variable is often essential. This can be done any method of the class, and also, as we saw in Section 1.3, in the main routine. (See discussion in Section 1.12 on these points!)

The extended UML class diagram and pseudocode of Figure 1.8 describes a change to the `OneString` class. The instance variable `my_string` is no longer instantiated in the `__init__` method. Instead, the `__init__` method calls the `set_string` method, which initialises the variable.

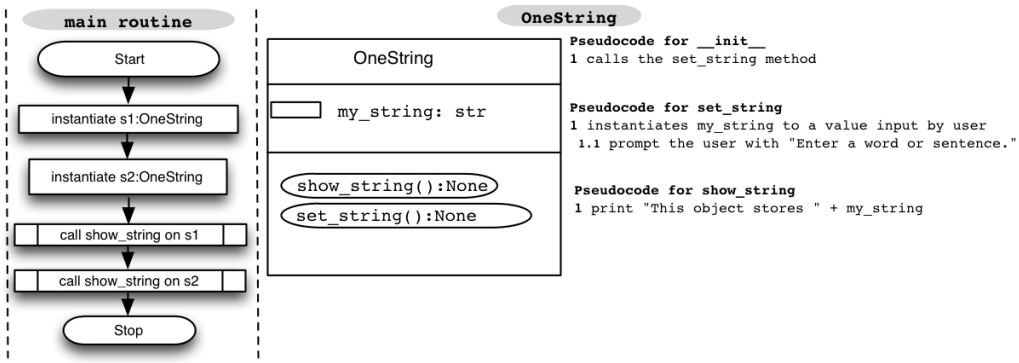


Figure 1.8

Task 7

Write the `OneString` and main routine to the specifications of Figure 1.8. Run the program. Type in two different words, e.g. "first" and "second" when prompted. You should see different outputs. In Figure 1.9 add the variables to the main routine and draw arrows to the instance objects these represent. Fill in the values you typed in the correct instance variable boxes.

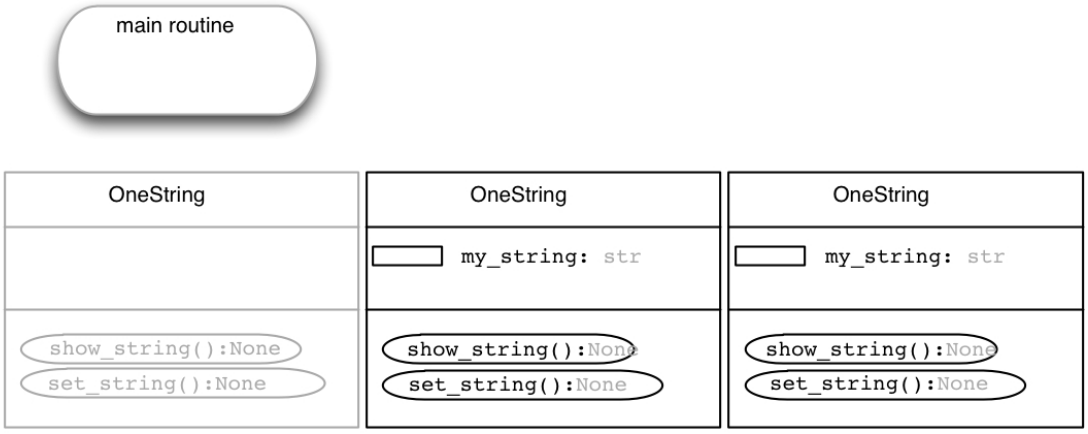


Figure 1.9

Points to note about instance variables

Although each instance object is created from the same class, and can perform the same methods on its data, the data stored in each object is different.

Instance variables can be created in any method of a class.

Instance variables can be given new values in the main routine or in any method in the class (but see Section 1.12).

An `__init__` method is not essential in a class, but a well-constructed class would generally have an `__init__` method which takes care of the initialisation of instance variables.

Task 8

The UML class diagram, flowchart and pseudocode in Figure 1.10 describe a program.

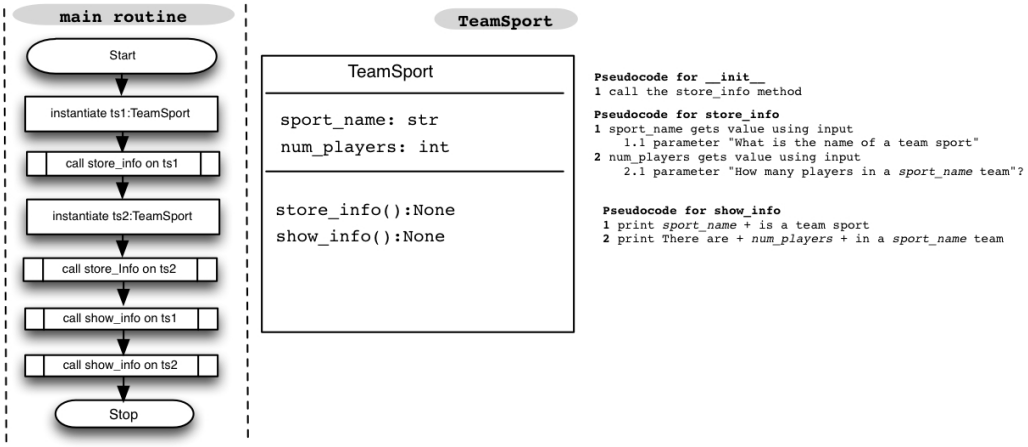


Figure 1.10

Write the class `TeamSport` to this design. Write a main routine which makes 2 instances of TeamSport and calls their methods as described in the flowchart. A sample output might be:

```
Cricket is a team sport
There are 11 players in a cricket team.
Netball is a team sport
There are 7 players in a netball team.
```

Combining classes with ordinary functions

A class definition is finished when its indenting finishes. In our examples so far the class definition is followed immediately by the main routine. It is also possible, as you did before you knew about classes, to define ordinary functions (unlike methods these are not indented into the class and do not use `self`). Code Listing 1.5 has an `Applicant` class and a `main` routine which uses a `collect_data` function. The main routine collects data and creates the objects using the values obtained.

```
class Applicant:
    def __init__(self, app_name, rating):
        self.app_name = app_name
        self.rating = rating
    def show_rating(self):
        print(self.app_name, "Rating", self.rating)

# the class definition has ended - an ordinary function follows
def collect_data():
    global name   # these variables are global so they will be in scope in the main routine
    global rating
    name = input("What is the applicant's name: ")
    rating = int(input("Rating: "))   # this should really be in a try except ValueError block - if this is
        done, there would be some flow-on effects to fix: (rating may not exist, so Applicant creation would fail with NameError)

# main routine
if __name__ == "__main__":
    collect_data()
    applicant1 = Applicant(name,rating)
    collect_data()
    applicant2 = Applicant(name,rating)
    applicant1.show_rating()
    applicant2.show_rating()
```

Code Listing 1.5

The `collect_data` function isn't strictly necessary here, as its statements could be repeated in the `main` routine. But the more complicated the process for collecting the data is, and the more often it will be called, the more likely it should be in a separate function. Actually this function should be more complicated, because the rating value should be checked for being an integer as it is entered.

When designing classes, it is best to try to create a class using `__init__`, with data which already exists.

Task 9

Plan, and then write, to these specifications:

A class represents a school. It stores data representing the school's name, the number of pupils on the roll, and the number of classrooms it has.

It has an `__init__` method to set the three instance variables.

It has a method that returns the average number of pupils per classroom.

It has a method (i.e `show_info`) that displays the school's name and the average (formatted to 2 significant decimal places) e.g. `Kaikorai Primary has 26.30 pupils per room`. Include the following doctest:

```
def show_info(self):
    """
    >>> s = School("Eveyln Intermediate", 1500, 96)
    >>> s.show_info()
    Eveyln Intermediate has 15.62 pupils per room
    """
```

Write a function, to be called by the `main` routine, which declares 3 global variables, one for each data item, then fills them with values typed by the user e.g.

```
global school_name
school_name = input("Enter the name of the school")
```

Write a `main` routine which

calls the function which sets the 3 variables
creates a `School` object called `school1` using the variables
calls the display method on the object

Repeat this sequence for another `School` object called `school2`

1.10. Lists of Objects

Let's go back to the `Cake` class we began this chapter with. As with any recipe, it is possible to make great numbers of `Cake` objects from the `Cake` class. So far in our code examples we have created only 3 or 4 instances of `Cake()` and each one of them was named. That is all very well for small numbers of things but what if we wanted a hundred `Cake` objects, or a thousand, or a million? It would make a lot of sense to store them in a list. We can even instantiate them in a loop.

Here is some code which will do just that.

```
# new code to go in the main routine for Code Listing 1.1
# appends a Cake instance to cakes[ ],  3 times
# main routine
if __name__=="__main__":

    cakes = []
    for i in range(3):
        cakes.append(Cake ())

    # And then we can bake them in a loop as well.
    for cake in cakes:
        cake.bake()
```

Code Listing 1.6

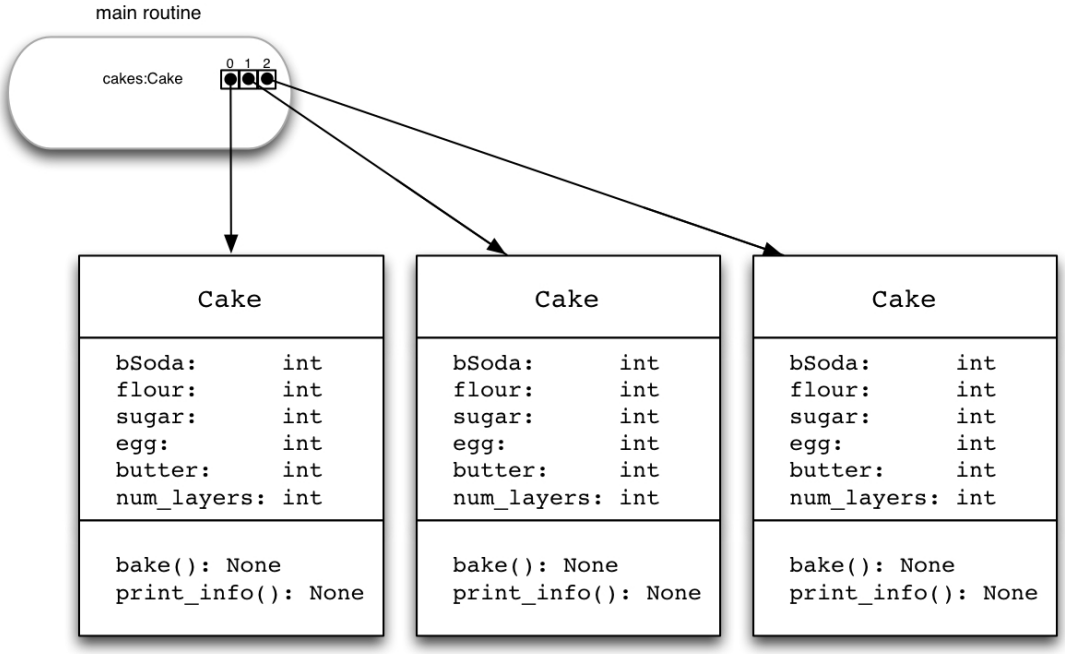


Figure 1.11 (Class object box omitted)

With no change to the `Applicant` class, or the `collect_data` function of Code Listing 1.5, the `main` routine of Code Listing 1.7 makes 4 `Applicant` objects and stores them in a list. It could just as easily make 1500 `Applicant` objects with no extra code required.

```
# main routine
if __name__=="__main__":

    applicants = []
    for i in range(4):
        collect_data()
        applicants.append(Applicant(name,rating))

    for applicant in applicants:
        applicant.show_rating()
```

Code Listing 1.7

Task 10

Convert the schools program from Task 9 so it stores the references to the `School` objects in a list. Follow the example in Code Listings 1.6 and 1.7. All changes will be to the `main` routine. The `School` class will not need to change at all.

Use a loop to instantiate each object in turn and append it to the list.

Use another loop to display each object's data in turn.

Extension: In the `main` routine, determine the school with the lowest class size and display that information.

Task 11

A computer game requires a squadron of helicopters, which are created from a class named `Helicopter`. You know from the documentation that `Helicopter` has methods called `down` and `up`. The UML class diagram for the `Helicopter` class is shown in Figure 1.12. Code this class, with the `down` method displaying "down", and the `up` method displaying "up". Display the instance variable `number` before the `up` or `down` message .

Design and write a main routine which creates a list of 100 `Helicopter` objects, sending each a number representing its creation order. Call the `up` method on the first 50, and the `down` method on the next 50. For example:

```
. . .
49 up
50 up
51 down
52 down
. . .
```

Test your program.

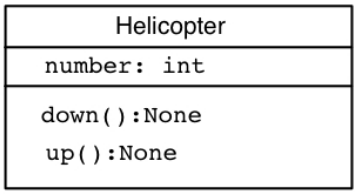


Figure 1.12

Task 12

A computer game requires 35 `Actor` objects. The `Actor` class has a `gender` instance variable, which is "**male**" by default. It has a `set_female` method, which will change the gender value to "**female**". It has a `show_gender` method, which displays the gender.

The extended UML class diagram is shown in Figure 1.13.

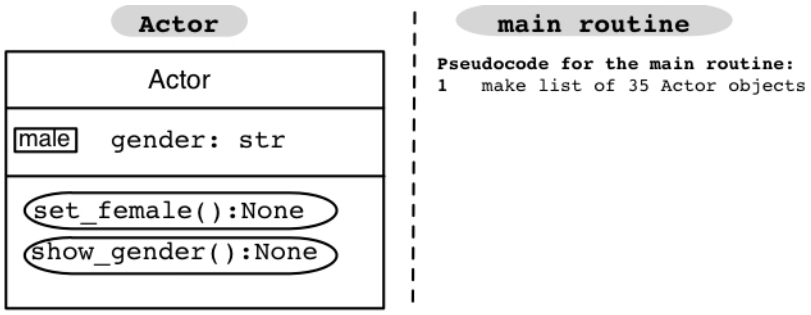


Figure 1.13

First, write the `Actor` class to the specification described in Figure 1.13.

Begin by finishing the pseudocode. It should describe a program to these specifications:

- make a list of 35 `Actor` objects.
- make every 5th `Actor` object a female – try to find a pattern rather than list every possibility in an if statement
- use a loop to display the gender of each `Actor` object in the list.

Then design, write and run the main routine.

Test that your program behaves correctly.

Robustness: Try changing the list size. With no other adjustments, does your code still perform the task correctly? Try 80, 203, 1001 . . .

Task 13

Write a `House` class which stores and displays information for a real estate agent.
A class diagram is provided. Write the `__init__` method. Write the methods `calculate_price_per_area` and `calculate_price_per_rooms` to calculate the price per square metre and price per number of bedrooms respectively. The `show_info` method should display a summary of the data, as to a client. Include the doctest:

```
    def show_info(self):
        """
        >>> h = House("DGB2353", "14 Orion Terrace", 2, 595000, 340)
        >>> h.show_info()
        ID: DGB2353 Address: 14 Orion Terrace
        Number of bedrooms: 2   Asking price: $595000
        """
```

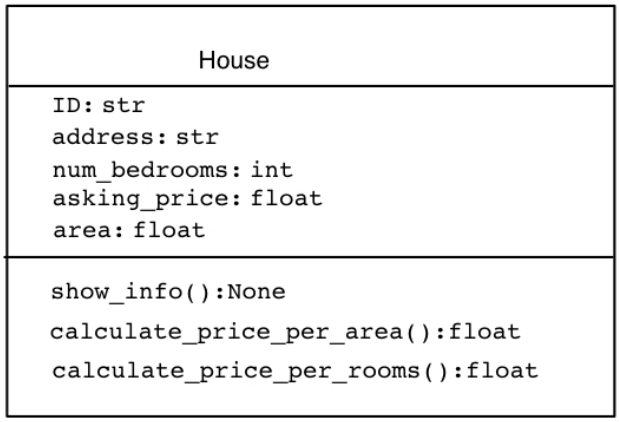


Figure 1.14

The main routine should create a list of three `House` objects by hardcoding the values e.g.

```
houses=[]
houses.append(House("DGB2355","2 Aston Crescent", 3,110000, 190))
```

etc.

A for loop should call the `show_info` method of each object . A sample output might be:

```
ID:DGB2354 Address: 22 Helens Rd
Number of bedrooms: 4   Asking price: $320000
Price by area $1333
Price per room $80000
***********
ID:DGB2355 Address: 2 Aston Crescent
Number of bedrooms: 3   Asking price: $110000
Price by area $578
Price per room $36666
***********
ID:DGB2356 Address: 5 Stratton St
Number of bedrooms: 5   Asking price: $550000
Price by area $1447
Price per room $110000
***********
```

1.11. Default parameters

Python allows us to increase the flexibility of methods (and functions – we just didn't mention it before) using *default parameters* to allow methods or functions to be supplied with a different number of arguments. This feature may prove particularly useful where the `__init__` method is concerned.

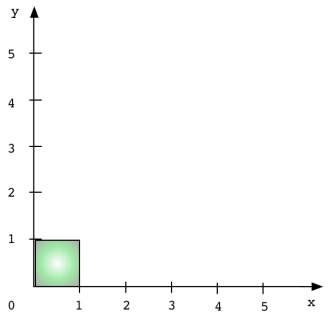


Figure 1.15

Default parameters will be demonstrated by introducing a `Rectangle` class. Its `__init__` method (see Code Listing 1.8) gives us flexibility and economy regarding the different ways to define a `Rectangle`. By default, we can consider a rectangle to have a position on a Cartesian Plane, with its bottom left corner at the origin (0,0). With no additional information, it is a unit rectangle where width, height and scale all have a measurement of one.

Figure 1.15 shows this default Rectangle.

```
class Rectangle:
    """
    The __init__ method for the rectangle class is defined in terms of its x and y position resting on a
    2-dimensional plane and the parameters: width, height and scale have default values.
    By default it is a unit rectangle
    """
    def __init__(self, pos_x, pos_y, width = 1, height = 1, scale = 1):
        """
        >>> r = Rectangle(0,0)
        >>> r = Rectangle(0,0, 2,3)
        >>> r = Rectangle(0,0, scale=5)
        >>> r = Rectangle(1,1,height=5)
        >>> r = Rectangle(1,1, width=5)
        """
        self.pos_x = pos_x
        self.pos_y = pos_y
        self.width = width * scale
        self.height = height * scale
```

Code Listing 1.8

The only two required parameters are the x and y coordinates:

```
pos_x, pos_y
```

The default parameters are:

```
width = 1, height = 1, scale = 1
```

A Rectangle object can be instantiated in different ways. A unit square (i.e. a rectangle that measures 1 in each dimension) at the origin needs no default parameters set.

```
r1 = Rectangle(0, 0)
```

The `width`, `height` and `scale` of `r1` are all set to the default value of 1, the default values in the `__init__` method's parameter list.

Alternatively we could pass in some different values for `width` and `height`:

```
r2 = Rectangle(1, 2, width = 2, height = 3)
```

In the case of `r2`, `scale` remains set to 1, its default value in the `__init__` method's parameter list.

In fact we do not need to mention the names of the default parameters IF THEY ARE IN THE RIGHT ORDER. We could just specify the values to get the same result:

```
r3 = Rectangle(1, 2, 2, 3)
```

So, `r3` has a `width` of 2, a `height` of 3 and still a default `scale` of 1. Exactly the same as `r2`.

Another thing we can do is create squares of different sizes quite simply by changing the value of `scale`:

```
r4 = Rectangle(2, 0, scale = 2)
r5 = Rectangle (3, 4, scale = 8)
```

Or we could specify the first three values (2 required, 1 default) and leave the rest (`height` and `scale`) at their default values:

```
r6 = Rectangle(0, 0, 2)
```

The order of default parameters does not matter IF we specify the names of the default parameters:

```
r7 = Rectangle(0, 0, scale = 5, width = 3, height = 1)
```

The general rule is that all required parameters are supplied in order and default parameters come after the required parameters.

So the method or constructor header could be represented in the form:

```
def method_name( ... required_parameters ... , ... default_parameters...):
    # method body
```

Cake example with default parameters

Let us apply this to the `Cake` class. Sometimes experienced bakers like to tweak their ingredients to compensate for a different sized baking tin or to compensate for the fact that the baking soda is a bit old and may have lost some punch or to achieve a slightly different texture. Let us change the `Cake` class so that the default amount of ingredients per layer can be passed in as parameters. We can use default parameters to provide the default amounts of ingredients if they differ from the usual values.

```
class Cake:

    def __init__(self, num_layersVal, flour = 2, eggs = 1,
                               sugar = 1, bSoda = 1,  butter = 200):
        self.num_layers = num_layersVal
        self.flour = flour * self.num_layers
        self.eggs = eggs * self.num_layers
        self.sugar = sugar * self.num_layers
        self.bSoda = bSoda * self.num_layers
        self.butter = butter * self.num_layers
```

Code Listing 1.9

Try this: Which of the following is not a perfectly valid creation of a Cake instance?

```
c1 = Cake(3, eggs = 2)
c2 = Cake(4)
c3 = Cake(2, 2.1, 2, 1, 1.5, 220)
c4 = Cake(bSoda = 2)
c5 = Cake(2, sugar = 0.5)
```

Task 14

For the `Rectangle` class of Code Listing 1.8 write a `display_position` method which displays the `pos_x` and `pos_y` values according to the doctest:

```
    def display_position(self):
        """
        >>> r = Rectangle(0,0)
        >>> r.move(1,2)
        >>> r.display_position()
        x: 1, y: 2
        >>> r.move(1,-1)
        >>> r.display_position()
        x: 2, y: 1
        """
```

Write a `move` method for the `Rectangle` class that changes the position of the `pos_x` and `pos_y` coordinates of rectangle objects by values passed in as parameters. Define it so that by default, it moves the rectangle 1 along the x axis and 2 along the y axis.

Write a main routine which alternately moves the x and y position of a `Rectangle` object, then displays the position. Use different arguments to test your method.

On Figure 1.15, draw the `Rectangle` in its next position according to your code.

Task 15

At Fred's Fast Food a burger combo consists of a burger, a medium chips and a medium soft drink. `ComboOrder` objects are created whenever a customer makes an order. As part of a health-drive the check out person no longer offers to "super-size" the order. Instead customers are only asked about the type of burger they want and the type of soft drink. If a customer requests it the order may be "super-sized" meaning both the chips and drink size are increased to large.

Write a `ComboOrder` class with an `__init__` method that requires the type of burger, the type of soft drink but provides default parameters for the size.

Design and satisfy your own doctests for a display method that prints the information for the person putting the order together to see. A possible input and output might be:

```
order1= ComboOrder("cheese", "cola")
order2 = ComboOrder("lamb", "lemonade", "large")

Burger: cheese
Drink:  medium cola
Chips:  medium

Burger: lamb
Drink:  large lemonade
Chips:  large
```

1.12. Encapsulation and OO design

These topics are important to designing well written programs, so we're going to spend a bit of time on them.

Encapsulation

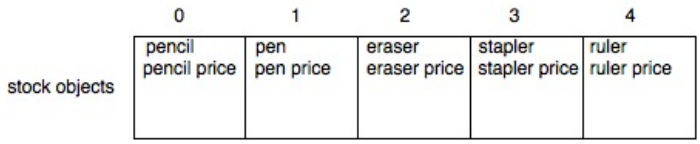
There is lots of terminology and theory relating to ideas of good OO design, but the term used in the Standard is **encapsulation**. It's a fuzzy concept, a combination of two closely related ideas, that the programmer should (1) bundle together data and the methods operating on it (into classes / objects), and (2) manage access to some data or methods.

The first of these points relates to the general design of programs and is always applicable. The whole point of OOP is to put data which belongs together, and methods which work on the data, together into a class, where classes represent "agents" or "entities" or significant aspects of the programming task. The concepts of **cohesion** and **coupling** discussed below are highly relevant. The more self-contained and robust those classes are (the easier to move between programs) the better we have done our job.

The second point is driven more by specific task requirements. It may be more or less relevant in different tasks, and it is supported by a range of different features in different programming languages. The underlying assumption is that the methods of the class have been written and tested together, and can be trusted to protect the data stored within the class. Code outside the class is less "trusted" (possibly written by someone else) and thus access from outside the class should be managed. After the discussion of cohesion and coupling, the remainder of this section explores these issues.

Cohesion and coupling

Remember from our Introduction in 1.1:



Objects should keep related data and the methods that operate on the data together. This is called **cohesion**. The program is thereby broken down into coherent units, each with its own purpose. The objects can interact with each other when required. This theoretically makes it easier to manage large-scale projects.

Coupling is the term used to describe an object accessing the instance variables or methods of another object. Some coupling is essential or the objects could never work together, but too much coupling suggests the class design could be improved.

We want to design classes with **high cohesion** and **minimal coupling**. In Figure 1.16 cohesion is illustrated by the double-ended arrows between each class's methods and its own data. The dotted line suggests the interaction between data and methods. Coupling is demonstrated by the arrows between classes. Encapsulation is symbolised by the heavy border around each class. The coupling arrows which cross the border lines should be kept to a minimum. Some coupling of course is essential, as dictated by the requirements of the task.

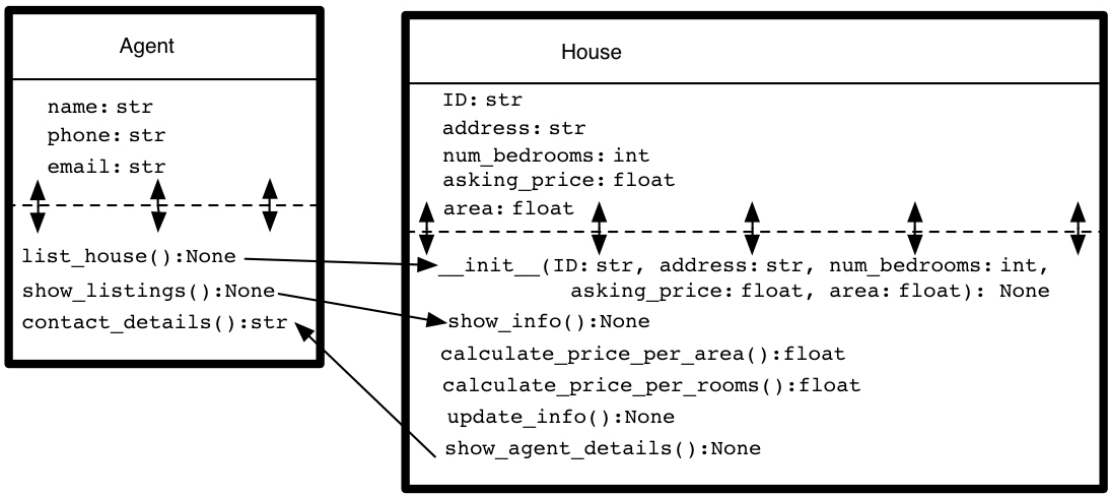


Figure 1.16

Managing access to data is discussed further in the Appendix.

Adding instance variables from anywhere

In many languages the instance variables (or data fields) are explicitly declared. Python, as we already know, is considerably more relaxed, and allows us to create variables "on the fly", e.g. in assignment statements. It would still be considered good practice by most to set up all the instance variables of a class in one place, the constructor (`__init__` method), so that they are easy to find and manage. Once again, however, this is just a guideline – anything goes in Python!

In fact, Python is so relaxed that we can create new instance variables, any time, from anywhere that we can access an object, including the main routine. Programmers from other languages would consider this a total violation of the concept of encapsulation – relaxed to the point of madness.

Consider the example in Code Listing 1.10. Remember that the function `vars(object)` returns a description of the instance variables of an object (see section 1.3).

Two instance variables, `name` and `age`, are added in the constructor, as would usually be considered good practice. The variable `married` is added by another method. A further variable, `aeg`, is added in the main routine. This illustrates the point, but also a danger. The programmer has probably mistyped age and created an instance variable by accident! The intended variable `age` now has an outdated value – all sorts of problems are likely to follow.

```
class Person:
    def __init__(self, name, age):
        # usual way of adding instance variables in the constructor
        self.name = name
        self.age = age

    def got_married(self):
        # but instance variables can be added in any method
        self.married = True

#main routine
if __name__ == "__main__":
    p = Person("Eric", 22)      # the constructor adds instance variables name and age
    print("At-1", vars(p))
    p.got_married()             # this method adds instance variable married
    print("At-2", vars(p))
```

```
    p.aeg = 23                  # an instance variable can even be added in the main routine
    print("At-3", vars(p))
```

Code Listing 1.10

Output:

```
At-1 {'age': 22, 'name': 'Eric'}
At-2 {'age': 22, 'married': True, 'name': 'Eric'}
At-3 {'age': 22, 'married': True, 'name': 'Eric', 'aeg': 23}
```

Adding instance variables from outside the class may not seem very desirable but in some contexts becomes crucial. We shall use this facility (with great care and discipline) in the next chapter. As we have already seen (Task 7) it can also be useful to add them within the class in methods other than the constructor.

1.13. Types, references and aliases

Types and references

In Python, all variables / names / identifiers are references to the address in memory where the information that implements the object is stored. We draw these references as arrows from a variable to an object, as shown for example in Section 1.8. We sometimes think of the identifier as the object itself, but often it is essential to remember that it is just a reference to the object.

All variables have a type. In Python a variable gets its type from the data that it refers to, and it can change its type at any time[1]. (The flexibility is nice, but also risky!)

```
x = 2           # x is type int
x = "Hi"        # now x is type str
```

We can inspect the type of a variable (the class it is made from) using the type function.

```
x = 2
print( type(x) )
x = "Hi"
print( type(x) )
```

Output:

```
<class 'int'>
<class 'str'>
```

There are various ways to check if the type of a variable is a particular type:

```
x = 2
print( isinstance(x, int) )     # prints  True
print( type(x) == int )         # prints  True
```

The type of a variable limits the kinds of operations that we can perform on it. If for example we make an object of type `Cake` (Code Listing 1.1), we can't add an integer value to it – that operation doesn't make sense[2]:

[1] Technically speaking, Python is a dynamically typed language whereby objects are bound to their data type at run-time. Since the interpreter does not have the role of checking types, only the name of an object (not its type) needs to be stated.

[2] The interpreter evaluates expressions during run-time execution, giving a run-time error if the operation is impossible. In this example the error arises because the interpreter will not find an `__add__` method requiring an `int` parameter in the `Cake` class – see Appendix.

```
cake1 = Cake()      # Cake object with the identifier cake1 is created
cake1 + 3           # causes an "unsupported operand" error
```

Let's just check the type of `cake1`:

```
print(type(cake1))
```

Output:

```
<class '__main__.Cake'>
```

The result tells is that the type of `cake1` is `Cake` in the module `__main__` (see Appendix). Like any variable, we can assign another value to `cake1`, which may be a reference to an object of a different type (an object made from a different class):

```
cake1 = "chocolate cake"
print(type(cake1))
```

Output:

```
<class 'str'>
```

Like any other variable `cake1` has no "loyalty" to a particular data type (here it has changed from `Cake` to `str`). This is something we have to be careful about when programming in Python – to ensure that variables / names / identifiers are being used correctly.

Aliases

It can be useful to have different ways of referring to an object (like it can be useful to have different email addresses for the same person). Python allows this. A given object can be referred to by multiple variables - in such cases the variables are called **aliases**.

In order to illustrate aliases we will use the built in function `id`, which returns (in most standard versions of Python) the address of an object in memory. This is the surest way of determining the identity of an object. Consider the following program:

```
x = [1, 2]          # variable x refers to a list object, Figure 1.17 (a)
print(id(x))        # which is at this address in memory – e.g. 4347487296
y = x               # y is equal to x, they refer to the same object, they are aliases, Figure 1.17 (b)
print(id(y))        # confirming that y refers to the same address / object as x – e.g. 4347487296
y += [3,4]          # change the state of the object, Figure 1.17 (c)
print(y)            # print the state of the object using variable y
print(x)            # print the state of the object using variable x
```

Code Listing 1.11

In this example `y` and `x` are **aliases**, referring to the same object. The output is:

```
4319128752
4319128752
[1, 2, 3, 4]
[1, 2, 3, 4]
```

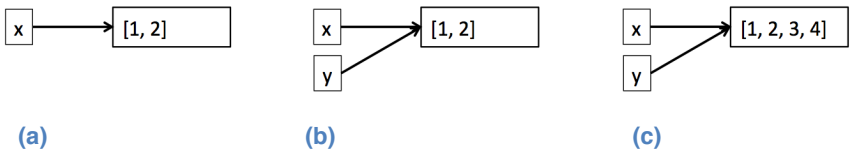


Figure 1.17 (See comments in Code Listing 1.11)

It is important to be aware of the difference **between operations that work on references, like `y = x`, and operations that change the state of the objects referred to, like `y += [3, 4]`.**

Assignment is not the only way of creating aliases. Passing an input / parameter to a function or method does it too. Consider Code Listing 1.12.

```
class Test:
    # a simple class with just a single instance variable set by this constructor
    def __init__(self, xx, y):
        self.y = y

def my_function(xx, tt): # note this is a function, not a method. It is not part of the Test class.
    # function creates local variables xx and tt with values set by input parameters
    print("at start of function local xx is", xx, "reference: ", id(xx),"and  local tt.y is", tt.y)
    print("modify locals!")
    xx = 3
    tt.y = 100
    print("at end of function local xx is", xx, "reference: ",id(xx),"and  local tt.y is", tt.y)
#main routine
if __name__ == "__main__":
    # main routine creates variables x and t (and in the process sets instance variable t.y to 2)
    x = 1
    t = Test(2,5)
    print("at start of program  main  x is", x, "reference: ",id(x),"and  main  t.y is", t.y)
    my_function(x, t)
    print("at end of program  main  x is", x, "reference: ",id(x),"and  main  t.y is", t.y)
```

Code Listing 1.12

Output:

```
at start  of program  main  x is 1 reference:  4296861728 and main  t.y is 5
at start of function local xx is 1 reference:  4296861728 and local tt.y is 5
modify locals!
at end  of function local xx is 3 reference:  4296861792 and local tt.y is 100
at end   of program  main  x is 1 reference:  4296861728 and main  t.y is 100
```

There is a lot going on in this example! The main routine has two variables, `x` (an integer) and `t` (a reference to an object of type `Test`, which in this case has an instance variable `y` set to 2). When we call `my_function(xx, tt)` we create **aliases**, local variables in the function with corresponding names and the same values.

The local variable `xx` is an **alias** for the main routine variable `x` – but only initially. When the function changes the value of `xx` it doesn't change the state of the object currently referenced by xx, but rather sets local `xx` to refer to a new value, the integer 3. We confirmed this using `print(id(x))` and `print(id(xx))` to see the address of the current value. When the function finishes we can see that the value of main routine `x` is the same as it was at the start.

The local variable `tt` is an **alias** for the main routine variable `t`, they both refer to the same object. In the function we use local `tt` to change the state of the object (set `tt.y` to 100). When the function finishes, the value of main routine `t` is the same but the state of the object (value of `t.y`) has changed. The objects instance variable `y` stores a different integer value.

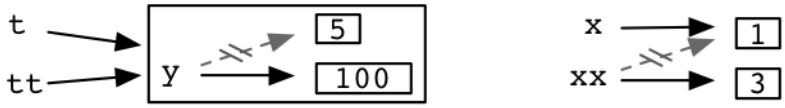


Figure 1.18

Yes, that's all a bit fiddly, but you do need to know about these things[3], or you might never work out what is going on in your programs! Just remember, **it is important to be aware of the difference** between operations that work on variables / references, like `y = x`, and "dot notation" operations that work on (change the state of) the objects referred to.

3 For **mutable** data types, like `list` or the instance of `Test`, instances can change their value / state. For **immutable** data types, like `int`, instances cannot change their value / state (we can't change values we can only replace them with new ones). See the Appendix.

What happens if you set t.y to 100 instead of tt.y?

What happens if you set t.y to 50 after you set tt.y to 100?

What happens to the id if you set xx to 1 instead of 3? (See Note 4)

Task 16

Consider the program in Code Listing 1.13.

```
def my_function(d):
    d += [42]

#main routine
if __name__ == "__main__":
    a = [1, 2]
    b = a
    c = b
    c += [9]
    print(a, b, c)
    b = [0] # this statement creates a separate new list that b refers to, a and c remain unchanged
    print(a, b, c)
    my_function(a)
    print(a, b, c)
```

Code Listing 1.13

When this program runs, what is the output? Create a diagram (similar to Figure 1.17) that shows the variables and the state of the list objects that they refer to at the point of each of the three print statements in the program.

Summary Notes

All instances of the same class start with the same instance variables and the same methods. The methods represent the **behaviours** that each class can perform. Behaviours are always the same for each instance of the class but the values stored in the instance variables can be different for each object. The objects are said to have a different **state**.

The instance variables and methods are called the **attributes** of the class.

A class must have a header. All other attributes are optional.

Comments: Each class should begin with a docstring describing the author, date and purpose. Each method should begin with a docstring describing its purpose.

Scope: Any variables which are declared within a method and not prefixed by `self`, including a method's parameter list, are local variables. **Local variables** can only be accessed within the method in which they are declared. **Instance variables** are those prefixed with `self`. All methods in the class can access the instance variables provided they remember to prefix with "`self.`". The main routine, and other classes, can access these instance variables using dot notation (`name_of_instance.name_of_variable`).

Global variables declared in the `main` routine in the same file are also available throughout the class but should probably not be used.

1.14. Examples

Task 17 - Egg Order Program

Using the OOP concepts we have covered in this chapter can you rewrite the Egg Order program, removing the need to maintain parallel arrays for customer names and their order quantities? What belongs in the main routine? What should be in a class? What should the class (or indeed classes) even be called? Jot down some ideas and if you are feeling confident have a go at coding your solution.

To make OO really pull its weight and work for us we suggest a `DailyTotalOrder` class that manages a list of individual `EggOrders` for the day.

Below is a suggested Object diagram.

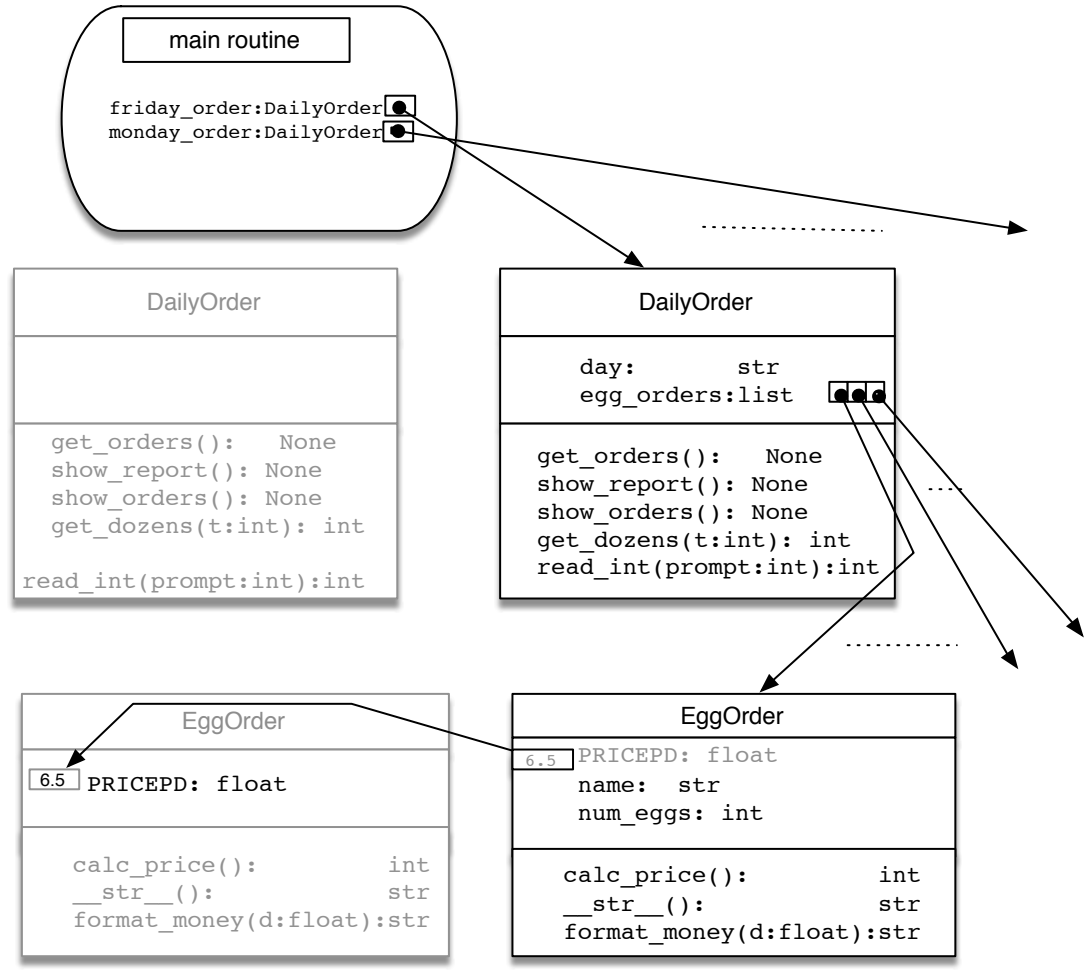


Figure 1.19

Task 18 - Quiz program

The task is to run and understand the following program!

The program presents the user with a quiz consisting of (at this stage) two multi-choice questions with four possible answers each. Every time the program runs the possible answers are presented with the correct answer at a different random position in the sequence.

The user is asked to enter an integer representing their selected answer. The input is checked to make sure that it is a int, and that it is within the range of the number of answers (in this version of the program 0 .. 3). For a valid input the user gets one message for a correct guess, and another for incorrect guesses. After two questions the program ends.

Make sure that you understand how the program works. Extend the quiz by adding two more complete questions and answers. Think about possible extensions to the program, such as keeping a count of the number of questions that the user gets right first time, or calculating the average number of guesses per question, and so on.

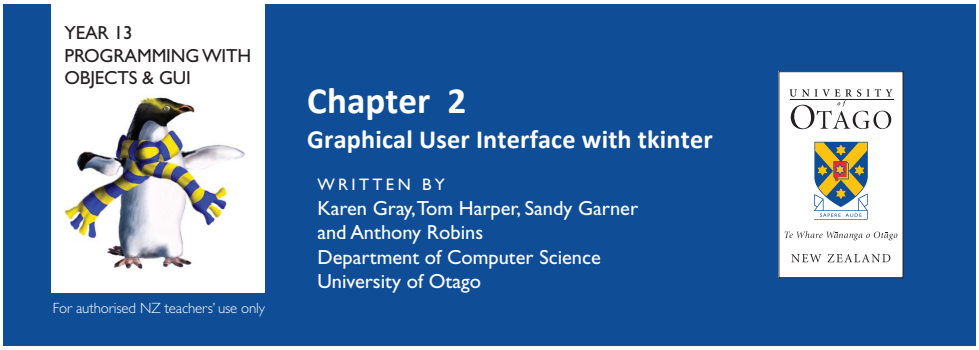
This is a simple text-based program. In future chapters it will be extended with a graphical user interface, and to read any number of questions and answers from a file. It is worth understanding well, because we will be getting back to it!

The code below is all one program which should be placed in a single .py file and run.

```
import random

class Question:
    """ Authors: Gray, Harper, Garner, Robins;  Date: 2 December 2012
        Purpose: This class supports the class Quiz.  Instances of this class represent individual
        questions in the quiz, and possible answers, with the correct answer at a random place in
        the list of possible answers (dummies).
    """

    def __init__(self, question, answer, dummies):
        """ Constructor for instances of Question, sets up the instance variables representing
            the question, and calls self.set_answer to set up the instance variable representing
            the possible answers (including the correct answer).
        """
        self.question = question
        self.answer = answer
        self.dummies = dummies
        self.set_answers()

    def set_answers(self):
        """ Inserts correct answer at random place in the list of possible answers (dummies). """
        self.answers = self.dummies
        self.answers.insert(random.randrange(len(self.dummies) + 1), self.answer)
```

```
class Quiz:
    """ Authors: Gray, Harper, Garner, Robins;  Date: 2 December 2012
    Purpose: This class manages a (text only) multi-choice quiz.  It will be  extended in future versions with a
    graphical user interface, and access to  files of multiple questions and answers.  This class is supported by the
     class  Question.  Instances of Question represent the individual questions (and  answers of the quiz).  In this
     version of the program there are only two questions.  """

    def __init__(self):
        """ Creates a list containing two Question objects by passing the Question constructor
            - a string representing the question
            - a string representing the correct answer
            - a list of strings representing the dummy answers
        """
        self.questions = [Question("What is the capital of Mongolia?",
                  "Ulan Bator",
                  ["Vladivostok", "Astana","Lhasa"]),
                  Question("Who wrote 'The Picture of Dorian Gray?",
                  "Oscar Wilde",
                  ["George Bernard Shaw", "Evelyn Waugh", "Somerset Maugham"])]

    def take_quiz(self):
        """ This method is the main driver for the quiz. It loops through questions,
            presents each one and its possible answers. It then calls the process_answer method
            passing in a reference to the current question.
        """
        for q in self.questions:      # for each question object in the list of questions
            print( q.question )              # print the question
            for i in range(len(q.answers)):       # print the possible answers
                print("\t" + str(i ) + "\t" + q.answers[i])
            print()
            self.process_answer(q)   # get answer from user and give appropriate response

    def process_answer(self, q):
        """ This method reads in the user's answer (should be an int in the range 0 to the number
            of possible answers).  It gives an appropriate response to the user's answer.
            The while loop continues until the user inputs an int that is in range. Once they do
            their answer is checked against the correct one to see if it matches.
            Remember input comes in as a string and must be converted to int.
        """
        user_answer = -1
        # keeps looping until user's input is an int in range
        while not 0 <= user_answer < len(q.answers):
            a = input("Please type the number of your answer here: ")
            try:
                user_answer = int(a)   # if input is not an int we go to the except clause
                if not 0 <= user_answer < len(q.answers):
                    # input is not in range no further action is taken (the while loop will repeat)
                    print("\nThat was out of range\n")
                elif user_answer == q.answers.index(q.answer):
                    # input is equal to the index of the correct answer (the while loop will end)
                    print("\nWell Done!!\n")
                else:
                    # input not equal to the index of the correct answer (the while loop will end)
                    print("\nIncorrect, the answer is " + q.answer + "\n")
            except ValueError:
                # if user's input is not an int this executes (anticipating errors is good design!)
                print("\nThat was not a sensible input. Integers only please.\n")

#main routine
if __name__ == "__main__":
    text_quiz = Quiz()
    text_quiz.take_quiz()
```

Code Listing 1.14



2.1. Introduction to Graphical User Interfaces

Let's face it, text is dull. Graphical interfaces are expected in the modern world. This chapter introduces a sufficient range of classes to be able to produce useful and interesting Graphical User Interface (GUI) applications.

Compared to the typical "imperative" programs that we have written so far, programming a GUI is a different kind of design, called "event driven programming". In an imperative program the programmer sets out the complete sequence of steps that the program will take. In an event driven program the programmer sets up a specification of how to respond to what the user does. Actions that the user takes in the GUI, like clicking a button or moving a slider, are called **events**.

We can see this design difference very clearly in the main routine. For our programs so far the main routine specified a complete sequence of actions. For our GUI programs the main routine can be much simpler. It brings up a GUI window, it will usually create an object which specifies the GUI to be built in the window. But it doesn't specify the sequence of actions that follows – what happens next is up to the user!

There are **four common requirements** involved in setting up any GUI.

1) You must specify how you want the GUI to *look* (write code that determines what the user will see on the screen).
2) You must specify what you want the GUI to *do* (write code that accomplishes the tasks of the program).
3) You must associate the *looking* with the *doing* (languages tend to have different ways of doing that).
4) You must initiate event handling so that the program actively waits for events from the user (this is usually pretty much automated for you).

Look out for these requirements being fulfilled in code as you progress through the chapter. How we meet these requirements differs from programming language to language. Python provides a range of optional resources (libraries / packages of classes) for making GUIs. In this book we will use the **tkinter** package, which comes already built in to most standard Python distributions. When we use tkinter (or GUI capabilities in most current programming languages) then we need to be aware that our programs are suddenly supported by lots of **automatic** processes – things happen behind the scenes that we can't control, but we can set up our programs to use.

By convention objects made from classes in tkinter are called **widgets**. There are widget objects of various kinds. Some represent windows that we can see on screen, some represent containers for holding and organising other widgets, some represent GUI elements like buttons, sliders, text fields, graphical shapes and so on.

When the user interacts with a GUI element / widget it is called an **event**. Widgets can be **bound** to methods or functions, which are then known as **callback functions** or just **callbacks**. (There are two kinds of binding, **command binding** and **event *binding***.) If a widget triggers an event the bound callback function(s) is called. We put the code, which implements the appropriate action to take, in these callback functions.

The overhead of GUI programs is that there is a fair amount of set-up code. Constructors are likely to be much longer than for text-based programs. Objects are needed for each GUI element and for windows and containers to organise them. Each object may need to be set-up, coloured, sized, grouped, given an initial value, and so on. Callback functions must be written and bound to respond to events. Although it is a different style of programming, planning is just as important as for any other program.

2.2. Introduction to tkinter

Tk is a basic graphics tool kit used in many programming languages. It is open source and can be used on many different platforms. In Python tkinter stands for **t**ool **k**it **inter**face, literally an extra layer of functionality over the basic tool kit. Tkinter or tkinter is included by default with most installations of Python. Tkinter is for Python 2.7 while tkinter is for Python 3, so we will be using tkinter. It offers us everything we need for developing quite sophisticated GUI programs. Python.org has links to tkinter reference material for more details. Resources that we recommend include:

http://wiki.python.org/moin/TkInter

http://www.tkdocs.com

http://www.ferg.org/thinking_in_tkinter/all_programs.html

http://infohost.nmt.edu/tcc/help/pubs/tkinter/index.html#intro

Our GUI programs using tkinter will all have the form of the example in Code Listing 2.1 below. In the main routine we create an instance of the class Tk, which is displayed on screen as a window. This instance is called `root` by convention but it would work as well if we called it `fred`. You should not call it `fred`. We would usually also construct another object (or objects) specifying the details of the GUI that will be built in the window. At the end of the main routine we call `root.mainloop()`. This starts up the automatic processes that implement event handling. The program now runs, waiting for user input (events), until the user quits the GUI or closes the root window (or kills the process from the host operating system).

The program in Code Listing 2.1 is the simplest possible example. It brings up the empty root window.

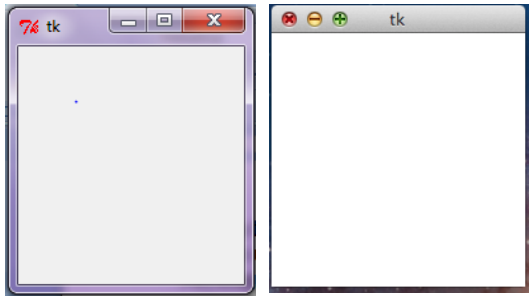
```
from tkinter import*
# main routine
root = Tk()      # root refers to a new instance of class Tk which is shown on screen as a window
# code in the main routine usually makes an instance of a class that defines a GUI in the root window
root.mainloop()  # starts the method / processes that handle events from the root window and GUI
```

Code Listing 2.1

Copy the code into a Python file and run it as usual.

The root window will appear on screen. Examples for two different operating systems are shown to the right.

Note: The root window will appear at the top left of the screen and may be hidden behind other windows (this is especially common on the Mac). So you might have to move other windows to find it!



Tk window on Windows OS **on Mac OS**

The `mainloop` method starts the automatic processes that listen for GUI events (Requirement 4 above). In the main routine we usually make an instance of a class that builds the GUI. This GUI class (or classes) makes and arranges the necessary widgets (Requirement 1), binds widgets to callbacks (Requirement 3), and fills in the body of the callbacks with the desired behaviour (Requirement 2).

For the first few sections of this chapter we'll just develop some background knowledge.

An important note on quitting tkinter programs

We might expect to be able to choose quit from the application menu, but (if you're writing in IDLE) that isn't reliable and programs hang (because IDLE is itself a tkinter program and sorting out the confusion seems to be "work in progress"). The simplest and most reliable way to quit these programs is to click the red close "button" in the top border of the window. It's crude (we can't implement any tidy up / shut down behaviour), but it's efficient and reliable, so we can live with it.

There are other ways of quitting a tkinter program. We could have a quit button in the GUI bound to a callback which executes various quit or destroy methods, but that takes work to set up.

For the most part the GUI programs in this book assume that we quit the program as just described. We will however introduce a quit button for use with keyboard events in section 2.18.

With the program in Code Listing 2.1 the Tk window appears at the default size, in the default position on the screen. There are some useful changes we can make to the window relatively easily. The title on its title bar can be set using the `title` method. The size of the window can be set using the `geometry` method.

```
from tkinter import *
# main routine
root = Tk()
root.title("Large and High")
root.geometry("600x600+0+0")
# GUI code goes here
root.mainloop()
```

Code Listing 2.2

The string passed into the `geometry` method takes the form `widthxheight+x+y` where width is the width of the window, height is the height of the window, x is the distance of the window from the left hand side of the screen and y is the distance of the window from the top of the screen. All are measured in pixels. **Note**: no spaces are allowed in this string.

Try this exercise 1: Make a tkinter window and give it a title. Make it take up the lower right quarter of your screen.

A final note on importing the tkinter classes: when we `import *` the asterisk is a wildcard, which matches any name, i.e. we import every class. It is not normally considered good practice to import everything from a package in this way because it can lead to naming conflicts, e.g. if we give a class we've defined the same name as an imported one the interpreter has no way of knowing which one we mean. The safer but much less convenient way is:

```
import tkinter
```

but then everything from tkinter that we want to use would have to be prefixed e.g.

```
root = tkinter.Tk()
```

GUI programs rely heavily on tkinter objects so this would make for some very long and tedious lines of code. For this reason it is common practice to `import *` when using tkinter. By keeping our code within properly formed classes and being careful about what we name things, we should reduce any risk.

2.3. Setting Fonts and Colours in tkinter

These skills will be useful in the following section on widgets.

Fonts in tkinter

Fonts in tkinter are set with a **tuple**. A tuple is a built-in type in Python. Tuples are very similar to lists except they are created using parentheses instead of square brackets and they are immutable (can not be altered).

List Example

```
my_list = [10, 20, 30]
print(my_list[0]) #prints 10
my_list[3] = 40
print(my_list) #prints [10, 20, 40]
```

Tuple Example

```
my_tuple = (10, 20, 30)
print(my_tuple[0]) #prints 10
my_tuple[3] = 40 #TypeError: 'tuple' object does
    # not support item assignment i.e it can't
    # be altered because it is immutable
```

The `font` tuple consists of two or three strings in parentheses. The first is the name of the font family e.g. "Courier" or "Times". The second is the size in points (pts). The third is optional, and consists of a series of style options separated by a space. The options available are **bold**, underline, *italic* and ~~strikethrough~~. The following three statements will each successfully set a tkinter font.

```
font = ("Comic Sans MS", "14", "bold")
font = ("Times", "24", "bold italic")
font = ("Times", "24")
```

Note:- Fonts are Operating System specific. If the font family specified does not exist in the Operating System being used, some other font will be assigned instead.

Colours in tkinter

In tkinter, colours are represented as strings.

There are a multitude of predetermined named colours available in tkinter. All the usual prime colours are available as well as several hundred more. A small selection spanning the main colour bands is given below.

snow	gainsboro	linen	cornsilk	ivory	seashell	honeydew
mint cream	azure	slate gray	gray	midnight blue	navy	cornflower
sky blue	light blue	turquoise	aquamarine	dark green	sea green	pale green
green	chartreuse	olive drab	goldenrod	sienna	beige	tan
chocolate	firebrick	brown	rosy brown	orange	coral	tomato
pink	light pink	violet	plum	orchid	purple	thistle

Other, unnamed, colours can be created using RGB format. RGB stands for Red Green Blue. Each of these is a number (typically between 0 and 255). In Python these numbers are represented in hexadecimal format prefixed by a hash (`#`) character e.g. `#ff1a0b`.

This is the same format as is commonly used in graphics programs and web design, so any colour information from these programs can be used in tkinter to create a similar colour.

Note: A hexadecimal colour representation uses base 16. The number 10 is represented by 'a', 11 by 'b' … 15 by 'f'. The base 10 number range 0 to 255 will be covered by two hexadecimal digits (one byte) 00 to ff. The hex string `#ff1a0b` which represents a reddish colour is made up of red = ff `(255)` blue = 1a `(26,` being 1 * 16 + 10`)` and green = 0b `(11)`.

Things to be aware of:

- Each colour string must be preceded by a hash symbol, #
- tkinter accepts either 3 digits, 6 digits or 12 digits so the colour red could be represented as "#f00", "#ff0000 or #ffff00000000. (12 digits for super-hero vision only)
- Black is #000000, white is #ffffff (or the 3 or 12 digit equivalents)

Code Listing 2.3 offers an easy way to generate a string representing a random colour.

```
from random import randrange
hex_chars = ['a','b','c','d','e','f','1','2','3','4','5','6','7','8','9','0']
rgbString = "#"
for i in range(6): # or could be 3 or 12
    rgbString += hex_chars[randrange(len(hex_chars))]
```

Code Listing 2.3

2.4. Introduction to Widgets

Widgets are the objects that make up a graphical user interface in tkinter. In this chapter we will introduce the widget classes listed below *. We shall look first at things they have in common before looking at examples of each one briefly. You will see that the basic pattern for using any widget is the same.

Widgets for displaying and reading in text	Widgets for making things happen	Container widget	Drawing widget
Label Entry ScrolledText	Button Checkbutton Radiobutton Scale	Frame	Canvas

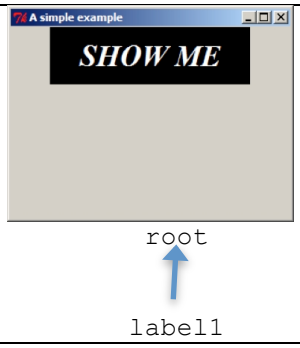
* *This is not a complete list of all widgets available! We have left out quite a few to preserve sanity.*

Widget Basics

- Creating a widget object involves calling the widget's constructor.
- Each widget constructor comes with a set of optional arguments that can be set on creation of the widget object.
- Most widget constructor options can be reset after a widget is created using the `configure` method.
- There are some basic options that all widgets have, e.g. width and height. Others are specific to each widget or to different sorts of widgets. With sometimes over 30 various options available to a widget, we shall just pick out the most useful.
- Most widgets have methods that can be called on them. We will focus on the core methods.
- **The first (compulsory) argument to any Widget constructor is the name of its parent widget.**

Code Listing 2.4 creates a `Label` widget called `label1`, which is added directly to the tkinter window (root) and is black with white text that says "SHOW ME". The font is set to 24pt, bold, italic Times.

```
from tkinter import *
#main routine

root = Tk()
root.title("A simple example")
root.geometry("300x200+200+200")
label1 = Label(root, bg = "black", fg = "white", padx = 30, pady = 10,
                  text = "SHOW ME", font = ("Times", "24", "bold italic"))
label1.pack()
root.mainloop()
```



Code Listing 2.4

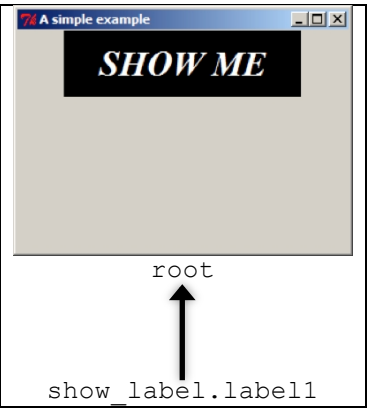
Important: `label1.pack()` arranges the `label1` object to fit on its parent container, in this case the Tk (`root`) widget. Without this line of code nothing would actually show in the window.

The parent/child relationship

The Tk root window is itself a widget. We can add widgets directly to the root widget or we can add them to other container widgets such as frames, which you will meet shortly. For example we might add a label and a button to a frame, and then add the frame to the root. The name of the parent (or containing) widget that you are adding to must be the first argument to the child widget's constructor. If you don't want an empty window something had better be added to the root widget!

The program in Code Listing 2.4 is well and good if our programs stay simple but (gasp) they're not going to. It is better practice to place your GUI code in a class and then add an instance of the class to the root. This is demonstrated in Code Listing 2.5. Let's start as we mean to go on.

```
from tkinter import *

class ShowExampleGUI:
        def __init__(self, parent):
          self.label1 = Label(parent, bg = "black", fg = "white",
                padx = 30, pady = 10, text = "SHOW ME",
                font = ("Times", "24", "bold italic"))
          self.label1.pack()
#main routine
root = Tk()
root.title("A simple example")
root.geometry("300x200+200+200")
show_label = ShowExampleGUI(root)
root.mainloop()
```



Code Listing 2.5

Note that the Label constructor now starts with `parent`, but is still referring to the root widget. Once again, we could have called `parent` something else, like fred, but we didn't and nor should you. Remember, the `__init__` method is called just once, on creation of the class instance. Generally all set-up code for GUI widgets belongs in the `__init__` method.

Some common widget options

width : The width of the widget. Oh, if only it was that simple. Some widgets measure their width straightforwardly in pixels. Others measure their width in "text units" if they contain text, or pixels if they contain an image. If using text units, 1 unit is approximately the width of 1 character in that font. (If you've used html and css then a text unit is like an em.)

height: as for width

bg: The widget's background colour. Refer to section 2.3 "Setting Fonts and Colours in tkinter".

fg: The widget's foreground colour. This is the colour any text will be.

text: Available where sensible.

underline: The underline specifies the index of the character to be underlined. It is usually used to indicate a corresponding key binding (discussed later in the chapter)

font: Available where text is. Refer to section 2.3 "Setting Fonts and Colours in tkinter".

pady and **padx** as widget options: The space between the widget's contents and the widget's edge (*if you've done web development it is a bit like padding in CSS).* Measured in pixels; padx refers to horizontal space, pady to vertical space.

anchor as a widget option: If the size of the widget is bigger than the content requires then anchor dictates how the content is aligned. Think of compass points with North at the top, East to the right etc. The anchor option can be set to N, NE, E, SE, S, SW, W, NW or CENTER

relief determines the existence and style of a widget's border. The default is flat. Other options include SUNKEN, RAISED, GROOVED and RIDGE.

bd determines the width of any border

takefocus determines whether the widget can be reached by tabbing to it (as an alternative to clicking it with a mouse). We will discuss tab traversal, also called focus traversal, in greater depth in the final section of this chapter.

Some common pack options

When adding a widget to its container, there are some options to do with layout which can be used with the `pack` method.

anchor while packing a widget to its container: If the space available to a widget is bigger than the widget, anchor dictates how the widget is aligned. It takes values N, NE, E, SE, S, SW, W, NW and CENTER (the default). (See Code Listing 2.7.)

pady and **padx** while adding a widget to its container: the space between the widget and the edge of the container or any neighbouring widgets (*if you've done web development it is a bit like margin in CSS).* Measured in pixels. (See Code Listing 2.8, 2.9, 2.10, 2.11.)

side The side of the container the widget will be placed against. May be LEFT, RIGHT, TOP or BOTTOM. The default is TOP. (See Code Listing 2.16, 2.19.)

fill determines whether the widget will fill its parent horizontally or vertically or completely. Takes X, Y or BOTH. (See Code Listing 2.16.)

expand determines whether a widget will expand to fill any available gaps when used with fill. Takes True or False. (See Code Listing 2.16.)

Useful pack methods

pack_forget: When called on a widget the widget will be removed from its parent. (Not used in this workbook)

Changing widget options

If we wanted to change any of the options of `label1` of Code Listing 2.5 at a later point we could assign a value to the option directly:

```
self.label1 ['fg'] = "red"
self.label1 ['text'] = "hide"
```

or we could use the `configure` method of the `Label` class, which can change any number of attributes at once e.g.

```
self.label1.configure(text = "Hide me.", fg = "red")
```

Code Listing 2.6 shows the `change_text` method being used to change the text on the label. The method is called from the main routine some time **after** the creation of the GUI object, not from the `__init__` method, which is called **on** creation of the object.

"""Chapter 2, Code Listing 2.6"""

```
from tkinter import *

class ShowExampleGUI:
                    def __init__(self, parent):
                        self.label1 = Label(parent, bg = "black", fg = "white",
                                padx = 30, pady = 10, text = "SHOW ME",
                                font = ("Times", "24", "bold italic"))
                        self.label1.pack()

                    def change_text(self, newText):
                        self.label1.configure(text = newText)
```



```
#main routine
root = Tk()
root.title("Configure example")
root.geometry("300x200+200+200")

show_label = ShowExampleGUI(root)
show_label.change_text("Hide Me")
root.mainloop()
```

Code Listing 2.6

Note:- The root container also has a configure method which can be used to set its options e.g.

```
root.configure( bg = "azure")
```

If we want to access any of the option values of a widget we can e.g.

```
self.label1['text']
self.label1['bg']
```

If the label was in a list called labels, the syntax for accessing the text on the first element would be

```
self.labels[0]['text']
```

Be aware, however, that option values are not always returned as the type you might expect them to be:

```
print(type(some_label['text']))  produces: <class 'str'>                    as expected
print(type(some_label['padx']))  produces: <class '_tkinter.Tcl_Obj'> NOT int
print(type(some_label['font']))  produces <class 'str'>                     NOT tuple
```

Try this exercise 2

Make a `Label` widget object with some text on it and display it on a tkinter window.

Try setting some of the Label widget's options using the `__init__` method.

Try changing some of the Label widget's options using `configure`.

Try changing some of the Label widget's options using direct access.

Make a `change_colour` method which creates a random colour string. Use this method to set the text colour on your label.

2.5. Images in tkinter

Naturally we may want to use images in our graphical user interfaces. The `tkinter` module offers the `PhotoImage` class. It allows us to place images on widgets instead of, or as well as, text.

```
self.smiley_img = PhotoImage(file = "smiley.gif")
self.my_label = Label(parent, image = self.smiley_img)
```

This code creates a `PhotoImage` instance from a file called `smiley.gif` and puts it on a Label widget. If `smiley.gif` is not in the same folder as your code, you will need to provide the correct path e.g. `file = "/images/smiley.gif"`

Important notes to refer back to when using images later:

1. It is crucial to keep a reference to your image that is not just a local variable. An annoying quirk of tkinter is that once `__init__` has finished executing and we enter the mainloop our widgets stay put but any image that was just local in `__init__` disappears *even though it was set as a widget option and should be displayed by the widget*! We have two choices*.

- Since we are using objects and writing code in our own user-defined GUI class it might make sense for the image to be an attribute of our object as shown above. In this case we have called it `smiley_img` to distinguish it from any other images which might be in our class.

```
self.smiley_img = PhotoImage(file = "smiley.gif")
```


- An alternative would be to make the image an attribute of the widget itself. We have used `image`, rather than `smiley_img`, as the identifier for this alternative because it isn't necessary to distinguish which image. Each Label can only have one image. If all Label images have the same name, they can be more easily accessed in a loop from a list e.g. `my_Labels[i].image`.

  ```
  self.my_label.image = PhotoImage(file = "smiley.gif")
  ```

  This is possible because Python lets you tack on instance attributes after an instance is created and from outside the class (see Chapter 1 Section 1.12).

If your image mysteriously vanishes from your widget, choosing one of these options will most likely correct the situation. Which option will depend on the circumstances of your program.

* There is a third choice - make the image global i.e. declare it at the same level as the declaration of root. This becomes very messy if there are a lot of them and is NOT good OO programming.

2. PhotoImage works with a limited number of formats. We will work with images that are in either GIF format or PPM format.

GIF (Graphics Interchange Format) images use a colour palette of only 256 colours. Consequently, they have very small file sizes but are rarely suitable for photographs.

PPM (Portable Pixel Map) images offer excellent quality but are really quite inefficient. File sizes will be much greater than say a JPEG of similar perceivable quality.

If you have images you wish to use in a program that are not in GIF or PPM format, they can be easily converted online at

http://www.sciweavers.org/free-online-image-converter or a multitude of other sites.

There are several approaches we might take if we needed to work with other image formats. One would be to install and import PIL (Python Imaging Library) but this is beyond the scope of this course. We will simply work with PhotoImage and the file formats available.

2.6. Variable Classes

Refer back to this section when using IntVar and StringVar.

Certain kinds of widgets need to be able to share their essential data with us. When a user engages with a widget they are putting information into our program. This information is likely to be needed later in the program. The information can be stored in and extracted from widgets using the special tkinter variable classes. For example, a check box widget will need to tell us that it has been selected.

The tkinter module provides the special Variable classes: BooleanVar, DoubleVar, IntVar and StringVar. Unlike ordinary Python variables, instances of these classes are designed to work with tkinter widgets.

The instance of the Variable class is associated with one or more widgets using a widget option (`variable` or `textvariable`). When the user interacts with the widget, they change the value stored in the variable. If other widgets are referencing that variable instance then they are automatically updated with the change in value.

Our program can also access that value using the **`get`** method provided in whichever Variable class was used.

Variable classes are used with the Entry widget in Code Listing 2.9, and for the on / off value for Checkbutton and Radiobutton widgets in Code Listings 2.12 - 2.15.

Main points:

- we create instances of them by calling the appropriate constructor
- we set the value of them using the `set` method

- providing we are within scope we can access the value of the instance by calling the `get` method on it
- As with `PhotoImage` (discussed earlier) we can either make each Variable instance an attribute of our object using self, or an attribute of the widget itself.
- Each Variable instance has a string representation. If you directly print the Variable you will see PY_VAR plus an index number.

Let's look at the creation, setting and accessing of Variable instances just on their own for now.

```
from tkinter import *
#main routine
root = Tk()

my_int_var = IntVar()
my_int_var.set(23)

print (my_int_var.get()) # prints 23
print(my_int_var)  # prints PY_VAR0
my_str_var = StringVar()
my_str_var.set('hello')

print (my_str_var.get()) # prints hello
print(my_str_var)# prints PY_VAR1
```

The widget examples that follow show how the Variable instances can be used in conjunction with them.

Of the widgets we are using in this workbook, Variable instances are only **required** for Checkbuttons and Radiobuttons to work. They are optional but often useful with a variety of other widgets.

2.7. Widgets for Displaying and Reading in text

The Label Widget

The print statement cannot put text on a tkinter window. Instead, the Label widget can be used for displaying text. It can also display an image. It is (as the name suggests) often used to label other widgets. You have already met the Label widget.

Useful Label Widget Options, on top of the common options listed in Section 2.4:

wraplength: The wraplength option can be used to say when a label's text should move to a new line. It is measured in pixels. (See label4, Code Listing 2.7.)

image: Assigned an instance of PhotoImage. If the Label's width or height is less than that required by the image the image will be cropped. (See label3, label5, Code Listing 2.7.)

compound: Required if the Label is to display both text and an image. It takes values TOP, BOTTOM, LEFT, RIGHT and CENTER. CENTER will place text over the image, TOP will place the image above the text and so on. (See label5, Code Listing 2.7.)

textvariable: This option can be assigned an instance of StringVar. If the value of the variable is changed the label will be updated to reflect the new value. (See Code Listing 2.9.)

Points to note:

When displaying text, width and height are measured in text units (to be safe allow 1 per character). When displaying an image, width and height are measured in pixels.

In Code Listing 2.7, `__init__` sets everything up. We then enter the `mainloop`. At this point our image would have been lost if we had only declared it as a local variable inside `__init__`. In this case we've made it an attribute of LabelsExampleGUI. This makes sense since it is being reused within the LabelsExampleGUI class.

The LabelsExampleGUI code is just concerned with creating and displaying the label widgets. Since there is no method other than `__init__` it wasn't necessary to make the widgets themselves attributes of LabelsExampleGUI for this or many of the following introductory widget examples.

```
"""Chapter 2, Code Listing 2.7, Label Widget Examples"""
from tkinter import *

class LabelsExampleGUI:

    def __init__(self, parent):
        self.smiley_img = PhotoImage(file = "smiley.gif")

        label1 = Label(parent, width = 20, anchor = W, text = "Label 1",
                       bg = "white")   # anchor applies to text placement in label
        label1.pack()

        label2 = Label(parent, width = 10, text = "Label 2", relief = RAISED)
        label2.pack()

        label3 = Label(parent, image = self.smiley_img, padx = 10, pady = 10)
        label3.pack(anchor = E)  # anchor applies to label placement in container (window)

        label4 = Label(parent, width = 20, wraplength = 150, bg = "white",
               text = "My really, really, really, . . . long label text")
        label4.pack()
        label5 = Label(parent, image = self.smiley_img, text = "Say cheese!",
                    font = ("Comic Sans MS", "14", "italic"),  compound = TOP,
                    padx = 10, pady = 10, relief = RIDGE)
        label5.pack(anchor = W)

        empty_label = Label(parent, width = 20, height = 5, relief = SUNKEN)
        empty_label.pack(padx = 2, pady = 2)

#main routine
if __name__=="__main__":
    root = Tk()
    labels = LabelsExampleGUI (root)
    root.mainloop()
```

Code Listing 2.7

The Entry Widget

The entry widget allows a single line of text to be typed in by the user.

Useful Options:

textvariable: Is assigned an instance of StringVar (see Section 2.6, Variable Classes). Any other widget referencing the same instance will be automatically updated to reflect whatever the user types in (see Code Listing 2.9).

Entry does not have an anchor option.

Useful methods:

get: To access the text that has been entered into an Entry widget use the `get` method, e.g. `e1.get()`. This always returns a string. (See Code Listing 2.11 for an example.)

focus Sets the program's focus to this widget. The focus method can actually be called on most widgets but it is particularly useful for Entry widgets. This is often an important consideration to enhance a program's usability (see Code Listing 2.9).

insert: Within your program code, text can be inserted into an entry widget using the insert method. The first argument is the string position at which you wish to insert the new text (see the code in bold in Code Listing 2.8).

delete: If you wish to clear the text on an entry widget you can use the delete method. The first argument is the first index of the portion of the string you wish to delete. The

second argument is optional and is the last index of the portion you wish to delete. So for an entry widget called `e1`:

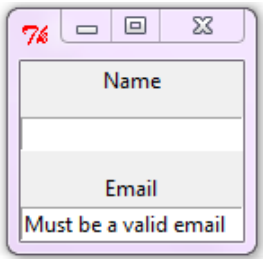
```
e1.delete(2)  # would delete the third character.
e1.delete(2, 5)  # would delete the 3rd, 4th and 5th characters
e1.delete(0, END)  # would clear it completely
```

To replace the text on an entry widget, `delete` it all first and then `insert` the new string at position 0.

Note: Because a user can type ANYTHING into an Entry widget, if you want to use their input for anything meaningful it is crucial to make sure it isn't going to crash your program. Remember that we can use try/except to anticipate any likely errors. Examples are seen in Code Listings 2.11 and 2.23.

Again, remember that in these introductory examples, the variable and widget scope is just local to `__init__`. To be used outside `__init__` you would need to add `self` to make them instance variables (as in Code Listing 2.10).

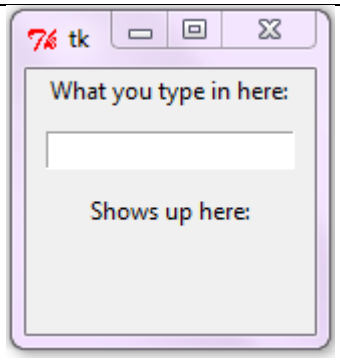
```
"""Chapter 2, Code Listing 2.8, Entry Widget"""
from tkinter import *

class EntriesExampleGUI1:

    def __init__(self, parent):
        label1 = Label(parent, text = "Name")
        label1.pack()
        e1 = Entry(parent)
        e1.pack(pady = 10)

        label2 = Label(parent, text = "Email")
        label2.pack()
        e2 = Entry(parent)
        e2.insert(0, "Must be a valid email")  # inserts initial string at position 0
        e2.pack()

#main routine
if __name__=="__main__":
    root = Tk()
    entry_ex = EntriesExampleGUI1(root)
    root.mainloop()
```



Code Listing 2.8

Our second example demonstrates the use of an instance of `StringVar` in conjunction with the `textvariable` option for a Label and an Entry widget.

```
"""Chapter 2, Code Listing 2.9, Entry Widget with StringVar"""
from tkinter import *

class EntriesExampleGUI2:

  def __init__(self, parent):
    entry_var = StringVar()  # note StringVar declaration
    entry_label = Label(parent, text = "What you type in here:")
    entry_label.focus()  # the user will not need to click in the field before typing
    entry_label.pack()
    e1 = Entry(parent, textvariable = entry_var)  # StringVar used here
    e1.pack(padx = 2, pady = 10)

    output_label = Label(parent, text = "Shows up here:")
    output_label.pack()

    # and StringVar used again here
    output = Label(parent, height = 3, textvariable = entry_var)
    output.pack()

#main routine
if __name__=="__main__":
  root = Tk()
  entry_ex = EntriesExampleGUI2(root)
  root.mainloop()
```



Code Listing 2.9

The StringVar instance `entry_var` is shared here by the Entry and the Label widget (they both specify it as their `textvariable` option). When entry_var is updated by typing in the Entry field, the Label's text is simultaneously updated. The `textvariable` option is used when the text on a Label etc. is expected to change.

2.8. Events and Binding

At this point we need to get back to some theory briefly. Now might be a good time to revise the introduction to this Chapter. Remember the four requirements for setting up a Graphical User Interface:

1) You must specify how you want the GUI to *look* (write code that determines what the user will see on the screen).
2) You must specify what you want the GUI to *do* (write code that accomplishes the tasks of the program).
3) You must associate the *looking* with the *doing* (languages tend to have different ways of doing that).
4) You must initiate event handling so that the program actively waits for events from the user (this is usually pretty much automated for you).

So far we have been dealing with setting up the look of a GUI (Requirement 1). Now it's time to start getting the GUI to do something, so this section marks an important step-up in our programs, at last we are on to events!

Requirement 4 is handled for us by the `mainloop` method. The programs we have been writing in this chapter have all been event-ready because of the statement `root.mainloop()` in the main routine.

The actions required as a response to an event should be described in methods (Requirement 2). There may be a different method for each event, or several events may use the same method. Methods are written in the usual way, as a series of statements to achieve a purpose. When a method is connected to an event, it is known as a callback method, often referred to just as a callback. (If the callback is in a class, it is a callback method. If not, it is a callback function)

These methods are registered as responses to events by a process called binding (Requirement 3). The goal of Python event handling is to **bind** the event to the **callback** (specified callback function or method). This may be done by **command binding** or **event binding**.

Command Binding (GUI Events)

Some **GUI events** are so commonly expected as part of a widget's purpose that the widget can be instructed to handle them via built-in **command binding**. This binds a widget to its relevant callback method. A typical example is where the user clicks on a GUI button, and the Button widget calls the relevant callback method. The event in this case is a button click, which causes a **GUI event**. Using this approach no information is sent to the callback – it is run but it doesn't get any input arguments. (One disadvantage is that if several widgets are bound to the same callback, the callback can't (easily) tell which individual widget generated the event, just that one of them did.)

But a GUI button click can be broken down into two physical actions: the user presses the mouse down, and the user releases the mouse. These physical actions cause the creation of **event objects**. Binding of event objects is handled by a different mechanism.

Event Binding (Event Objects)

In situations where there is no built-in command option, or where we need to access detailed information about an event, we can't use command binding. Instead we can use **event binding**. This binds a widget to a callback via a specific **event object**. A reference to the event object is sent as an input argument to the callback. The callback can inspect the attributes of the event object to get information about the event. A typical example is when we want to inspect an object representing a "mouse down" to get the x and y coordinates of the event. A key press on the keyboard is another example.

The Path Ahead

We look at command binding in the upcoming Sections 2.9, 2.12, 2.13 and 2.14, and event binding in Section 2.15. You don't need to worry too much if the underlying theory seems a bit complicated at this stage, you can mostly get by just following the patterns.

2.9. Widgets for making things happen

Some widgets come with a number of built-in command binding options for handling typical GUI events. In this section we will take our first look at some of them.

The Button Widget

A Button is designed to be clicked. Like the Label widget, it may have text and/or an image on it.

Useful Options on top of the common options listed in Section 2.4

command: This binds the callback method to be executed if the user clicks the button.

textvariable, image, compound: As for the Label widget.

state: set to NORMAL by default, can be set to DISABLED in which case no callback will be triggered if it is clicked and it will appear grey.

In our first example, Code Listing 2.10, we'll explain what is happening in some detail. First run the code, click on each of the buttons and observe the behaviour, and then read on.

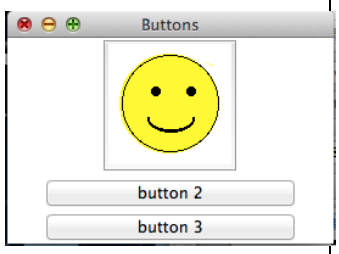
The `b2_method` is bound as the callback for the `b2` Button widget using the **command** option. When `b2` is pressed, `b2_method` sets the text on the `b2` Button. (Note that `b2` is being accessed from methods outside `__init__` so it needs to be an instance variable `self.b2` .)

The `not_b2` method is bound as the callback for both the `b1` and `b3` Button widgets using the **command** option. When either `b1` or `b3` is pressed, the `not_b2` method is called. It displays a message to the Python shell, and also sets the text on the `b2` Button (yes the same button as above).

In both cases these method calls happen **not** via an explicit method call in our code the way we are used to e.g. `b2_method()` or `not_b2()`, but automatically when the user clicks the button (generates an event on the Button widget) by the processes started in the `mainloop` method at the end of the main routine. Examples of statements performing the four GUI requirements from the introduction are indicated in the comments of Code Listing 2.10.

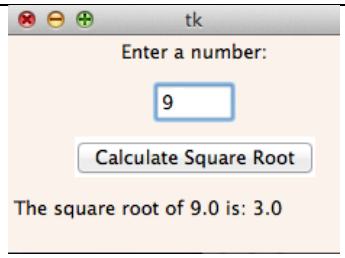
```
"""Chapter 2, Code Listing 2.10, Button Widgets"""
from tkinter import *

class ButtonsExampleGUI:
  def __init__(self, parent):
    self.photo = PhotoImage(file = "smiley.gif")
    # req. 1 - make a button with an image
    b1 = Button(parent,text = "button 1", image = self.photo,
                bg = "white", command = self.not_b2) # req. 3 (associate . . .)
    b1.pack()
    self.b2 = Button(parent, width = 15, anchor = W, text = "button 2",
                relief = RIDGE, command = self.b2_method) # req. 3
    self.b2.pack(padx = 30)
    b3 = Button(parent, width = 15, anchor = E, text = "button 3",
                relief = RIDGE, command = self.not_b2) # req. 3
    b3.pack()

  def b2_method(self): # req. 2 (specify action) - callback for button widget b2
    self.b2['text'] = "you pushed me"

  def not_b2(self): # req. 2 (specify action) - callback for button widgets b1 and b3
    print("It isn't actually good practice to combine a GUI with output
        to the Python shell but it sure is useful for testing purposes!")
    self.b2['text'] = "wasn't me"

# main routine
if __name__=="__main__":
  root = Tk()
  root.title("Buttons")
  buttons = ButtonsExampleGUI (root)
  root.mainloop() # req. 4 (initiate event handling)
```

Note that on a Mac, the operating system may prefer to display buttons in the Macintosh style, shamelessly ignoring our anchor and relief commands:



Code Listing 2.10

Code listing 2.11 uses many of the features covered so far. The user is prompted to enter a number in the Entry widget. A Button connected to a callback method waits to be clicked. In the callback method, the "number" is retrieved from the entry widget using the widget's `get` method and then converted to a float value in a try/except block. If the conversion is successful, the output label displays a string showing the square root of the number.




```
"""Chapter 2, Code Listing 2.11 Square root GUI"""
from tkinter import *

class SquareRootGUI:

    def __init__(self, parent):
     entry_label = Label(parent, text = "Enter a number:")
     entry_label.pack()

     self.e1 = Entry(parent, width = 5)
     self.e1.pack(padx = 2, pady = 10)

     b1 = Button(parent, text = "Calculate Square Root", command = self.square_root)
     b1.pack()

     self.output_label = Label(parent, text = "The square root is:")
     self.output_label.pack(padx = 2, pady = 10, anchor = W)


    def square_root(self): # callback for button b1
     try:
       f = float(self.e1.get())
       self.output_label.configure(text = "The square root of " + str(f) + " is: " + str(f**0.5))
     except:
       self.output_label.configure(text = "Improper input")

#main routine
if __name__=="__main__":
    root = Tk()
    entry_ex = SquareRootGUI(root)
    root.geometry("300x200+25+25")
    root.mainloop()
```

Code Listing 2.11

Remember:

- **Do not include the parameter list brackets with the method name when you assign it to the command option**. i.e.

  `command = self.square_root` **NOT** `command = self.square_root()`

  The second case would cause the method to be called before the widget has been created. Confusion would ensue!

Try this exercise 3

Write a GUI converter program which takes a value representing some measurement and converts it to a different scale e.g. farenheit to Celsius, kg to pounds, miles to kilometres, American $ to NZ $ . . .

Make sure the information is clearly labeled, and it is clear what the user should do.

The Checkbutton Widget

The Checkbutton widget provides a labeled check box for the user to click. It is used in situations where the user is allowed to choose many out of many.

Useful Options

command: As for the Button widget

variable: (**required** option) Each Checkbutton is assigned a separate instance of a Variable class, typically IntVar

onvalue: The value given to the Variable instance if the Checkbutton is selected. Default is 1.

offvalue: The value given to the Variable instance if the Checkbutton is not selected. Default is 0.

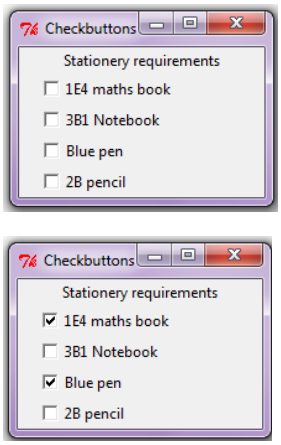
Useful Methods

deselect Deselects (unchecks) the Checkbutton and sets the value of its Variable instance to its off value (see Code Listing 2.13).

If an instance of IntVar has been used, the `IntVar` method `get` will access the value that is associated with the current state of the Checkbutton widget (see Section 2.6). For a Checkbutton this will be either the `onvalue` or the `offvalue`, depending on whether it is selected or not.

Take a look at Code Listing 2.12 then read the explanation which follows.

```
"""Chapter 2, Code Listing 2.12, Checkbutton Widget"""
from tkinter import *

class CheckbuttonsExampleGUI1:

  def __init__(self, parent):
      label1 = Label(parent, text = "Stationery requirements")
      label1.pack()
      # one IntVar for each Checkbutton
      v1 = IntVar()
      v2 = IntVar()
      v3 = IntVar()
      v4 = IntVar()

      self.cb1 = Checkbutton(parent, width = 20, variable = v1, anchor = W,
                 text = "1E4 maths book", command = self.highlight)
      self.cb1.var = v1 # adding an instance variable
      self.cb1.pack(padx = 15)

      self.cb2 = Checkbutton(parent, width = 20, variable = v2, anchor = W,
                text = "3B1 Notebook")
      self.cb2.var = v2
      self.cb2.pack()

      self.cb3 = Checkbutton(parent, width = 20, variable = v3, anchor = W,
                 text = "Blue pen")
      self.cb3.var = v3
      self.cb3.pack()

      self.cb4 = Checkbutton(parent, width = 20, variable = v4, anchor = W,
                text = "2B pencil")
      self.cb4.var = v4
      self.cb4.pack()

    def highlight(self):
        """incomplete method - see Try this exercise 4"""
        self.cb1.configure(bg = "yellow")

#main routine
if __name__=="__main__":
    root = Tk()
    buttons = CheckbuttonsExampleGUI1(root)
    root.mainloop()
```



Code Listing 2.12

The value of each Checkbutton in Code Listing 2.12 is stored in an `IntVar` object. By default the value is set to 1 if checked (`onvalue`) and 0 if unchecked (`offvalue`). In Code Listing 2.12 we have made each variable instance an attribute of the Checkbutton they belong to. The name we chose to give this attribute is `var`. We have consistently used the same attribute name for each of our Checkbutton instances. Maintaining good, consistent naming is an important discipline to have. We have also made each Checkbutton an attribute of our CheckbuttonExampleGUI1 class using `self`.

The `highlight` method is only called when `cb1` is clicked (selected **or** deselected). The first time cb1 is clicked, this works well. The `1E4 maths book` text is highlighted by a yellow background. But what if we click it again to deselect it? The yellow remains.

Try this exercise 4

We can access the state of the Checkbutton using the `get` method of its variable class instance, which will return 1 if the Checkbutton is selected and 0 otherwise.

```
    if self.cb1.var.get() == 1:
# do something that responds to the fact that cb1 is selected
    else:
# do something that responds to the fact that cb1 is not selected
```

Add code to the `highlight` method of Code Listing 2.12 so that the cb1 Checkbutton's background will turn yellow when it is selected and white when it is not selected. Then add the same functionality to the other three Checkbuttons.

A List of Checkbuttons

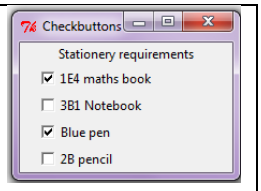
Code Listing 2.13 is similar to Code Listing 2.12 except that the Checkbuttons are stored in a List and created in a loop. We've used `StringVar` instead of `IntVar`, so `onvalue` and `offvalue` will be strings. The `onvalue` and the `text` options are both set to the next item in the `cb_strings` list, so the text returned for each selected Checkbutton will be the same as the text showing on it.

Each time a Checkbutton is selected or deselected the `display_selections` method is called. If the Checkbutton is selected its value will not be `"*"`, and therefore its value will be printed.

Yes, we are printing to the Python Shell again in the middle of a GUI program, sigh. But it is okay because we are just testing . . .right? Run the code, then see if you can follow how it works.

Note:- we could have chosen not to specify the offvalue, leaving it at the default rather than setting it to `"*"`. In this case in the `if` statement we would have had to check for the string value `"0"` rather than the int value `0`, because a `StringVar` is being used.

```
"""Chapter 2, Code Listing 2.13, Checkbutton Widgets in a List"""
from tkinter import *

class CheckbuttonsExampleGUI2:

    def __init__(self, parent):
        label1 = Label(parent, text = "Stationery requirements")
        label1.pack()
        cb_strings = ["1E4 maths book","3B1 Notebook","Blue pen","2B pencil"]
        self.check_btns = []
        for i in range(len(cb_strings)): # set up each Checkbutton
            v = StringVar()
            self.check_btns.append(Checkbutton(parent, width = 20, variable = v,
                       anchor = W, onvalue = cb_strings[i], offvalue = "*",
                       text = cb_strings[i],
                       command = self.display_selections))
            self.check_btns[i].var = v
            self.check_btns[i].deselect()
            self.check_btns[i].pack()

    def display_selections(self):
         for i in range(len(self.check_btns)):
            if self.check_btns[i].var.get() != "*":
                print(self.check_btns[i].var.get())
         print( " * * * ")
```



Shell output with 1E4 and Blue pen selected:

```
1E4 maths book
Blue pen
 * * *
```



```
#main routine
if __name__=="__main__":
    root = Tk()
    buttons = CheckbuttonsExampleGUI2(root)
    root.mainloop()
```

Code Listing 2.13

The Radiobutton Widget

Radiobuttons are used when a user is intended to select only one of many. Radiobuttons that are logically grouped together share a variable instance, but each have their own specific value. As each Radiobutton is clicked the variable is set to the value associated with that Radiobutton.

Within a group of Radiobuttons, when one is selected the others are automatically deselected. There can be only one. The initial selection is set by calling the `set` method on the Variable instance. If the variable is set to a value belonging to a Radiobutton in the group then that Radiobutton will be the one initially selected. If set to a value which is not a value belonging to any of the Radiobuttons in the group, no Radiobutton will be initially selected.

Useful Options

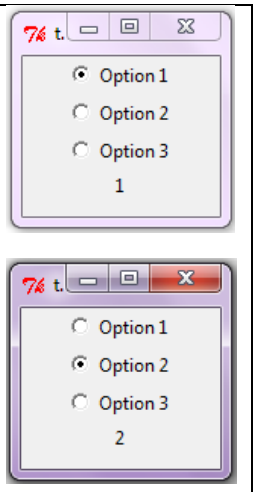
command, variable: As for Checkbuttons.

value: The value given to the Variable instance if the Radiobutton is selected. Equivalent to Checkbutton's onvalue (no equivalent to offvalue is needed since the Variable instance is shared).

Command is not used in the examples below. We get away with using local scope for the Radiobuttons and Label because no other method is called. The IntVar however does not behave properly as an initial value for the Radiobutton unless it is an instance variable.

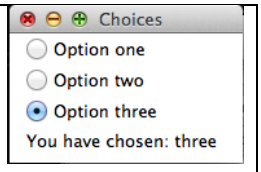
Code Listing 2.14 shows a group of Radiobuttons sharing a variable with a `Label`. The first screenshot shows the initial setup. The second screenshot shows the `Label` updated with the value of the new selection.

```
"""Chapter 2, Code Listing 2.14, Radiobutton Widgets Example using IntVar"""
from tkinter import *

class RadiobuttonsExampleGUI:

  def __init__(self, parent):
    self.v = IntVar() #just one instance of IntVar is created

    self.v.set(1) # sets the initial selection -this is rb1's value so rb1 will be the intital selection

    rb1 = Radiobutton(parent, variable = self.v, value = 1, text = "Option 1")
    rb2 = Radiobutton(parent, variable = self.v, value = 2, text = "Option 2")
    rb3 = Radiobutton(parent, variable = self.v, value = 3, text = "Option 3")
    label1 = Label(parent,  textvariable = self.v) # tkinter seems happy to convert
                                                   #  IntVar to text for textvariable

    rb1.pack()
    rb2.pack()
    rb3.pack()
    label1.pack()
#main routine
if __name__=="__main__":
  root = Tk()
  buttons = RadiobuttonsExampleGUI (root)
  root.mainloop()
```



Code Listing 2.14

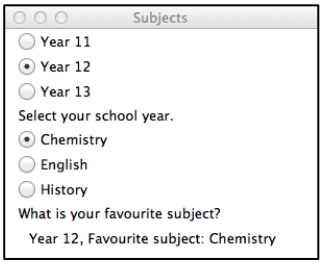
Code Listing 2.15 performs a similar purpose to that of Code Listing 2.14, but uses an instance of `StringVar` instead of `IntVar`, and also uses command binding. The differing length of the output string in this example has required some extra layout instructions - see the embedded comments.




```
"""Chapter 2, Code Listing 2.15, Radiobutton Widgets StringVar Example"""
from tkinter import *

class RadiobuttonsStringVarExampleGUI:
   def __init__(self, parent):
      self.v = StringVar() # just one instance of StringVar is created
      self.v.set("")  # sets the initial selection to no selection

      rb1 = Radiobutton(parent, variable = self.v, value = "one", text = "Option one",
                                                          command = self.display_choice)
      rb2 = Radiobutton(parent, variable = self.v, value = "two", text = "Option two",
                                                          command = self.display_choice)
      rb3 = Radiobutton(parent, variable = self.v, value = "three", text = "Option three",
                                                          command = self.display_choice)
      self.output_label = Label(parent,  text = "What will you choose?")
      rb1.pack( anchor = W, padx = 10) # anchor W to left align the Radiobuttons and Label
      rb2.pack( anchor = W, padx = 10)
      rb3.pack( anchor = W, padx = 10)
      self.output_label.pack( anchor = W, padx = 10)

   def display_choice(self):
      self.output_label.configure(text = "You have chosen: " + self.v.get())

#main routine
if __name__=="__main__":
   root = Tk()
   buttons = RadiobuttonsStringVarExampleGUI (root)
   root.geometry("180x100+0+0") # set window size, to keep width constant while text length varies
   root.title ("Choices")
   root.mainloop()
```

Code Listing 2.15

Try this example 5



Write a program which uses two sets of Radiobuttons, one for school year, and one for subject choice.

Each set of buttons has a prompt label to instruct the user what to do.

An output label should display the data selected, in one sentence.

A possible solution is pictured.

2.10. A Container Widget: Frame

The Frame widget provides a useful container to group other widgets, either for layout purposes or because they logically belong together.

In Code Listing 2.16, each Label and Entry pair represent a question and its answer. We have put each question and answer pair on its own frame. Notice that `labQ1` and `entA1` are added to `f1`, while `f1` is added to `root` with `f1 = Frame(parent)`. Each question and answer widget is packed in to its frame, and each frame is packed in to its parent (`root`).

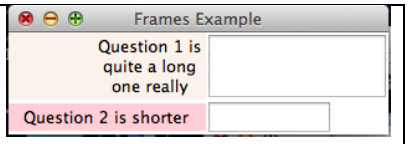
The `side` option of each Label's `pack` method keeps the related question and answer pairs beside each other within the frame rather than each widget being centred vertically in a column. The `side` option of the f2 `pack` method keeps the two frames left aligned.

The width of each Label is set by the WD constant, so the Entry widgets are left-aligned with each other.

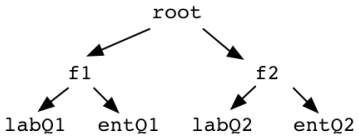
The `anchor = E` option of `labQ1` aligns the text to the right within the widget.

The `fill = Y, expand = TRUE` options of `entA1` make the Entry widget for the first answer take up the same height as the question, filling its frame height, although when the answer is displayed it will be centred within the vertical space.


```
"""Chapter 2, Code Listing 2.16, Frame Widget Example"""
from tkinter import *

class FrameExampleGUI:

  def __init__(self, parent):
    WD = 17 # width of Question labels
    f1 = Frame(parent)
    f2 = Frame(parent)

    labQ1 = Label(f1, text="Question 1 is quite a long one really",
        bg = "linen", wraplength = 100, width = WD, anchor = E)
    entA1 = Entry(f1, width = 15)
    labQ1.pack(side = LEFT)
    entA1.pack(fill = Y, expand = TRUE) # expands vertically to fill space
    labQ2 = Label(f2, text="Question 2 is shorter",
        bg = "pink", width = WD)
    entA2 = Entry(f2, width = 10)
    labQ2.pack(side = LEFT)
    entA2.pack()
    f1.pack()
    f2.pack(side = LEFT)

#main routine
if __name__=="__main__":
  root = Tk()
  frames = FrameExampleGUI (root)
  root.title("Frames Example")
  root.mainloop()
```



Parent child widget hierarchy for FrameExampleGUI:



Code Listing 2.16

Play around with this code. Add some more frames with their own Q and A widgets. Change, add or remove settings one by one and see what happens to the layout

2.11. Arranging Widgets

It isn't enough just to create a widget. We have to add it to our tkinter window. For this we will use a choice of two geometry manager methods, pack and grid.

Pack

Pack is appropriate to use as long as your desired layout is simple but soon feels cumbersome if things get more complicated. Pack was described on page 54 and has been used in every widget example so far. We will show some ways to use pack options to achieve layout effects in this section.

Code Listing 2.17 creates two columns of three buttons each. Each column is in its own frame, and the first frame is packed to the left hand side.

```
"""Chapter 2, Code Listing 2.17, Pack Buttons Example"""
from tkinter import *

class PackExGUI1:
    def __init__(self, parent):
        WD = 10
        f1 = Frame(parent)
        b1 = Button(f1,  text = "Button 1", WD = 10)
        b1.pack()
        b2 = Button(f1, text = "Button 2", WD = 10)
        b2.pack()
        b3 = Button(f1, text = "B U T T O N  3", WD = 10)
        b3.pack()
        f2 = Frame(parent)
        b4 = Button(f2,  text = "Button 4", WD = 10)
        b4.pack()
        b5 = Button(f2, text = "Btn 5", WD = 10)
        b5.pack()
        b6 = Button(f2, text = "Button 6", WD = 10)
        b6.pack()
        f1.pack(side = LEFT)
        f2.pack()
```

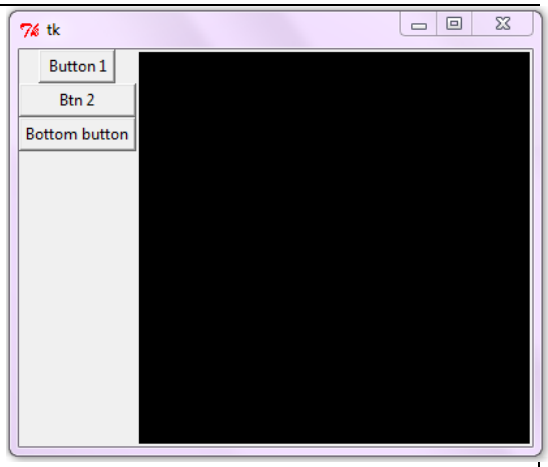
Mac screenshot.


```
#main routine
if __name__=="__main__":
    root = Tk()
    pack_layout = PackExGUI1(root)
    root.mainloop()
```

Code Listing 2.17

Code Listing 2.18 creates a frame of three buttons in a column side by side with the main_frame. The buttons are all on the `f1` frame, which is packed to the left of the main_frame.

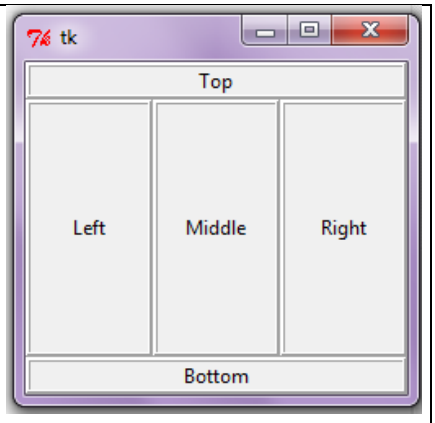
```
"""Chapter 2, Code Listing 2.18, Pack Example 2"""
from tkinter import *

class PackExGUI2:
    def __init__(self, parent):

        f1 = Frame(parent)
        f1.pack(anchor = N, side = LEFT)

        b1 = Button(f1, text = "Button 1")
        b1.pack()
        b2 = Button(f1, text = "Btn 2")
        b2.pack( fill = X)
        b3 = Button(f1, text = "Bottom button")
        b3.pack()

        main_frame = Frame(parent, width = 300,
               height = 300, bg = "black")
        main_frame.pack()

#main routine
if __name__=="__main__":
    root = Tk()
    pack_layout = PackExGUI2(root)
    root.mainloop()
```



Code Listing 2.18

Code Listing 2.19 creates a header and footer label and a row of three labels in between.

```
"""Chapter 2, Code Listing 2.19, Border Layout"""
from tkinter import *

class BorderLayoutGUI:

  def __init__(self, parent):
    top_label = Label( parent, text = "Top",
                       borderwidth = 3, relief = RIDGE)
    top_label.pack(side = TOP, fill = X)

    bottom_label = Label( parent, text = "Bottom",
              borderwidth = 3, relief = RIDGE)
    bottom_label.pack(side = BOTTOM, fill = X)

    left_label = Label(parent, text = "Left", width = 10,
              height = 10, borderwidth = 3, relief = RIDGE)
    left_label.pack(side = LEFT)

    right_label = Label(parent, text = "Right", width = 10,
              height = 10, borderwidth = 3, relief = RIDGE)
    right_label.pack(side = RIGHT)

    mid_label = Label(parent, text = "Middle",
              width = 10, height = 10,
              borderwidth = 3, relief = RIDGE)
    mid_label.pack() # defaults to the centre

#main routine
if __name__=="__main__":
    root = Tk()
    borders = BorderLayoutGUI (root)
    root.mainloop()
```



Code Listing 2.19

The order in which you pack things matters.

Try this exercise 6 - See the difference to the output of Code Listing 2.19 when you pack `bottom_label` last.

Complicated arrangements require more and more containers (typically frames). This rapidly results in unnecessarily bloated code. For this reason we shall introduce the grid geometry manager.

Grid

The grid method allows us to specify the row and column into which we wish to insert the widget. The grid instruction applies to the container object the widget is in. If containers are nested, it is the immediate container to which the grid refers. Thus you can have more than one grid going on at a time in a Tk window.

Useful parameters for the grid method

row: The number of the row in which the widget is being placed. Row numbers begin counting at 0. If any row numbers are skipped the next row will behave as though it is consecutive.

column: The number of the column in which the widget is being placed. Column numbers begin counting at 0. If any column numbers are skipped the next column will behave as though it is consecutive.

rowspan: The number of rows the widget is to span

columnspan: The number of columns the widget is to span

sticky: Essentially grid's version of anchor. It can be set to N, NE, E, SE, S, SW, W, NW and CENTER. You can achieve a fill effect by adding them eg `sticky = E+W` will fill horizontally. Likewise `sticky = N+W+S+E` will fill in all directions.

Useful methods

grid_forget: When called on a widget, this method removes the widget and forgets its position on the grid. The widget still exists and can be added back just by using grid as normal. You will need to specify the widget's position again.

grid_remove: When called on a widget, this method removes the widget and remembers its position on the grid. You can "regrid" it by simply calling `widget_name.grid()`. It will assume its former position. In some operating systems, the screen will need to be refreshed following a regrid, using the statement `root.update_idletasks()`

grid_propagate: In certain circumstances it might be desirable to create an empty widget (often a frame) of a specific width and height that at a later stage has widgets added to it to provide feedback or options to the user. If the widget (let us call it `f3`) tries to collapse, this can be solved by setting the propagate option to false with

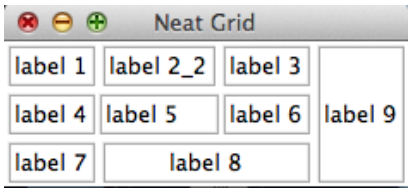
```
f3.grid_propagate(0)
```

This stops the Grid Geometry Manager from trying negotiate our widget's space away. See Code Listing 2.39.

Code Listing 2.20 shows grid being used to line some labels up neatly. Some labels have used `sticky` to fill the grid space completely. One has used `anchor` to align its text left. The Neat Grid screenshot is the result.


```
"""Chapter 2, Code Listing 2.20, Grid Geometry Neat Grid Example """
from tkinter import *
class NeatGridGUI:
    def __init__(self, parent):
        lab1 = Label(parent, text = "label 1", relief = RIDGE, padx = 2, pady = 2)
        lab1.grid(row = 0, column = 0, padx = 2, pady = 2)
        lab2 = Label(parent, text = "label 2_2", relief = RIDGE, padx = 2, pady = 2)
        lab2.grid(row = 0, column = 1, padx = 2, pady = 2)
        lab3 = Label(parent, text = "label 3", relief = RIDGE, padx = 2, pady = 2)
        lab3.grid(row = 0, column = 2, padx = 2, pady = 2)
        lab4 = Label(parent, text = "label 4", relief = RIDGE, padx = 2, pady = 2)
        lab4.grid(row = 1, column = 0, padx = 2, pady = 2)
        lab5 = Label(parent, text = "label 5", relief = RIDGE, padx = 2, pady = 2, anchor = W)
        lab5.grid(row = 1, column = 1, pady = 2, sticky = W+E)
        lab6 = Label(parent, text = "label 6", relief = RIDGE, padx = 2, pady = 2)
        lab6.grid(row = 1, column = 2, padx = 2, pady = 2)
        lab7 = Label(parent, text = "label 7", relief = RIDGE, padx = 2, pady = 2)
        lab7.grid(row = 2, column = 0, padx = 2, pady = 2)
        lab8 = Label(parent, text = "label 8", relief = RIDGE, padx = 2, pady = 2)
        lab8.grid(row = 2, column = 1,  columnspan = 2, sticky = E+W, padx = 2, pady = 2)
        lab9 = Label(parent, text = "label 9", relief = RIDGE, padx = 2, pady = 2)
        lab9.grid(row = 0, column = 3, rowspan = 3, sticky  =  N+S, padx = 2, pady = 2)

#main routine
if __name__=="__main__":
    root = Tk()
    root.title("Neat Grid")
    buttons = NeatGridGUI(root)
    root.mainloop()
```

Code Listing 2.20



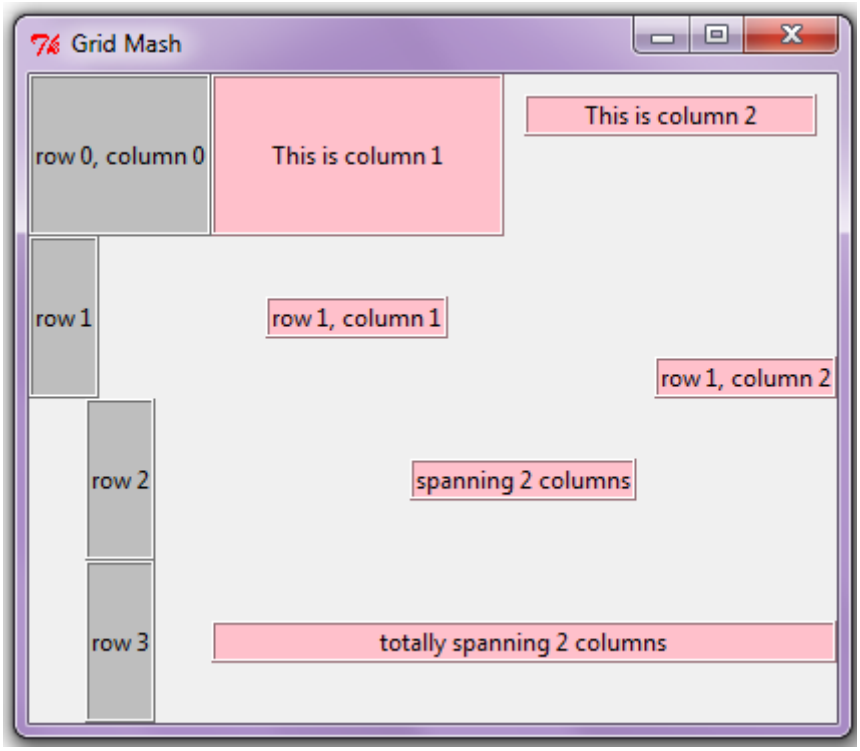
Code Listing 2.21 is a thorough but hideous example appropriately titled "Grid Mash"

```
"""Chapter 2, Code Listing 2.21, Grid Geometry Grid Mash Example """
from tkinter import *

class GridMashGUI:
   def __init__(self, parent):
      lab1 = Label(parent, text = "row 0, column 0",height = 5, bg = "gray", relief = RIDGE)
      lab1.grid(row = 0, column = 0)

      lab2 = Label(parent, text = "This is column 1", width =20, bg ="pink", relief = RIDGE)
      lab2.grid(row = 0, column = 1, sticky = N+W+S+E)

      lab3 = Label(parent, text = "This is column 2",width = 20, bg ="pink", relief = RIDGE)
      lab3.grid(row = 0, column = 2, sticky = NW, pady = 10, padx = 10)

      lab4 = Label(parent, text = "row 1", height = 5, bg = "gray", relief = RIDGE)
      lab4.grid(row = 1, column = 0, sticky = W)

      lab5 = Label(parent, text = "row 1, column 1",  bg = "pink", relief = RIDGE)
      lab5.grid(row = 1, column = 1)

      lab6 = Label(parent, text = "row 1, column 2",  bg = "pink", relief = RIDGE)
      lab6.grid(row = 1, column = 2, sticky = SE)

      lab7 = Label(parent, text = "row 2", height = 5, bg = "gray", relief = RIDGE)
      lab7.grid(row = 2, column = 0, padx = 10)

      lab8 = Label(parent, text = "spanning 2 columns",  bg = "pink", relief = RIDGE)
      lab8.grid(row = 2, column = 1,  columnspan = 2)
```



```
        lab9 = Label(parent, text = "row 3", height = 5, bg = "gray", relief = RIDGE)
        lab9.grid(row = 3, column = 0, padx = 10)

        lab10 = Label(parent, text = "totally spanning 2 columns", bg = "pink", relief = RIDGE)
        lab10.grid(row = 3, column = 1,  columnspan = 2, sticky = W+E)

#main routine
if __name__=="__main__":
    root = Tk()
    root.title("Grid Mash")
    buttons = GridMashGUI(root)
    root.mainloop()
```

Code Listing 2.21



Try this exercise 7

Replicate the `pack` version of the `BorderLayoutGUI` example of Code Listing 2.19 using `grid`.

Try this exercise 8

Replicate the appearance of `PackExGUI1` of Code Listing 2.17, using `grid`. Try using a loop to create the buttons from a list of strings. Store them in a list and use loops to add them to the grid.

2.12. More Practice with Widget Commands

We have already used the command option with Buttons, Checkbuttons and Radiobuttons to assign callbacks that will handle the events. This section will give you some more practice with these.

Remember:

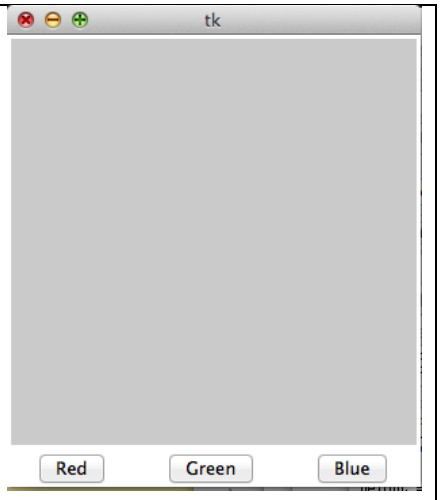
- **Do not include the parameter list brackets with the method name when you assign it to the command option**. i.e.

  `command = self.setColour` **NOT** `command = self.set_colour()`

Background Buttons Example

Code Listing 2.22 presents a coloured frame and three buttons. Each button when clicked, calls a method to change the background colour of the frame.

```
"""Chapter 2, Code Listing 2.22, Callback function, colour frame, 1"""
from tkinter import *

class BackgroundColourButtonsGUI:

    def __init__(self, parent):

        self.frame = Frame(parent, width = 300,
               height = 300, bg = "gray")
        self.frame.grid(row = 0, columnspan = 3)
        self.red_btn = Button(parent, text = "Red",
               command = self.set_red)
        self.red_btn.grid(row = 1, column = 0)
        self.green_btn = Button(parent, text = "Green",
               command = self.set_green)
        self.green_btn.grid(row = 1, column = 1)
        self.blue_btn = Button(parent, text = "Blue",
               command = self.set_blue)
        self.blue_btn.grid(row = 1, column = 2)

    def set_red(self):
        self.frame.configure(bg = "red")

    def set_green(self):
        self.frame.configure(bg = "green")

    def set_blue(self):
        self.frame.configure(bg = "blue")

#main routine
if __name__=="__main__":
    root = Tk()
    bg_buttons = BackgroundColourButtonsGUI(root)
    root.mainloop()
```



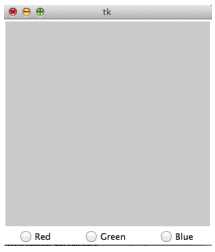
Code Listing 2.22

Try this exercise 9

The example above was a little clunky. A better choice is to use Radiobuttons. This would remove the need for three separate methods. The Radiobuttons will need to share an instance of StringVar. Using StringVar rather than IntVar means the value (i.e. the colour) can be directly assigned to the `bg` option of the frame. Adapt the code as described:

- Change the Buttons to Radiobuttons.
- Make an instance of StringVar called `colour_var`. Use each Radiobutton's `variable` option to share this instance.
- Give each Radiobutton a `value` which is the name of the colour to be shown.
- Remove two of the callback methods, and rename the remaining one `set_bg`. Each Radiobutton should be bound by its `command` option to this callback method.
- Call the `get` method on `colour_var` when assigning the `bg` value (as in Code Listing 2.15)
- Set the variable instance initially to a value not held by any Radiobutton, so none of the Radiobuttons are initially selected.

Run your adapted program. If your adaptation is successful it works like this: the text of the `value` attribute of any Radiobutton is assigned to `colour_var` if that button is selected, so if `red_btn` is clicked, the value "`red`" is assigned to `colour_var`. If `green_btn` is clicked, "`green`" is assigned to `colour_var`. Since `colour_var` is shared, it will always hold the value of the last Radiobutton clicked.

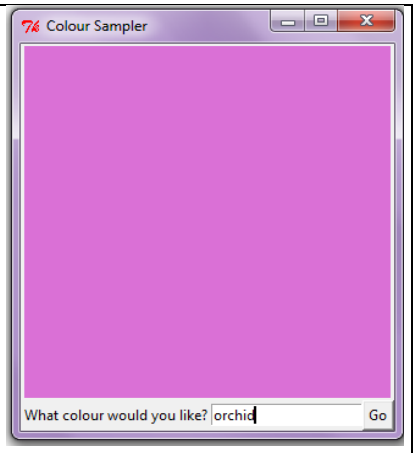


When the `set_bg` method is called, whatever value `colour_var` has at that moment is what will be used to set the background colour.

Selecting a Colour with an Entry widget Example

Code Listing 2.23 has a similar purpose to the previous example. The difference is that the user can choose any named or RGB format colour desired. When the Go button is pressed, the text from the Entry widget is used to set the background colour. This offers a richer reward, but it comes at a risk. Entry inputs of `firebrick` and `#ff0000` will produce the desired result, but what if you accidentally type `firbrick`, or leave out the `#`, or some other error?

```
"""Chapter 2, Code Listing 2.23, Callback function, Entry widget"""
from tkinter import *

class ColourEntryExampleGUI:

    def __init__(self, parent):

        self.frame = Frame(parent, width = 350,
               height = 300, bg = "gray")
        self.frame.grid(row=0, columnspan = 3)

        instruction_label = Label(parent,
               text = "What colour would you like?")
        instruction_label.grid(row = 1, column = 0)

        self.colour_entry = Entry(parent, width = 14)
        self.colour_entry.grid(row = 1, column = 1)

        self.go_btn = Button(parent, text = "Go",
               command = self.set_col)
        self.go_btn.grid(row = 1, column = 2)

    def set_col(self):
        try:
            self.frame.configure(bg=self.colour_entry.get())
        except TclError:
            self.frame.configure(bg = "gray") # incorrect colour
#main routine
if __name__=="__main__":
    root = Tk()
    bg_color_ex = ColourEntryExampleGUI (root)
    root.title("Colour Sampler")
    root.mainloop()
```



Code Listing 2.23

The program copes with this problem by using a try/except in the callback method. How did we know what error would be triggered if the string entered didn't match a valid tkinter colour? Well, we triggered one and looked at the error message. If this error occurs, the background will be gray. Otherwise it will be some other colour as typed into the Entry widget by the user.

Task 1: Fun With Cats

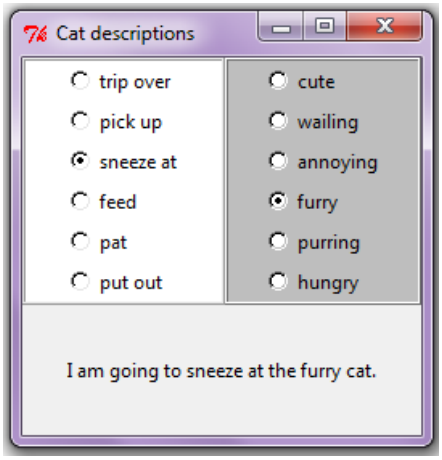
Make a list of verbs, which are things you might do to a cat (be nice).

Make another list of adjectives, which might describe a cat.

Using the lists create two groups of Radiobuttons.

A label displays a sentence with the chosen words incorporated each time there is a change in selection e.g.

I am going to **feed** the **hungry** cat.
I am going to **pat** the **cute** cat.
I am going to **put out** the **annoying** cat.



2.13. More Widgets - Scale and ScrolledText

The Scale Widget

The scale widget is a nifty way to select a number value from a specified range using a 'draggable' slider.

Useful Options

command: A similar pattern to the other widgets you've seen so far but with one difference - the callback method will be passed the value of the scale. **The callback method definition needs a second parameter in addition to self to store this value.**

variable: can be assigned an instance of either `IntVar()` or `DoubleVar()` but this is not required. We can access the value of a Scale widget by calling the Scale widget's `get` method.

from_: The beginning of the range. In Python `'from'` is a reserved word, hence the underscore. (Trailing underscores are used to avoid conflict with a reserved word.)

to: The end of the specified range

resolution: The amount by which the range increments

orient: Takes HORIZONTAL and VERTICAL (VERTICAL is the default).

length: Whether the orientation is vertical or horizontal the length is always how far it slides. Measured in pixels.

width: The 'thickness' of the scale. Measured in pixels

sliderlength: The length of the slider.

label: An in-built label.

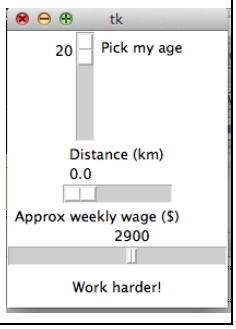
Useful Methods

get: Scale has a get method to access its value, e.g. `wage_picker.get()`

set: Scale has a set method to set its value, e.g. `wage_picker.set(240)`

- Scale does not have an anchor option.
- Scale does not have a value option.


```
"""Chapter 2, Code Listing 2.24, Scale Widget Example"""
from tkinter import *
class ScalesExampleGUI:

   def __init__(self, parent):
        age_picker = Scale(parent, from_ = 20, to = 100, showvalue = 80,  label = "Pick my age")
        age_picker.pack()

        distance_picker = Scale(parent, orient = HORIZONTAL, to = 10, label = "Distance (km)",
                                                                          resolution = 0.5)
        distance_picker.pack()
        wage_picker = Scale(parent, orient = HORIZONTAL, from_ = 100, to = 5000,
                  label = "Approx weekly wage ($)", sliderlength = 10, length = 200,
                                                    resolution = 100, command = self.assess_wage)
        wage_picker.pack()
        self.wage_response = Label(parent, height = 2)
        self.wage_response.pack()

   #Must accept the value of the scale widget for which it is a callback.
   def assess_wage(self, wage):
      if int(wage) > 3000:
         self.wage_response.configure(text = "Awesome")
      else:
         self.wage_response.configure(text = "Work harder!")

#main routine
if __name__=="__main__":
   root = Tk()
   scales = ScalesExampleGUI (root)
   root.mainloop()
```



Code Listing 2.24

The ScrolledText Widget

The standard tkinter library includes a text widget and a scrollbar widget. The ScrolledText widget is a combination of both of these. It provides a text pane for both entering and displaying multiple lines of text as well as a scrollbar so that any text that does not fit within the widget's prescribed width and height is still accessible to the user.

Text inside a ScrolledText widget can be formatted by the program. Different fonts and sizes can be used within the same text pane.

ScrolledText does require a slightly special import statement. This is bolded in Code Listing 2.25 below.

Useful Options

width, height: measured in text units. The default width is 80 and the default height is 24.

wrap: can be assigned "char" (the default) which means that at the end of a line a word may just be split in the middle at whichever point there is no more room on the line. Other values are "word" which will wrap at the space closest to the end of the line, and "none" which will place long lines of text off the end of the line where it will not be visible to the user without introducing a horizontal scrollbar (NOT RECOMMENDED!)

state: The default is "normal". This allows the user to type into the window, highlight, delete, cut, paste etc. If state is set to "disabled" the widget simply displays text to the user but does not permit the user to edit it. While "disabled" the widget's insert and delete methods (see next section) will not work. If the program wishes to alter the contents of the widget but stop the user from doing so, it must first set the state to "normal", make the change, and then immediately change the state back to "disabled"

Useful Methods

insert: Similar to the insert method of the entry widget except that the index needs two parts. We must specify the line as well as the character position in the line. Lines start counting at 1 while character positions start counting from zero. Indices are typically written as strings, e.g."3.7" or '10.59'. The position of the first character in the text is "1.0". Another important index is END which means the position just past the current end of the text.


```
#inserts the word 'Hello' at the beginning of the text (at line 1, position 0).
        widget_name.insert('1.0', "Hello")

#inserts the text "Line 3, position 7" at the 8th character position on the third #line
        widget_name.insert('3.7', "Line 3, position 7")
```

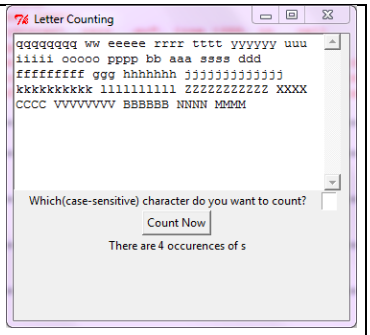
To move down a line simply insert the newline character `"\n"`

delete: If passed just one argument (an index) it removes just the character at that position. If passed two indices it removes text from the first index up to *but not including* the second index.

get: Returns the specified text. Takes the same parameters as delete.

focus As for Entry widgets.

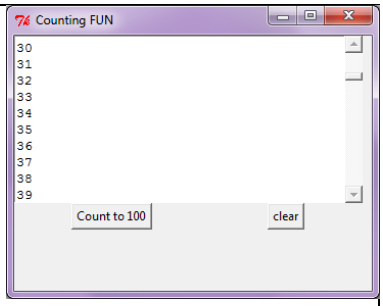
Code Listing 2.25 shows an example of a ScrolledText widget in a normal state (i.e. the user can type into it). The user can then enter a character into the Entry widget. Clicking the Count Now button displays how many times that character currently occurs in the ScrolledText widget




```
""" Chapter 2, Code Listing 2.25  Scrolled Text Example"""
from tkinter import *
from tkinter.scrolledtext import *

class ScrolledTextEx1:
   def __init__(self, parent):

      self.user_typing = ScrolledText(parent, width = 55,
                                      height = 10, wrap = 'word')
      self.user_typing.grid(row = 0, columnspan = 2)

      self.count_label = Label(parent,
                           text = "Which(case-sensitive) character do you want to count?" )
      self.count_label.grid(row = 1, column = 0, sticky = SE, padx = 10)

      self.char_entry = Entry(parent, width = 2)
      self.char_entry.grid(row = 1, column = 1, sticky = SW)

      self.count_btn = Button(parent, text = "Count Now",
                          command = self.count_chars)
      self.count_btn.grid(row = 2, columnspan = 2)
      self.feedback_label = Label(parent, text = "")

   def count_chars(self):
      user_input = self.user_typing.get('1.0', END)
      target = self.char_entry.get()
      num_chars = user_input.count(target)
      # Different grammar depending on how many
      if num_chars == 0:
            self.feedback_label.configure(text = "There are no occurences of " + target)
      elif num_chars == 1:
            self.feedback_label.configure(text = "There is just one " + target)
      else:
            self.feedback_label.configure(text = "There are " str(num_chars)+ " occurences of " + target)
      self.feedback_label.grid(row = 3, columnspan = 2)
      self.char_entry.delete(0, END)

#main routine
if __name__=="__main__":
   root = Tk()
   root.title("Letter Counting")
   root.geometry("420x300+200+200")
   scroll_ex = ScrolledTextEx1(root)
   root.mainloop()
```

Code Listing 2.25

Code Listing 2.26 shows a ScrolledText widget being used just to display text (the numbers from 1 to 100). Note the changes to the state of the widget to prevent the user from being able to edit the ScrolledText widget. Notice also the usefulness of the built-in scrollbar.




```
""" Chapter 2, Code Listing 2.26  Scrolled Text Example 2"""

from tkinter import *
from tkinter.scrolledtext import *

class ScrolledTextEx2:
    def __init__(self, parent):
        self.display_nums = ScrolledText(parent, width = 40,
                height = 10, state = 'disabled', wrap = 'word')
        self.display_nums.grid(row = 0, columnspan = 2)
        self.count_100 = Button(parent, text = "Count to 100", command = self.count_to_100)
        self.count_100.grid(row = 1, column = 0)

        self.clear = Button(parent, text = "clear", command = self.clear)
        self.clear.grid(row = 1, column = 1)

    def count_to_100(self):
        self.display_nums.configure(state = 'normal')
        for i in range(101):
            self.display_nums.insert(END, str(i) + "\n")
        self.display_nums.configure(state = 'disabled')

    def clear(self):
        self.display_nums.configure(state = 'normal')
        self.display_nums.delete('1.0', END)
        self.display_nums.configure(state = 'disabled')

#main routine
if __name__=="__main__":
    root = Tk()
    root.title("Counting FUN")
    root.geometry("350x250+200+200")
    scroll_ex = ScrolledTextEx2(root)
    root.mainloop()
```

Code Listing 2.26

Planning for GUI applications

Many of the examples so far have been quite simple in terms of their function. As more complicated tasks are attempted, we need to make sure we have planning strategies in place.

Designing a Graphical User Interface requires extra levels of planning.

- A physical description of the layout - where should the items sit in the tkinter window - is a good starting point. A diagram or sketch of the layout can describe the placement of elements.
- A flowchart can describe the behaviour expected of each event (Button/Checkbutton click, Scale slide etc.).
- The class itself with all its widgets and methods can be represented by a class diagram.

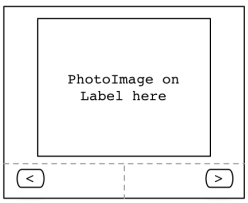
Code Listing 2.27 contains code which will deliver a slide show so long as there are images with the correct name in the correct place. The three diagrams below show the different planning processes for this program.

The **layout diagram** shows how the items might be arranged in the tkinter window. The dotted lines show the underlying grid layout.

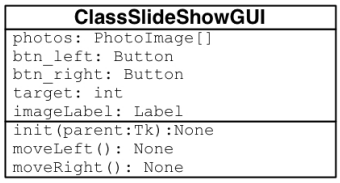
The **UML class diagram** shows all the instance variables and methods. Make sure all necessary widgets, images and variables have the correct scope to be used/accessed in a second method.

- Are all necessary widgets attributes of our GUI class.
- Are all necessary Variable instances either attributes of our GUI class or attributes of widgets that are attributes of our GUI class?
- Are all images either attributes of our GUI class, or attributes of widgets that are attributes of our GUI class?

There is a **flow chart for** each event (in this case each button click).

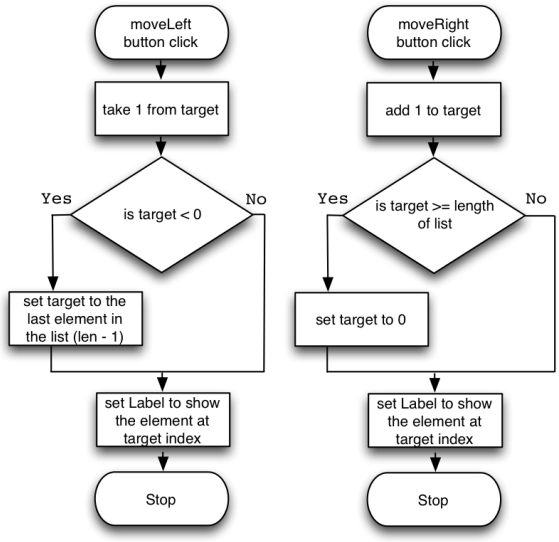


Layout Diagram



UML Class Diagram

Note that there is just one Label, and a different image (list element, PhotoImage) gets placed on it when a button is clicked.



Flowcharts to describe each button's behaviour


```
"""Chapter 2, Code Listing 2.27: Slide Show - loops in either direction"""
from  tkinter import *
class SlideShowGUI:

    def __init__(self, parent):
        self.photos = []

        self.photos.append(PhotoImage(file = "images/pic1.ppm"))
        self.photos.append(PhotoImage(file = "images/pic2.ppm"))
        self.photos.append(PhotoImage(file = "images/pic3.ppm"))
        self.photos.append(PhotoImage(file = "images/pic4.ppm"))
        self.photos.append(PhotoImage(file = "images/pic5.ppm"))
        self.photos.append(PhotoImage(file = "images/pic6.ppm"))
        self.photos.append(PhotoImage(file = "images/pic7.ppm"))
        self.photos.append(PhotoImage(file = "images/pic8.ppm"))
        self.photos.append(PhotoImage(file = "images/pic9.gif"))

        self.btn_left = Button(parent, text = "<", command = self.moveLeft)
        self.btn_right = Button(parent, text = ">", command = self.moveRight)
        self.target = 0
        self.imageLabel = Label(parent, image = self.photos[0], height = 150,
                            width = 200, padx = 20)
        self.imageLabel.grid(row = 0, columnspan = 2)
        self.btn_left.grid(row = 1, column = 0, sticky = W)
        self.btn_right.grid(row = 1, column = 1, sticky = E)
        parent.configure(bg = "white")

    def moveLeft(self):
        self.target -= 1
        if self.target < 0:
          self.target =  len(self.photos) - 1
        self.imageLabel.configure(image = self.photos[self.target])

    def moveRight(self):
        self.target += 1
        if self.target >= len(self.photos):
          self.target = 0
        self.imageLabel.configure(image = self.photos[self.target])

#main routine
if __name__ == "__main__":
    root = Tk()
    window = SlideShowGUI(root)
    root.geometry("200x185+0+0")
    root.mainloop()
```

Code Listing 2.27

Encapsulation and GUIs

Of the concepts relating to encapsulation set out in section 1.12 of Chapter 1, the logical design of classes (maximised cohesion, minimised coupling) are always relevant. Managing access to data is often of concern in OO programming but is likely to be of limited concern in the kinds of simple GUI programs that we are demonstrating in this workbook.

In the case of simple GUIs we need to make sure that callback methods do all that is required (including any checks and updates) when they are triggered by GUI events.

Defining additional classes

So far in this chapter, as we've been getting used to a lot of new tkinter material, our programs have been quite simple. They have consisted of a main routine and a GUI class. It is important to remember that as programs become more complicated, a crucial aspect of good OO programming is the definition of well-structured support classes. We are going to look at a very simple example below.

Imagine if our slide show items from code listing 2.27 didn't just consist of pictures but a title, a picture and a corresponding caption. Well, we could try to keep track of three separate lists but the perils of parallel list processing are best avoided (see chapter1, section 1.1). Instead we can define a second class, SlideShowItem, so that these three things can be stored together as one coherent object.

In this case the class is an extremely simple one, consisting of just an `__init__` method to initialise the instance variables. It is easy to imagine however a situation in which a support class might need more functionality than this.

Code listing 2.28 shows the new version of SlideShowGUI. Instead of a list of photos we have a list of `SlideShowItem` objects.

```
"""Chapter 2, Code Listing 2.28: Slide Show with 2 classes"""
from  tkinter import *
class SlideShowItem:
    """stores the information for one image - title, caption and the image itself as a PhotoImage"""
    def __init__(self, title, photo_file, caption):
        self.title = title
        self.photo = PhotoImage(file = photo_file)
        self.caption = caption

class SlideShowGUI:
    """Presents a slide show of images, with captions. Has buttons for the user to scroll forward and back"""
    def __init__(self, parent):
        # make a list of SlideShowItem objects
        self.slide_items = []
        self.slide_items.append(SlideShowItem("Title 1", "pic1.ppm", "Caption for this item"))
        self.slide_items.append(SlideShowItem("Title 2", "pic2.ppm", "Caption for this item"))
        self.slide_items.append(SlideShowItem("Title 3", "pic3.ppm", "Caption for this item"))
        self.slide_items.append(SlideShowItem("Title 4", "pic4.ppm", "Caption for this item"))
        self.slide_items.append(SlideShowItem("Title 5", "pic5.ppm", "Caption for this item"))
        self.slide_items.append(SlideShowItem("Title 6", "pic6.ppm", "Caption for this item"))
        self.slide_items.append(SlideShowItem("Title 7", "pic7.ppm", "Caption for this item"))
        self.slide_items.append(SlideShowItem("Title 8", "pic8.ppm", "Caption for this item"))
        self.slide_items.append(SlideShowItem("Title 9", "pic9.ppm", "Caption for this item"))
        # set up the GUI objects
        self.btn_left = Button(parent, text = "<", command = self.moveLeft)
        self.btn_right = Button(parent, text = ">", command = self.moveRight)
        self.target = 0
        self.title_label = Label(parent, text = self.slide_items[0].title,
                                                    font = ("Times", "24", "bold"))
        self.image_label = Label(parent, image = self.slide_items[0].photo,
                                                  height = 150, width = 200, padx = 20)
        self.caption_label = Label(parent, text = self.slide_items[0].caption,
                                                          font = ("Times", "12"))
        self.title_label.grid(row = 0, columnspan = 2)
        self.image_label.grid(row = 1, columnspan = 2)
        self.caption_label.grid(row = 2, columnspan = 2)
        self.btn_left.grid(row = 3, column = 0, sticky = W)
        self.btn_right.grid(row = 3, column = 1, sticky = E)
        parent.configure(bg = "white")
    def moveLeft(self):
        """previous button click, shows previous element in list"""
        self.target -= 1
        if self.target < 0:
           self.target =  len(self.slide_items) - 1
        self.title_label.configure(text = self.slide_items[self.target].title)
        self.image_label.configure(image = self.slide_items[self.target].photo)
        self.caption_label.configure(text = self.slide_items[self.target].caption)
    def moveRight(self):
        """next button click, shows next element in list"""
        self.target += 1
        if self.target >= len(self.slide_items):
           self.target = 0
        self.title_label.configure(text = self.slide_items[self.target].title)
        self.image_label.configure(image = self.slide_items[self.target].photo)
        self.caption_label.configure(text = self.slide_items[self.target].caption)
#main routine
if __name__ == "__main__":
    root = Tk()
    window = SlideShowGUI(root)
    root.geometry("200x245+0+0")
    root.mainloop()
```

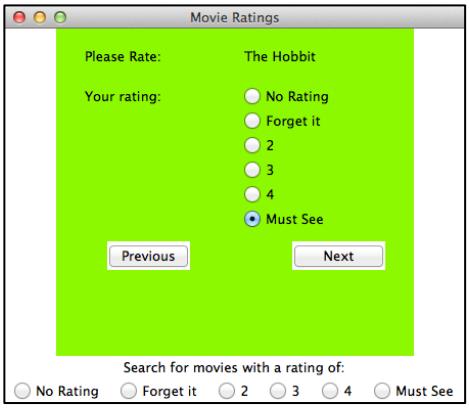
Code listing: 2.28

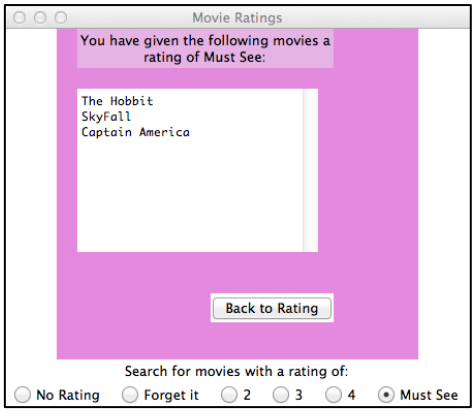
2.14. Guided Task: Rating Movies

The screen shots below show two screen layouts produced by a program that lets a user scroll through a list of movies, rating each one in turn. At any point in the program the user can search for movies of a particular rating by selecting one of the radio buttons along the bottom of each screen.

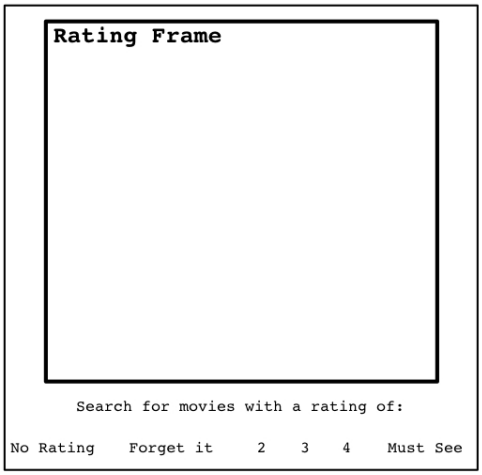
It uses a simple support class called `Movie` with instance variables for a movie's `title` and `rating`.

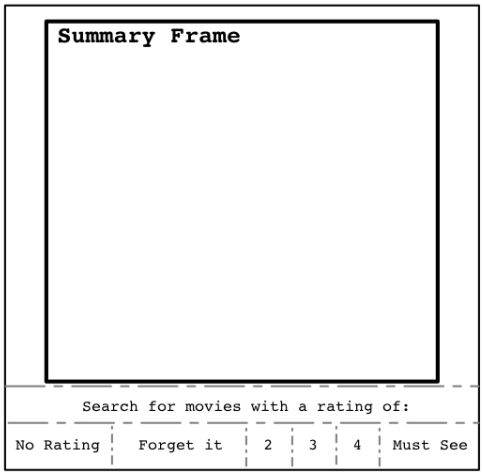
The MovieRating class provides the GUI and establishes the list of Movie objects, so needs a well-planned design and set-up. It has two layout options. The layouts share the content of the parent's grid rows 1 and 2, but row 0 has a different frame spanning the grid for each layout. Which frame is displayed and which is removed depends on the events triggered by the users actions. (You can remove an element from a grid with `grid_forget`).





Each frame has its own grid to arrange its own components. The grid of the parent window is suggested in the right hand layout diagram below. Plan the grid layout for the two frames.





Identify the widgets common to both layouts.	Identify the widgets just used on the Rating Frame	Identify the widgets just used on the Summary Frame

These are a good starting point for the UML class diagram.

What should the default rating of a movie object be?

_____

What would you want to happen when the user clicks a Radiobutton in the coloured part of the first (rating) screen?

_____

_____

What would you want to happen when the user clicks any Radiobutton in the bottom part of the first screen?

_____

_____

What would you want to happen when the user clicks any Radiobutton in the bottom part of the second screen?

_____

_____

What would you want to happen when the user clicks the Next button?

_____

_____

What would you want to happen when the user clicks the Previous button?

_____

_____

What would you want to happen when the user clicks the Back to Rating button?

_____

_____

Using what you have written above produce UML class diagrams and flow diagrams like those on page 78. You will need four flowcharts, one for each button and one for any change in a "search Radiobutton"

When you are happy with your understanding of the task and your plan, take a look at the flow charts, design and coded solution provided in code listing 2.39 (sneakily hidden at the end of the chapter on page 101). This is a working program but it leaves room for some improvements. We have put some parts of particular concern in bold. What improvements to the solution could make it more flexible and robust? Easier to read and understand? You might like to discuss this in a group. When you are happy, use your suggestions to improve the code.

Checklist to guide your discussion:

What happens when the user:

- Scrolls to the end of the movies list?
- Finds only 1 movie with a particular rating?
- Finds no movies with a particular rating?

What happens when the programmer:

- Decides to add another rating category or remove an existing one
- Decides to change the colour scheme

Look for values which are repeated, which should be constants.

Look for widgets which could be stored in a data structure such as a list.

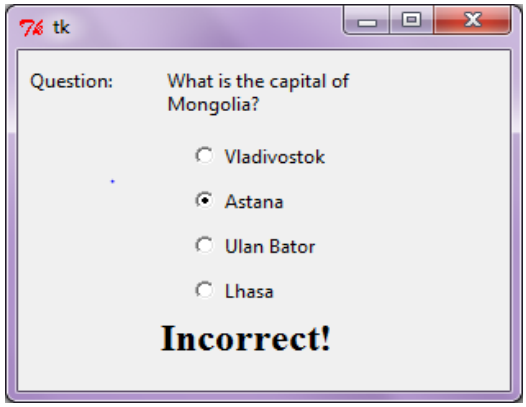
Extension:

It might be nice to search for a movie by name while in "rating mode" in order to change its rating, or to search for movies which haven't yet been rated. Think about how you might add these features.

Task 2: Multi Choice Quiz

Create a multi choice quiz with just one question.

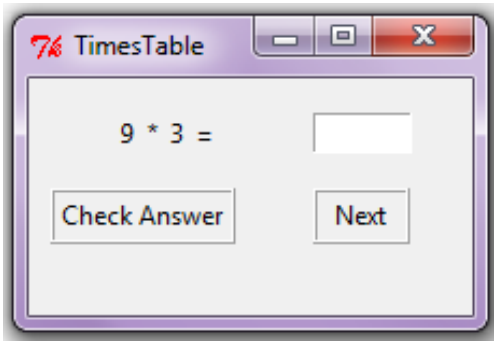
You should be able to reuse the Question class from Chapter 1 as a support class. Plan and write a GUI class to go with it.



Task 3: Times Table Tester

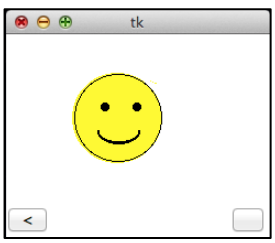
Write a times table tester with a GUI similar to that shown below.

Note: The question is generated randomly and the answer calculated as needed - there is no need for a support class.



Task 4: Slide Show

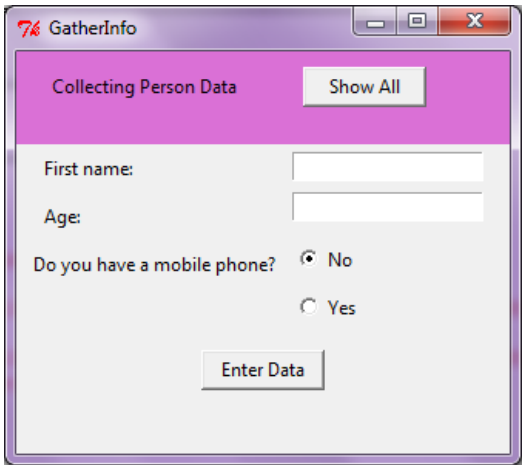
Create a slide show of your choosing. Stop at each end of the slide show and "disable" the appropriate arrow button with `button_name.configure(state = DISABLED)`. The button can be reactivated with `button_name.configure(state = NORMAL)`.

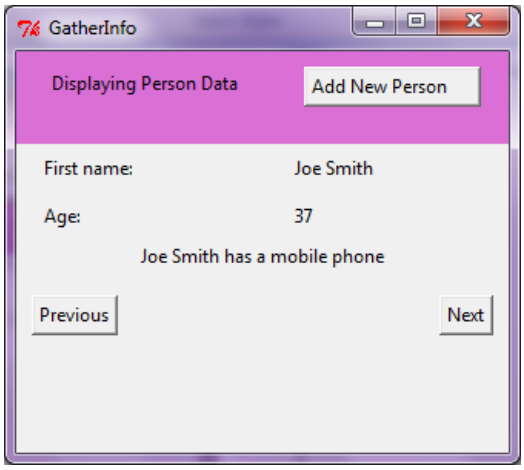


Task 5: Collecting Data

Make a GUI application which collects name, age and the answer to the survey question "Do you have a mobile phone?".When an Enter Data button is pressed, the information is collected from the GUI and used to construct an object for this person, stored in a list. When a Show All button is pressed, the data collection frame is removed and replaced by a frame in which each person's information is presented one by one using the Next/Previous buttons. An 'Add New Person' button will toggle the layout back to "data collection mode".

Plan carefully, following a similar process to that used for the Rating Movies task.

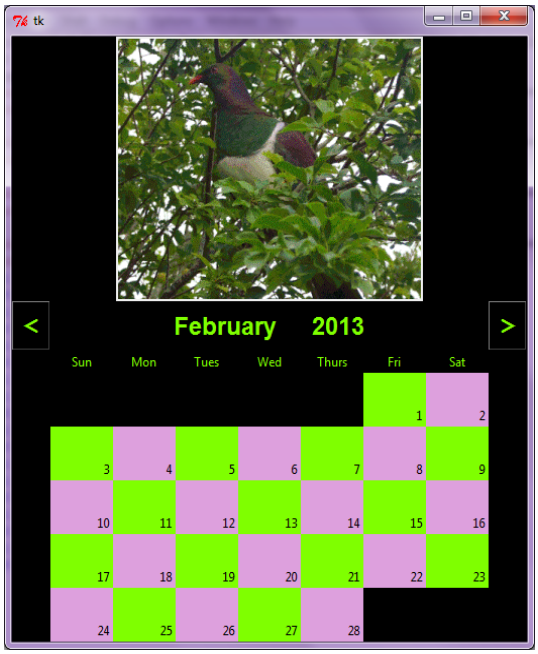




Task 6: Calendar

Make a calendar with a different picture for each month, and the ability to scroll through the months of a year.

Planning is by far the most important part of this project.



2.15. Event Binding

Responding to a component simply and directly using the command option is great up to a point BUT:

Only some widgets have a command option.

- What if we want something to happen when we click on a Frame or a Label?
- What if we want to implement behaviour for which there is no command option, e.g. type into an Entry widget and have something happen when we press the enter/return key instead of reaching for a mouse to click another button?
- What if we want to be able to get detailed information about an event, e.g. the x and y coordinates of a mouse click?
- What if we want to respond to a list of widgets with the same sort of behaviour but we need to know which widget was clicked?

To do this we need to have access to the underlying language **event objects** (see Section 2.8). We need to be able to bind callbacks to widgets via event objects. In short, we need **event binding**.

Many events support event binding in tkinter. We will focus on just a few. In this section we look at a number of mouse events. In the optional section at the end of the chapter we look at keyboard events.

Basic pattern for event binding

widget_name.bind(event, callback)

- the binding applies to a particular widget (this could possibly be the root widget in some circumstances)
- the event is one of those listed below e.g. "<Enter>" represents a mouse entering a widget
- the callback is the name of a function/method

e.g. `self.frame.bind("<Enter>", method_1)`

If the mouse enters self.frame, method_1 is called

NOTES:

1) Events are described using strings which contain the name of the event inside angle brackets
2) All event objects have a `type` attribute which returns a digit representing a code for the type of event e.g. the mouse event "<Enter>" is 7, "<Leave>" is 8, a key press of any key is 2.
3) All event objects have a `widget` attribute, which returns a reference to the widget that generated the event.
4) The callback function that handles the event must accept a reference to the **event object** as a parameter.

Mouse events

A mouse enters a widget

"<Enter>" An unpressed mouse enters the area covered by a widget

A mouse leaves a widget

"<Leave>" An unpressed mouse leaves the area covered by a widget

A mouse button is pressed

"<Button-1>" Button-1 is the left mouse button (or for a one-button mouse, the only button). This button is pressed somewhere within the area covered by a widget.

"<Button-2>" Button-2 is the middle mouse button

"<Button-3>" Button-3 is the right mouse button

A mouse is dragged

A drag consists of a mouse button being pressed, the mouse being moved and the mouse button released.

"<B1-Motion>" B1 is the left mouse button

"<B2-Motion>" B2 is the middle mouse button

"<B3-Motion>" B3 is the left mouse button

The mouse must be pressed within the area of the attached widget but the "drag" will continue until the mouse button is released, even if the mouse leaves the area of the widget.

Mouse event attributes

x: The x position of the mouse over the widget to which the event has been bound.

y: The y position of the mouse over the widget to which the event has been bound.

x_root: The x position of the mouse when in relation to the screen

y_root: The y position of the mouse in relation to the screen

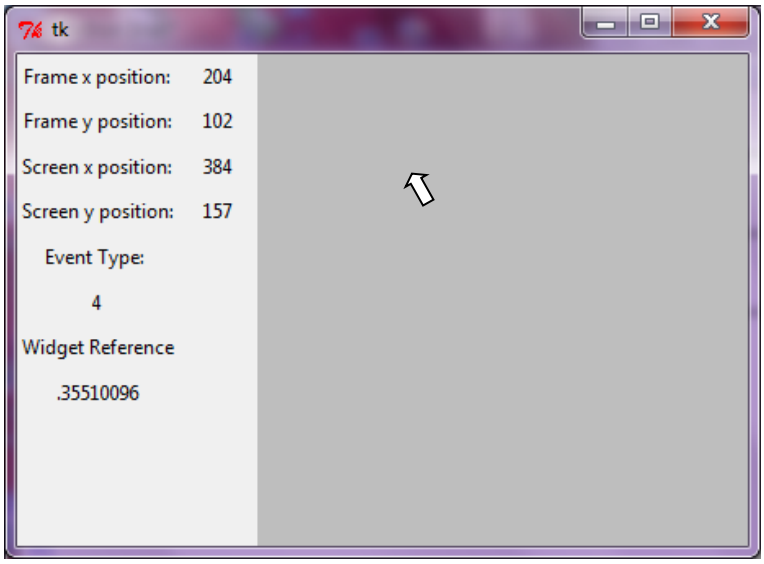
Code Listing 2.29 shows how mouse event attributes can be used. It displays information about mouse button press events on a Frame widget.

```
"""Chapter 2, Code Listing 2.29, Mouse Position"""
from tkinter import *

class MousePosition:

    def __init__(self, parent):

        x_label = Label(parent, text = "Frame x position:")
        x_label.grid(row = 0, column = 0)

        self.x_pos = Label(parent, width = 6,  text="n/a")
        self.x_pos.grid(row = 0, column = 1)

        y_label = Label(parent, text = "Frame y position:")
        y_label.grid(row = 1, column = 0)

        self.y_pos = Label(parent, width = 6, text="n/a")
        self.y_pos.grid(row = 1, column = 1)

        x_root_label = Label(parent, text = "Screen x position:")
        x_root_label.grid(row = 2, column = 0)

        self.x_root_pos = Label(parent,width = 6, text="n/a")
        self.x_root_pos.grid(row = 2, column = 1)

        y_root_label = Label(parent, text = "Screen y position:")
        y_root_label.grid(row = 3, column = 0)

        self.y_root_pos = Label(parent, width = 6, text="n/a")
        self.y_root_pos.grid(row = 3, column = 1)

        type_label = Label(parent, text="Event Type: ")
        type_label.grid(row = 4, column = 0)

        self.event_type = Label(parent, text="n/a")
        self.event_type.grid(row = 5, column = 0)

        widget_label = Label(parent, text = "Widget Reference")
        widget_label.grid(row = 6, column = 0)

        self.event_widget = Label(parent, text = "n/a")
        self.event_widget.grid(row = 7, column = 0)

        self.frame = Frame(parent, width = 300, height = 300, bg = "grey")
        self.frame.grid(row = 0, rowspan = 20, column = 2)

        self.frame.bind("<Button-1>", self.show_xy)
```



```
    def show_xy(self, event):
        """ accepts a reference to the event object that triggered this callback - Note 4
        we could have called it fred but we didn't """
        self.x_pos.configure(text = event.x)
        self.y_pos.configure(text = event.y)
        self.x_root_pos.configure(text = event.x_root)
        self.y_root_pos.configure(text = event.y_root)
        self.event_type.configure(text = event.type) # Note 2
        self.event_widget.configure(text = event.widget) # Note 3
# main routine
if __name__ == "__main__":
    root = Tk()
    clicks = MousePosition(root)
    root.mainloop()root.mainloop()
```

Code Listing 2.29



To see the screen position coordinates changing, pick a spot on the frame to click e.g. the top left corner. Then shift the window to another place on your screen and click on the same spot on the frame. The x and y positions should stay roughly the same, while the screen x and screen y positions will be quite different.

The numerical representation of the widget reference is not very useful or human-friendly! Being able to use event.widget inside our callback method however, is **very** useful.

We can access information about the widget, e.g.

```
print(event.widget['text'])
```

OR

```
if event.widget['height'] > 5 :
    #do something
```

We could reconfigure that particular widget, e.g.

```
event.widget.configure(bg = "pink")
```

If multiple widgets are bound to the same callback we can specify different responses depending on which widget triggered the callback e.g.

```
if event.widget == self.btn_1 or event.widget == self.btn_2:
    #do something
elif event.widget == self.calculate_btn:
    #do something else
```


Try adding a second Frame widget to the code listing 2.29:

```
self.frame_2 = Frame(parent, width = 300, height = 300, bg="white")
self.frame_2.grid(row = 0, rowspan=20, column = 3)
self.frame_2.bind("<Button-1>", self.show_xy)
```

Note the different widget reference when you click on each frame. Let's make the widget reference display something more user friendly. Replace the final statement in the `show_xy` method with this:

```
if event.widget == self.frame:
   self.event_widget.configure(text = "frame 1")
elif event.widget == self.frame_2:
   self.event_widget.configure(text = "frame 2")
```

Now when you click on each frame it will report which frame was clicked in English.

Mouse Drag example

To see mouse motion in action change the binding in your MousePosition class to be:

```
self.frame.bind("<B1-Motion>", self.show_xy)
```

Hold the mouse button down and move the mouse around the canvas to see the positions change.

Leave and Enter Example

A widget may have two events bound to the same callback method. The method can identify the source of the event using `event.type`. This is a digit code for the event type. The "<Enter>" event produces a "7" and the "<Leave>" event produces an "8". (Using a print statement to the shell is one way of identifying which code is attached to which event.) The digit is a character rather than an integer, so needs to be compared using, for instance, `i == "7"`. Or it could be converted to an integer. We have chosen the latter for Code Listing 2.29, which, when run, changes the colour of the frame when the mouse enters it and again when it leaves it.

```
""" Chapter 2, Code Listing 2.30, Mouse Position Colours """
from tkinter import *

class MouseFrameColour:
    def __init__(self, parent):

        c_label = Label(parent, text = "Event example:")
        c_label.grid(row = 0, column = 0, sticky = N)

        self.frame = Frame(parent, width = 300, height = 300, bg = "gray")
        self.frame.grid(row = 0, column = 1)

        self.frame.bind("<Leave>", self.change_colour)
        self.frame.bind("<Enter>",self.change_colour)


    def change_colour(self, event):
        print(event.type)  # prints event type to the shell
        i = int(event.type)  # no try except needed because event.type is known to be a digit character
        if i == 7:
            self.frame.configure(bg = "yellow")
        if i == 8:
            self.frame.configure(bg = "red")
#main routine
if __name__ == "__main__":
    root = Tk()
    clicks = MouseFrameColour(root)
    root.mainloop()
```

Code Listing 2.30

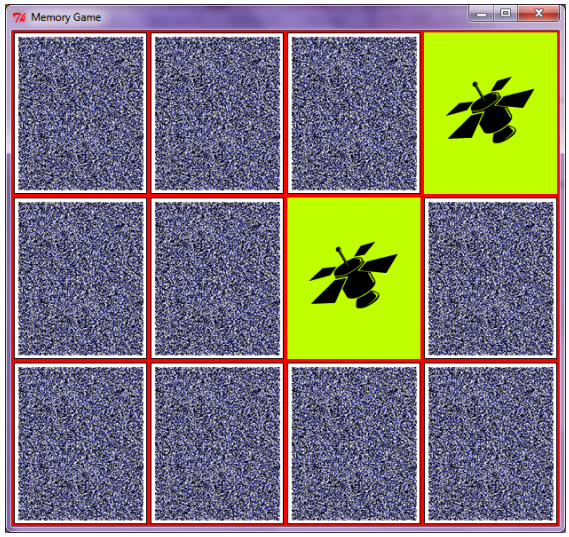
Flipping cards

The code in listing 2.31 represents the set up for a memory card game where cards have to be turned over and back again. Pairs of cards have the same image on the 'front'. At the beginning they are all shown 'face down'. All the labels use the same image for their 'backs'

We want to respond to each label with exactly the same behaviour BUT we definitely need to know which one was clicked. Event binding allows us to do precisely that using event.widget.

Each label has its own 'front' image attribute.

Each label has its own instance of IntVar() to keep track of whether it is currently face down (0) or face up (1).




```
""" Chapter 2, Code Listing 2.31, Flipping Cards """
from tkinter import *

class Memory:
    def __init__(self, parent):
        """ Sets up a list of 12 labels with pairs of Label instances possessing the same 'front' image.
        All labels use the same 'back' image. Each label has its own instance of IntVar() to keep track of
        whether it is flipped to show the 'front' image (1) or the 'back' image (0 - the default).
        """
        self.labels = []

        # Since this image is used by all the label widgets it makes sense to store it as an attribute of this instance of the Memory class
        self.card_back_image  = PhotoImage(file = "card_back.gif")

         # Ideally these would be given a random order
        image_names = ["bus.gif", "plane.gif", "train.gif", "satellite.gif",
                       "house.gif", "ship.gif", "satellite.gif", "plane.gif",
                       "bus.gif", "train.gif", "ship.gif", "house.gif"]
        for i in range(len(image_names)):

            self.labels.append(Label(parent, image = self.card_back_image,
                                     width = 150, height = 182, bg = "red"))

            # this label gets its own 'front' image
            self.labels[i].img = PhotoImage(file = image_names[i])

            # this label gets its own instance of IntVar()
            self.labels[i].var = IntVar()

        " put each Label on the grid and bind a mouse click on them to the callback
        i = 0
        for r in range(3):
            for c in range(4):
                self.labels[i].grid(row = r, column = c)

                # binds each label to self.flip_card if clicked
                self.labels[i].bind("<Button-1>", self.flip_card)
                i += 1

    def flip_card(self, event):
        """ Code applies to whichever widget has been clicked. This is determined by event.widget
            The method checks the widget's state by inspecting the value of the widget's variable.
        """
        if event.widget.var.get()==0:
            event.widget.var.set(1)
            event.widget.configure(image = event.widget.img)
        else:
            event.widget.var.set(0)
            event.widget.configure(image = self.card_back_image)

#main routine
if __name__ == "__main__":
    root = Tk()
    game = Memory(root)
    root.mainloop()
```

Code Listing 2.31

The unbind method

```
widgetname.unbind(event)
```

- This method deletes all bindings between the specified widget and the specified event. This is useful if we, for example, only want something to happen the first time a widget is clicked or the first time a certain key is typed. To unbind a widget which has just been clicked with a mouse we might use the statement `event.widget.unbind("<Button-1>")`.

Task 7: Tic Tac Toe

Create a game of tic tac toe for two (human) players. Provide information to the users about whether it is X's or O's turn. When the game is over state who, if anyone, won (X, O or "No winner"). Our code used a list of 9 Labels. Plan carefully the actions required after each turn.

You do not need to collect information about the players.

Remember:

- you can disable a binding with the unbind method.
- you can access the widget which was the source of an event using event.widget, so could use expressions such as

```
event.widget.configure( text = "some text")
if event.widget == buttons[0] :
```

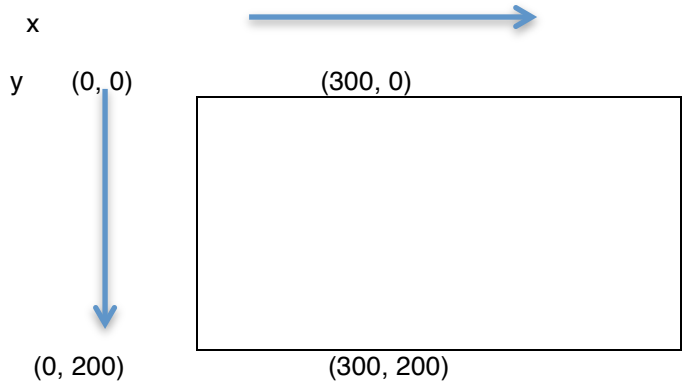
Congratulations. If you've reached this point you have a great skill set to progress with programming and Computer Science.

The chapter changes tack slightly from here to introduce other useful concepts, including some graphics content in preparation for animation.

2.16. A Graphics Widget : Canvas

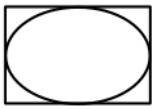
The Canvas widget provides a graphics environment. It comes with a host of methods that allow us to create lines, ovals, rectangles etc. We will keep it simple and just use a few basics.

Canvas coordinates can seem a little topsy-turvy if you're used to graphs in Mathematics. Coordinates in tkinter consist of two x,y pairings. The `x` coordinate is incremented from left to right (as per usual so far) and `y` from top to bottom (yikes!). A 300 by 200 canvas has the corner coordinates shown below.



Items are drawn on the Canvas at positions determined by coordinates.

create_line	takes 4 integer arguments representing the coordinates of the first point and the coordinates of the second point, and any number of options e.g. width, fill
create_rectangle	takes 4 integer arguments representing the x, y coordinates of the left top corner and the x, y coordinates of the right bottom corner, then any number of options e.g. fill, outline, width (for border)
create_oval	An oval is defined by the rectangle which would contain it, called its bounding box. Takes 4 integer arguments representing the x, y coordinates of the left top corner and the x, y coordinates of the right bottom corner of the bounding box, then any number of options e.g. fill, outline, width (for border)
create_text	takes 2 integer arguments representing the x and y coordinates for positioning the text, and any number of options, especially `text` e.g. `text = "show me"`
coords	accesses the coordinates of a Canvas item. if used with one argument, the name of the Canvas item, it returns a tuple listing the coordinates of that item. if used with more than one argument, it redraws a canvas item at a new position on the canvas. The first argument is the name of the canvas item. Then come the new coordinates at which the item will be redrawn. These must be in the right order.



Useful Options

fill may refer to fill colour (default is transparent) OR text colour (default is black)

outline border colour (default is black)

width (border or line) thickness of border or line in pixels (default is 1)

width (text) length of line (for create_text)

text text to display (for create_text)

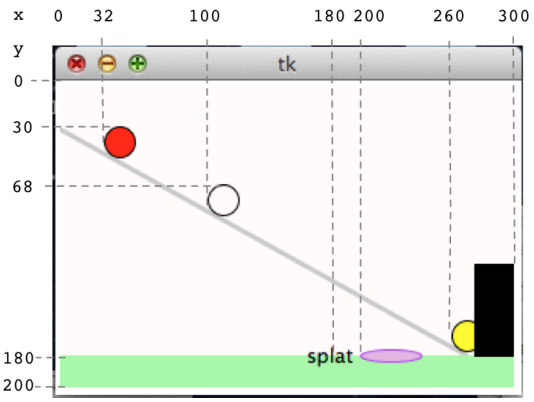
Useful methods:

Code Listing 2.32 creates eight canvas items: a line, four ovals, two rectangles and a text item. Constants were used for some of the significant coordinate values. The diagram below shows some of the relevant coordinates used for drawing the shapes.

Notice that these create methods are called on the canvas object. They create objects (collectively known as Canvas items) that can be assigned to variables. (See also Code Listing 2.33.)

```
"""Chapter 2, Code Listing 2.32, Canvas Widget Example"""
from tkinter import *

class CanvasExampleGUI:
   def __init__(self, parent):
      WD = 300
      HT = 200
      GRASS_HT = 180
      BALL_HT = 20

      canvas = Canvas(parent, width = WD, height = HT, bg = "snow")

      slide = canvas.create_line(0, 30, 270, GRASS_HT, width = 3, fill = "gray")

      grass = canvas.create_rectangle(0, GRASS_HT, WD, HT, fill = "pale green",
                                                           outline = "pale green")

      ball1 = canvas.create_oval(32, 30, 32 + BALL_HT, 30 + BALL_HT, fill = "red")

       # no fill means ball2 will be transparent
      ball2 = canvas.create_oval(100, 68, 100 + BALL_HT, 68 + BALL_HT)

      ball3 = canvas.create_oval(200, GRASS_HT - 4, 200 + BALL_HT * 2, GRASS_HT + 4,
                                                      fill = "plum", outline = "purple")

      ball4 = canvas.create_oval(260, GRASS_HT - BALL_HT - 3, 260 + BALL_HT,
                                                        GRASS_HT - 3, fill = "yellow")

      box = canvas.create_rectangle(WD - 25, GRASS_HT - 60, WD, GRASS_HT, fill = "black")

      splat = canvas.create_text(180, GRASS_HT, text = "splat")

      canvas.pack()

#main routine
if __name__ == "__main__":
   root = Tk()
   splat = CanvasExampleGUI(root)
   root.mainloop()
```

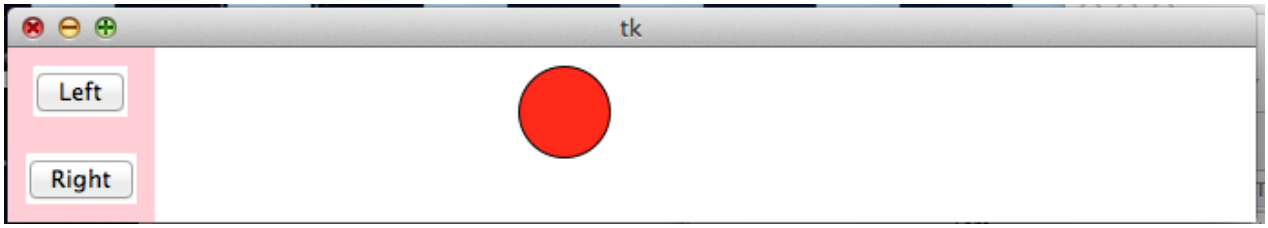
Code Listing 2.32



Canvas items will be drawn in order, so an item may cover or partially cover an item drawn previously.
(These items can be moved, grouped, deleted and so on, but it is not the subject of this chapter.)


```
"""Chapter 2, Code Listing 2.33, Jumping Ball"""
from tkinter import *
class CoordsActionGUI:
    def __init__(self, parent):
        f1 = Frame(parent, bg = "pink")
        f1.pack(anchor = N, side = LEFT, fill = Y)
        b1 = Button(f1, text = "Left", command = self.jump_left)
        b1.pack(padx = 10, pady = 10)
        b2 = Button(f1, text = "Right", command = self.jump_right)
        b2.pack(padx = 10, pady = 10)

        self.canvas = Canvas(parent, width = 600, height = 90, bg = "white")
        x = 300
        y = 10
        self.DIAMETER = 50
        self.circle = self.canvas.create_oval(x, y, x + self.DIAMETER, y + self.DIAMETER,
                                               fill = "red")
        self.canvas.pack()

    def jump_right(self):
        x = self.canvas.coords(self.circle)[0] # gets current x value which is first element of coords tuple
        y = self.canvas.coords(self.circle)[1] # gets current y value which is second element of coords tuple

        x += 100 # new x value

        self.canvas.coords(self.circle, x, y, x + self.DIAMETER, y + self.DIAMETER) # redraw itself

    def jump_left(self):
        x = self.canvas.coords(self.circle)[0] # gets current x value which is first element of coords tuple
        y = self.canvas.coords(self.circle)[1] # gets current y value which is second element of coords tuple

        x -= 100 # new x value

        self.canvas.coords(self.circle, x, y, x + self.DIAMETER, y + self.DIAMETER) # redraw itself
#main routine
if __name__ == "__main__":
    root = Tk()
    coords_ex = CoordsActionGUI(root)
    root.mainloop()
```

Code Listing 2.33



When a button is pressed, the circle is redrawn. The callback methods collect the current x and y position using the `coords` method with one argument. The circle is redrawn with a new x value using the `coords` method with five arguments.

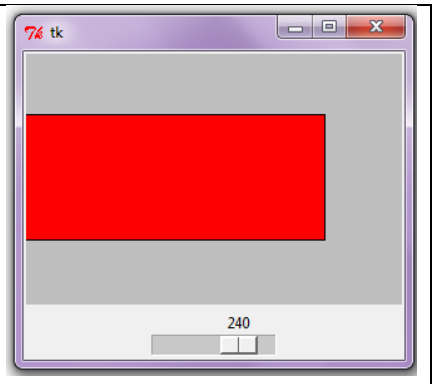
You could now take another look at the Bakery and GraphicalCake classes from chapter 1, which use the Canvas widget in all its glory.

2.17. Working with the Scale Widget

Scaling a rectangle example

Command is also available with the Scale widget. As before we simply assign to it the name of the callback method we wish to be executed when the widget is clicked / selected / moved.

Code Listing 2.34 creates a rectangle on a canvas which can be widened and narrowed by the user.




```
"""Chapter 2, Code Listing 2.34, Scaling a rectangle"""
from tkinter import *

class ResizeRectangleGUI:

  def __init__(self, parent):

    WD = 300

    HT = 200

    self.STRIPE_HT = 100

    self.STRIPE_Y = 50


    self.canvas = Canvas(parent, width = WD, height = HT, bg = "gray")
    self.canvas.grid(row  =  0, column = 0 )
    self.rect = self.canvas.create_rectangle(0, self.STRIPE_Y, 10, self.STRIPE_Y +
                                                          self.STRIPE_HT, fill = "red")
    self.width_picker = Scale(parent, from_ =  10, to = WD, orient = HORIZONTAL,
                                              resolution = 10, command = self.resize_rect)
    self.width_picker.grid(row = 1, column = 0)
  def resize_rect(self, width):
    self.canvas.coords(self.rect, 0, self.STRIPE_Y, width, self.STRIPE_Y + self.STRIPE_HT)
#main routine
if __name__ == "__main__":
  root = Tk()
  rr = ResizeRectangleGUI (root)
  root.mainloop()
```

Code Listing 2.34

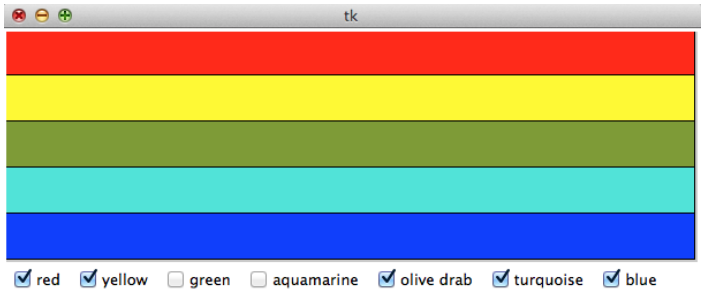
The callback function uses the `width` returned by the Scale widget for the "right" coordinate of the rectangle `coord` instruction. (Remember the callback method will be passed the value of the scale. **The callback method definition needs a second parameter in addition to self to store this value.)**

Try this exercise 10

Adapt the program of Code Listing 2.34 to make a square, which enlarges while staying square.

Task 8: Stripes

Write a program which offers the user a choice of colours. When the user selects a colour the canvas should be filled with stripes of equal width - one in each colour selected. The stripes could be horizontal or vertical. A possible output sequence is pictured.



Extension: Add a Scale to specify the width of the stripes, and repeat the pattern until the Canvas is full.

2.18. More Event Binding: Key Events

Events can be generated by a key press of any key on the keyboard. A different event code is generated by each key combination. Key events are bound to a callback function. The basic pattern is the same as for mouse events.

Normal character keys

The events for alpha numeric keys are predictable when you know the pattern.

e.g. "<k>" The letter k is pressed.

"<j>" The letter j is pressed.

"<K>" The letter K is pressed (i.e.; either the shift key and the k key are pressed at the same time, or the caps lock is on and k is pressed)

Special character keys

There are a range of non-alpha-numeric named key events. Some useful ones are listed below.

"<Return>" The return key.

"<Space>" The space bar.

"<Up>" The up arrow

"<Key>" Any key

Two keys together

"<Shift-Up>" the shift key and the up arrow key are pressed at the same time

"<Alt-k>" The letter k is pressed (i.e.; the alt key and the k key are pressed at the same time)

Key event attributes

char: The character typed. Special keys return an empty string.

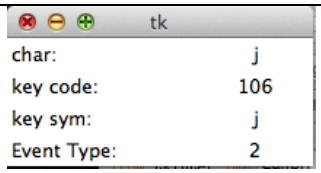
keycode: The code of the key typed. This is an integer value.

keysym: The symbol of the key typed.

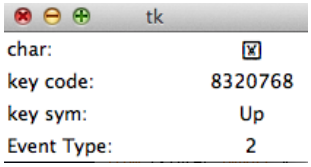
Code Listing 2.35 displays information about keystrokes. The "<Key>" event is bound to the entire root (i.e. parent) widget. If it were bound to an individual widget, you would have to use the tab key to move through the widgets until the one you wanted had the focus. This is called tab traversal, and in most circumstances would confuse the user.

KeyInfo example

```
"""Chapter 2, Code Listing 2.35, KeyStrokes"""
from tkinter import *

class KeyStrokes:
    def __init__(self, parent):
        char_label = Label(parent, text = "char:", width =
14,
                anchor = W)
        char_label.grid(row = 0, column = 0, sticky = W)
        self.char = Label(parent, width = 6,  text = "n/a")
        self.char.grid(row = 0, column = 1)

        code_label = Label(parent, text = "key code:")
        code_label.grid(row = 1, column = 0, sticky = W)
        self.keycode = Label(parent, width = 6, text = "n/a")
        self.keycode.grid(row = 1, column = 1)

        sym_root_label = Label(parent, text = "key sym:")
        sym_root_label.grid(row = 2, column = 0, sticky = W)

        self.key_sym = Label(parent, width = 8, text = "n/a")
        self.key_sym.grid(row = 2, column = 1)

        type_label = Label(parent, text="Event Type: ")
        type_label.grid(row = 4, column = 0, sticky = W)

        self.event_type = Label(parent, text = "n/a")
        self.event_type.grid(row = 4, column = 1)

        parent.bind("<Key>", self.show_key_info)

    def show_key_info(self, event):
        self.char.configure(text = event.char)
        self.keycode.configure(text = event.keycode)
        self.key_sym.configure(text = event.keysym)
        self.event_type.configure(text = event.type)
#main routine
if __name__ == "__main__":
    root = Tk()
    keys = KeyStrokes(root)
    root.mainloop()
```



Note that windows machines will produce the same key code (74) for J and j, where Mac will produce 74 for J and 106 for j, reacting to whether the shift key is down.



The up arrow, when pressed, generates the event "<Up>") which has the `keysym` attribute `Up` (this is accessed by the expression `event.keysym`). While alpha-numeric characters have a keycode related to their ASCII value, other keys do not.

Code Listing 2.35

The mouse example from Code Listing 2.30 is adapted in Code Listing 2.36 so it uses a Canvas widget and changes the canvas colour using the R (for red) and Y (for yellow) keys. Any key other then "R", "r", "Y" and "y" will have no effect on the colour, but will print its value to the Shell. A program accepting keyboard input should always cope where appropriate with both an uppercase and lowercase letter.

Note that `keycode` is an integer.

```
"""Chapter 2, Code Listing 2.36,  Colour changing with Keys"""
from tkinter import *

class CanvasColour:
    def __init__(self, parent):
        c_label = Label(parent, text = "Key example:")
        c_label.grid(row = 0, column = 0, sticky = N)

        self.canvas = Canvas(parent, width = 300, height = 300, bg = "gray")
        self.canvas.grid(row = 0, column = 1)

        parent.bind("<Key>", self.change_colour)

    def change_colour(self, event):
        print(type(event.keycode))
        print(event.keycode)
        if event.char == "Y" or event.char == "y":
            self.canvas.configure(bg = "yellow")
        if event.char == "R" or event.char == "r":
            self.canvas.configure(bg = "red")
#main routine
if __name__ == "__main__":
    root = Tk()
    keys = CanvasColour(root)
    root.mainloop()
```

Code Listing 2.36

Quitting from within your program

Up to now we have quit our programs by clicking the red button at the top of our root window. You might like to consider adding a quit button to your programs as well as an appropriate key binding. The actual command we will use is:

```
root.destroy()
```

In our programs this may very well be phrased as:

```
self.parent.destroy()
```

When called on a widget this means "Destroy this and all descendant widgets". When called on our root widget it destroys the root and all other widgets in it. Importantly it leaves IDLE still going.

DO NOT use `root.quit()` if you are working in IDLE. This will quit the Tcl interpreter. Since IDLE is written in Tkinter and the shell window is, itself, a widget, it will crash (or at least become very confused).

An example of a quit button is shown in code listings 2.36 and again in 2.37.

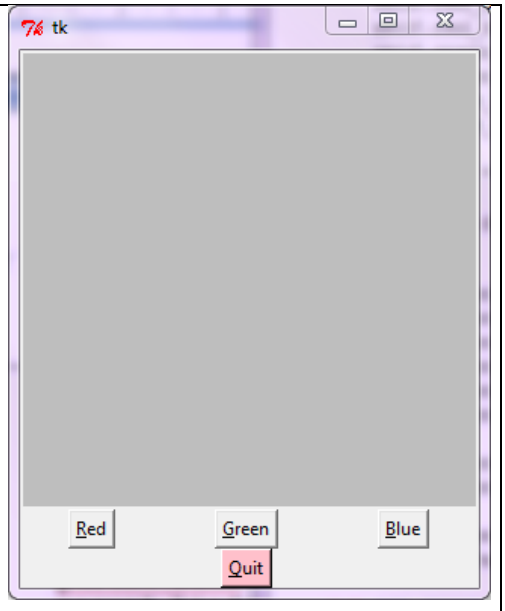
Background colours revisited example

Adapting Code Listing 2.21 from earlier, let's add key bindings so that "R", "G", "B", "r", "g" and "b" will produce the same results as clicking on the buttons. It is polite in this situation to indicate the key option by underlining the appropriate letter on the button's text, if possible. (Sorry Mac users - your Buttons will be Mac style). To do this, the first letter of any string is underlined using `underline = 0`.

This code also shows a quit button with additional key bindings. The program can now be exited if the user clicks on this button, types the letter 'q' or 'Q', or, of course, in the usual way (clicking the red button in the top corner of the window.)

Note: One potential frustration is that the original callback function assigned to the command option only took one argument - `self`, but any method responding to a binding must also accept the event as an argument. It makes sense to be able to use the same method for each event source, as the same behaviour is expected (and it always rings alarm bells for programmers if we find ourselves repeating code). This is easily solved by including the event as an optional argument and setting its default value to `None`.

```
"""Chapter 2, Code Listing 2.37, Background colours again"""
from tkinter import *

class BackgroundColourButtons:

    def __init__(self, parent):

        self.canvas = Canvas(parent, width = 300,
               height = 300, bg = "gray")
        self.canvas.grid(row=0, columnspan = 3)

        self.parent = parent

        red_btn = Button(parent, text = "red",
               underline = 0, command = self.set_red)
        red_btn.grid(row = 1, column = 0)
        green_btn = Button(parent, text = "green",
               underline = 0, command = self.set_green)
        green_btn.grid(row = 1, column = 1)
        blue_btn = Button(parent, text = "blue",
               underline = 0, command = self.set_blue)
        blue_btn.grid(row = 1, column = 2)

        quit_btn = Button(parent, text = "Quit",
               underline = 0,
               command = self.quit_program)
        quit_btn.grid(row = 2, column = 1)
```




```
        parent.bind('r', self.set_red)
        parent.bind('g', self.set_green)
        parent.bind('b', self.set_blue)
        parent.bind('R', self.set_red)
        parent.bind('G', self.set_green)
        parent.bind('B', self.set_blue)

        self.parent.bind("q", self.quit_program)
        self.parent.bind("Q", self.quit_program)


    def set_red(self, event = None):
        self.canvas.configure(bg = "red")

    def set_green(self, event = None):
        self.canvas.configure(bg = "green")

    def set_blue(self, event = None):
        self.canvas.configure(bg = "blue")

    def quit_program(self, event = None):
        self.parent.destroy()


#main routine
if __name__ == "__main__":
    root = Tk()
    bg_colours = BackgroundColourButtons(root)
    root.mainloop()
```

Code Listing 2.37

Choosing which widget to bind to

In this example we have bound the key presses to the root widget. As usual this is called parent inside the BackgroundColourButtons class. Binding to the root widget means that so long as the Tk (or root) window is the active window then the bound key presses will work as expected. In other situations we might bind a key press to a particular widget. For example we might want a user to trigger a callback method by pressing return after typing into an Entry widget. In such a case we **only** want pressing return to trigger the callback **if** the Entry widget has the focus at the time so we would bind the callback to the Entry widget. Indeed we could then bind "<Return>" to a different callback and a different Entry widget. See section 2.21 for further discussion.

Another Pattern

In Code Listing 2.37 we have 3 methods performing similar tasks (configuring a bg value). We can consolidate these into one method, shown in Code listing 2.38. This method needs to check the event type. If the event was a button click (event type '4'), `col` is set to the text of that button widget. If the event was a keystroke (event type '2') , the key pressed is stored as a char using `event.char` then `col` is set to the corresponding colour name for that letter. Regardless of how `col` got its value, the last line of the method uses `col` to set the background colour.

Although the Code Listing 2.38 code ends up being slightly longer than for Code Listing 2.37 while performing the same task, it shows a pattern that could prove useful in some situations. Note that command is no longer used. Instead the Button widgets are bound to the callback if clicked (i.e. if the event object referred to as "<Button-1>" has been created).

```
"""Chapter 2, Code Listing 2.38, Background colours with widget events"""
from tkinter import *

class BGButtons:

    def __init__(self, parent):
        self.canvas = Canvas(parent, width = 300, height = 300)
        self.canvas.grid(row=0, columnspan = 3)

        self.red_btn = Button(parent, text = "Red", underline = 0)
        self.red_btn.grid(row = 1, column = 0)
        self.green_btn = Button(parent,text = "Green", underline = 0)
        self.green_btn.grid(row = 1, column = 1)
```



```
        self.blue_btn = Button(parent, text = "Blue", underline = 0)
        self.blue_btn.grid(row = 1, column = 2)

        quit_btn = Button(parent, text ="Quit", underline = 0, bg = "pink",
                          command = self.quit_program)
        quit_btn.grid(row = 2, column = 1)

        # Bindings for key presses
        parent.bind("R", self.change_bg)
        parent.bind("G", self.change_bg)
        parent.bind("B", self.change_bg)
        parent.bind("r", self.change_bg)
        parent.bind("g", self.change_bg)
        parent.bind("b", self.change_bg)

        self.parent.bind("q", self.quit_program)
        self.parent.bind("Q", self.quit_program)

        # Bindings for button clicks
        self.red_btn.bind("<Button-1>", self.change_bg)
        self.green_btn.bind("<Button-1>", self.change_bg)
        self.blue_btn.bind("<Button-1>", self.change_bg)

    def change_bg(self, event):
        col = "black"
        print(event.type)

        if event.type == '4':
            col = event.widget['text']
        elif event.type == '2':
            char = event.char
            if char =='R' or char=='r':
                col = "red"
            elif char =='G' or char == 'g':
                col = "green"
            else:
                col = "blue"

        self.canvas.configure(bg = col)

    def quit_program(self, event = None):
        self.parent.destroy()
# main routine
if __name__ == "__main__":
    root = Tk()
    bg_colours = BGButtons(root)
    root.mainloop()
```

Code Listing 2.38

Task 9: Add Key Bindings

- Add a key binding to the Times Table task so that pressing the enter key has the same effect as clicking the Check Now button
- Add key bindings to the Stripes task

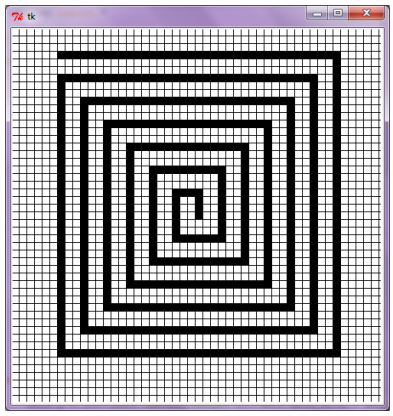
Task 10: Revisit the Colour Sampler Example

Remove the GO button and have the canvas change colour when the user presses the enter/return key in the Entry widget.

Task 11: Random Colour

Adapt one of your colour Changing classes so that when a mouse enters (or, if you prefer, leaves) the canvas, the background colour of the canvas is set to a new random colour. See the earlier section Colour in tkinter for helpful code.

Task 12: Trail Maker



Create a canvas that shows a grid (use two loops to draw the grid lines at regular intervals). Fill in the centre "square" as a starting point. Allow the user to fill in one square at a time using the arrow keys.

Options / extension

Can you expand it to allow the user to skip squares?

Can you add a facility for a mouse click to fill in a square?

Colour over mistakes?

What about letting them change color?

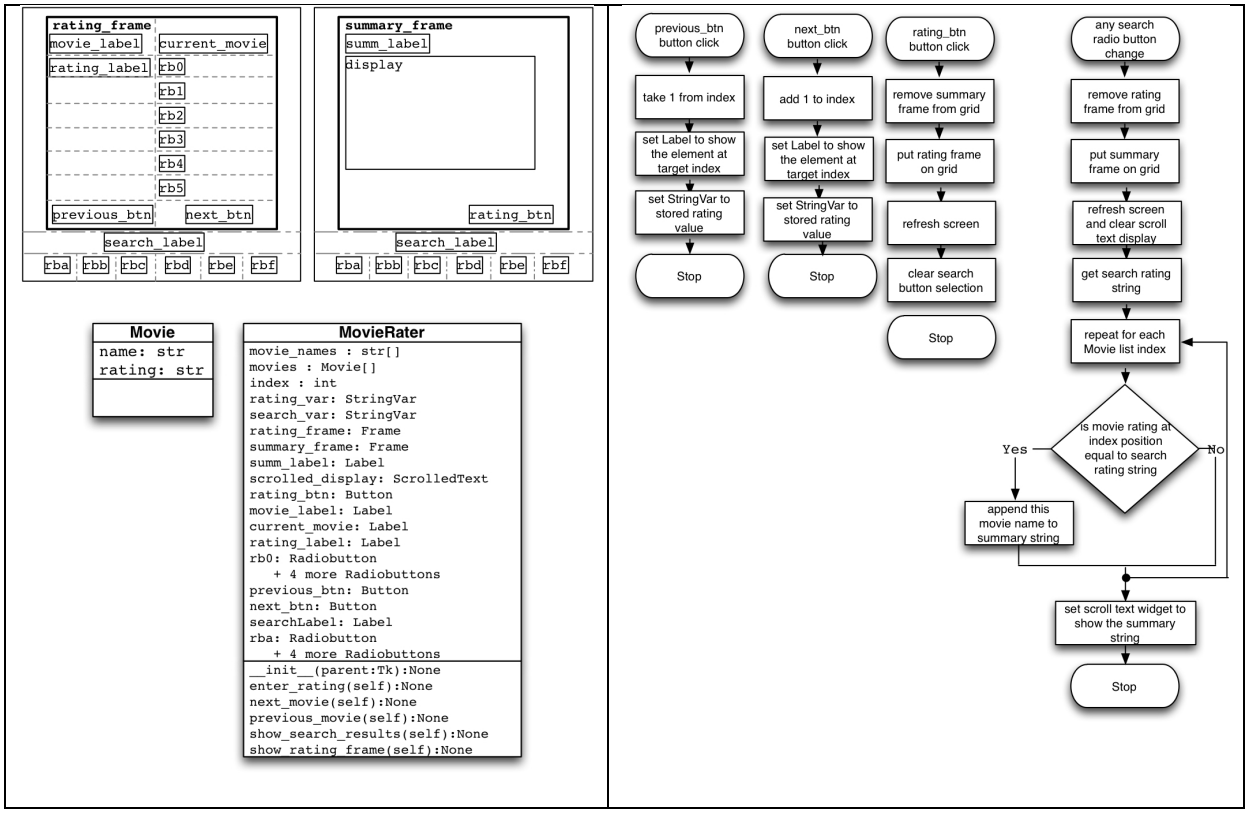
Repeat the task above with button motion (mouse drag) instead of key strokes. Can you still stay within the lines?

Task 13: Freestyle Drawing Program

Write a program that allows the user to draw freestyle onto a blank canvas by dragging the mouse. Add widgets to allow them to pick the colour and width of the canvas item (we suggest an oval) they are drawing.

2.19. Minimal Solution for Movie Rater

Guided Task from section 2.15 - Movie Rater program




```
"""Chapter 2, Code Listing 2.39,  Movie rater solution"""
from tkinter import *
from tkinter.scrolledtext import *

class Movie:
   def __init__(self, name):
         self.name = name
         self.rating = "No Rating"  # default setting

class MovieRater:

   def __init__(self, parent):
      # make a list of Movie objects
      movie_names = ["The Hobbit", "SkyFall", "Any movie with Adam Sandler in it",
         "Monsters Inc", "The Incredibles", "King Kong", "The Avengers",
         "Thor", "Captain America"]

      self.movies = []
      for m in movie_names:
            movie = Movie(m)
            self.movies.append(movie)

      # index will be used for keeping track of current movie in movies list
      self.index = 0

      #GUI set up
      self.rating_var = StringVar()
      self.rating_var.set("No Rating")  # initial setting rating radiobuttons

      self.search_var = StringVar()
      self.search_var.set("*")  # initial setting search radiobuttons : no valid value, so none set
```

```
        self.rating_frame = Frame(parent, width = 350, height = 320, bg = "chartreuse")
        self.rating_frame.grid_propagate(0) # to reserve space required for frame
        self.summary_frame = Frame(parent, width = 350, height = 320, bg = "orchid")
        self.summary_frame.grid_propagate(0)

        # widgets for summary_frame

        # This is being placed in the same position as the rating frame
        # The call to show_rating_frame at the end of init will sort this out.
        self.summary_frame.grid(row = 0, columnspan = 6, padx = 50)

        self.summ_label = Label(self.summary_frame, bg = "orchid", wraplength = 280)
        self.scrolled_display = ScrolledText(self.summary_frame, width = 30, height = 10)
        self.rating_btn = Button(self.summary_frame, text = "Back to Rating",
                                                    command = self.show_rating_frame)

        self.summ_label.grid(row = 0, column = 0,padx = 20, sticky = W)
        self.scrolled_display.grid(row = 1, column = 0, sticky = W, padx = 20,pady = 20)
        self.rating_btn.grid(row = 2, column = 0, sticky = SE, padx = 20, pady = 20)

        # widgets for rating_frame

        # This is being placed in the same position as the summary frame
        # The call to show_rating_frame at the end of init will sort this out.
        self.rating_frame.grid(row = 0, columnspan = 6, padx = 50)

        movie_label = Label(self.rating_frame, text = "Please Rate: ", bg = "chartreuse")
        movie_label.grid(row = 0, column = 0, sticky = W, padx = 25, pady = 10)

        self.current_movie = Label(self.rating_frame,  wraplength = 100, height = 3,
                                   text = self.movies[self.index].name, bg = "chartreuse")
        self.current_movie.grid(row = 0, column = 1, sticky = W)

        rating_label = Label(self.rating_frame, text ="Your rating:", bg = "chartreuse")
        rating_label.grid(row = 1, column = 0, sticky = W, padx = 25)
        self.rb0 = Radiobutton(self.rating_frame, text = "No Rating", value="No Rating",
                variable = self.rating_var, command = self.enter_rating, bg = "chartreuse")
        self.rb1 = Radiobutton(self.rating_frame, text = "Forget it", value="Forget it",
                variable = self.rating_var, command = self.enter_rating, bg = "chartreuse")
        self.rb2 = Radiobutton(self.rating_frame, text = "2", value = "2",
                variable = self.rating_var, command = self.enter_rating, bg = "chartreuse")
        self.rb3 = Radiobutton(self.rating_frame, text = "3", value ="3",
                variable = self.rating_var, command = self.enter_rating, bg = "chartreuse")
        self.rb4 = Radiobutton(self.rating_frame, text = "4", value ="4",
                variable = self.rating_var, command = self.enter_rating, bg = "chartreuse")
        self.rb5 = Radiobutton(self.rating_frame, text = "Must See", value = "Must See",
           variable = self.rating_var, command = self.enter_rating, bg = "chartreuse")
        self.rb0.grid(row = 1, column = 1, sticky = W)
        self.rb1.grid(row = 2, column = 1, sticky = W)
        self.rb2.grid(row = 3, column = 1, sticky = W)
        self.rb3.grid(row = 4, column = 1, sticky = W)
        self.rb4.grid(row = 5, column = 1, sticky = W)
        self.rb5.grid(row = 6, column = 1, sticky = W)

        self.previous_btn = Button(self.rating_frame, text = "Previous", underline = 0,
                                                        command = self.previous_movie)
        self.previous_btn.grid(row = 7, column = 0, sticky = SW, padx = 50, pady = 10)
        self.next_btn = Button(self.rating_frame, text="Next", underline = 0, width = 7,
                                                          command = self.next_movie)
        self.next_btn.grid(row = 7, column = 1, sticky = SW, padx = 50, pady = 10)

        # widgets for searching
        self.search_label = Label(parent, text = "Search for movies with a rating of:")
        self.search_label.grid(row = 1, columnspan = 6)

        self.rba = Radiobutton(parent, text = "No Rating",value = "No Rating",
                    variable = self.search_var, command = self.show_search_results)
        self.rbb = Radiobutton(parent, text = "Forget it", value = "Forget it",
                    variable = self.search_var, command = self.show_search_results)
        self.rbc = Radiobutton(parent, text = "2", value ="2", variable = self.search_var,
                                               command = self.show_search_results)
        self.rbd = Radiobutton(parent, text = "3", value ="3", variable = self.search_var,
                                               command = self.show_search_results)
        self.rbe = Radiobutton(parent, text = "4", value = "4", variable = self.search_var,
                                                  command = self.show_search_results)
        self.rbf = Radiobutton(parent, text = "Must See", value = "Must See",
                    variable = self.search_var,command = self.show_search_results)
```

```
        self.rba.grid(row = 2, column = 0, padx = 5)
        self.rbb.grid(row = 2, column = 1, padx = 5)
        self.rbc.grid(row = 2, column = 2, padx = 5)
        self.rbd.grid(row = 2, column = 3, padx = 5)
        self.rbe.grid(row = 2, column = 4, padx = 5)
        self.rbf.grid(row = 2, column = 5, padx = 5)

        # This removes the summary frame and ensures the rating frame is in place on the grid
        self.show_rating_frame()


    def enter_rating(self):
        """ store movie rating in Movie object at current index of movies list """
        self.movies[self.index].rating = self.rating_var.get()


    def next_movie(self):
        self.index += 1 # keep a track of which Movie in the list we are looking at
        #update movie name
        self.current_movie.configure(text = self.movies[self.index].name)
        # update the rating radiobutton showing as selected
        self.rating_var.set(self.movies[self.index].rating)


    def previous_movie(self):
        self.index -= 1 # keep a track of which Movie in the list we are looking at
        #update movie name
        self.current_movie.configure(text = self.movies[self.index].name)
        # update the rating radiobutton showing as selected
        self.rating_var.set(self.movies[self.index].rating)

    def show_search_results(self):
        """ Removes the rating frame and grids the summary frame. Establishes the rating
        being searched and clears the scrolled display.The method then searches the
        list of movies for those with a rating matching the rating being searched for and
        inserts each title into the scrolled display"""
        self.rating_frame.grid_remove()
        self.summary_frame.grid()
        root.update_idletasks() # necessary on some Op. Systems to refresh the screen properly

        search_rating = self.search_var.get()

        self.scrolled_display.delete('1.0', END) # clear any previous content

        self.summ_label.configure(text = "You have given the following movies a rating of " +
                                search_rating + ": ", bg = "plum")
        for m in self.movies:
              if m.rating == search_rating:
                  self.scrolled_display.insert(END, m.name + "\n" )

    def show_rating_frame(self):
        """ Toggles back to the rating frame """
        self.summary_frame.grid_remove()
        self.rating_frame.grid() # return rating_frame to its original grid position
        root.update_idletasks() # to refresh the screen properly
        self.search_var.set("*")

#main routine

if __name__ == "__main__":
    root = Tk()
    root.title("Movie Ratings")
    rater = MovieRater(root)
    root.mainloop()
```

Code Listing 2.39

2.20. Focus traversal

Focus Traversal

Some of you may already have noticed that it is not necessary to reach for your mouse in order to click on a Button widget or use your mouse to focus on an Entry widget in order to type something into it. By default tkinter enables tab traversal (also called focus traversal) to certain kinds of widgets.

Whether or not a widget can be reached by pressing the tab key on your keyboard is determined by the **takefocus** option which was discussed briefly in section 2.4. If this option is set to True the widget can be reached by pressing the tab key, if set to False it is ignored. The takefocus option of Entry, ScrolledText, Button, Radiobutton, Checkbutton and Scale widgets is set to True by default.

- Once reached, the Entry and ScrolledText widgets can be typed into.
- Once the Button, Radiobutton and Checkbutton widgets are reached by tabbing, pressing the **space bar** on your keyboard is the same as clicking the mouse, i.e. Radiobuttons and Checkbuttons will be selected or deselected and any callback associated with the widget using the command option will be executed.
- Once the Scale widget is reached, pressing the arrow keys on your keyboard is the same as moving the slider with a mouse drag, i.e. a new value is set and any callback associated with the widget using the command option will be executed.
- The order in which widgets are traversed is the order in which they were created.

Note: Since ScrolledText widgets can themselves contain tab characters you will need to use control and tab together to exit the widget .

This default tab traversal behaviour for buttons breaks when event binding is introduced since it only works with command. "<Button-1>" binds specifically to a mouse press NOT a key press. The user can still tab to a widget with this binding but pressing the space bar will not execute the callback unless you have specifically bound that as well. See code listing 2.40 for an example of how to do this.

Focus traversal does not work reliably with MacOS since Mac's button styles completely override tkinter's. You may still tab to a button and press the space bar and the right thing may happen behind the scenes BUT you won't see any indication that the button has been given focus. The resulting behaviour is confusing and best avoided.

Choosing which widget to bind to

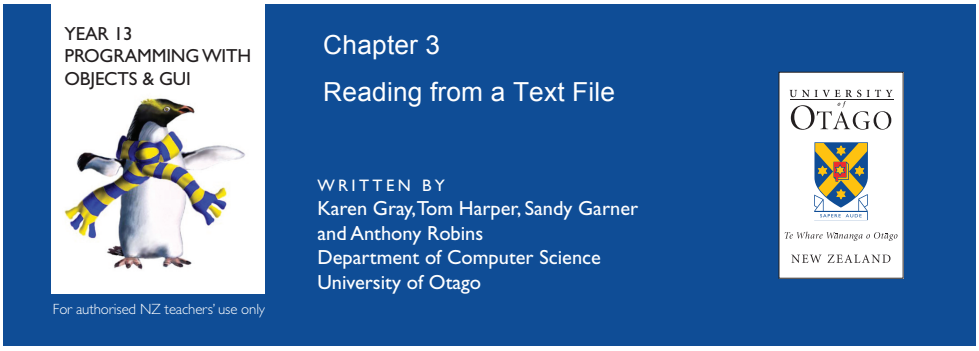
If your key binding is intended to work in conjunction with tab traversal you may bind to the particular widget. The user must then tab to the particular widget (a Canvas, Button, Label, Checkbox etc) and, once it has focus, press the bound key (e.g. the space bar) in order for the callback to be executed. This is only suitable if the user needs to engage with all or most of the callback associated widgets in a particular order, e.g. a form.

If you are not relying on tab traversal but would rather the user be able to press a key and have the callback be executed regardless of where the focus may currently be, you will need to bind to the root widget. This is particularly suitable in a situation where several (or even many) widgets can be 'activated' in any order, i.e. tab traversal would require too many key presses.

In the code listed in 2.38, users can still tab to a button but pressing the space bar will no longer achieve the same result as clicking the mouse. The default tab traversal behaviour can be restored by adding the code in bold shown in code listing 2.40.


```
"""Chapter 2, Code Listing 2.40, Background colours with widget events and a Quit button"""
from tkinter import *

class BGButtons:

    def __init__(self, parent):

        self.canvas = Canvas(parent, width = 300, height = 300)
        self.canvas.grid(row=0, columnspan = 3)

        self.parent = parent

        red_btn = Button(parent, text = "Red", underline = 0)
        red_btn.grid(row = 1, column = 0)
        green_btn = Button(parent,text = "Green", underline = 0)
        green_btn.grid(row = 1, column = 1)
        blue_btn = Button(parent, text = "Blue", underline = 0)
        blue_btn.grid(row = 1, column = 2)

        quit_btn = Button(parent, text ="Quit", underline = 0,
                          command = self.quit_program)
        quit_btn.grid(row = 2, column = 1)

        self.parent.bind("R", self.change_bg)
        self.parent.bind("G", self.change_bg)
        self.parent.bind("B", self.change_bg)
        self.parent.bind("r", self.change_bg)
        self.parent.bind("g", self.change_bg)
        self.parent.bind("b", self.change_bg)

        self.parent.bind("q", self.quit_program)
        self.parent.bind("Q", self.quit_program)

        red_btn.bind("<Button-1>", self.change_bg)
        green_btn.bind("<Button-1>", self.change_bg)
        blue_btn.bind("<Button-1>", self.change_bg)

        red_btn.bind("<space>", self.change_bg)
        green_btn.bind("<space>", self.change_bg)
        blue_btn.bind("<space>", self.change_bg)


def change_bg(self, event):
        if event.type == '4' :
            col = event.widget['text']
        elif event.type == '2':
            char = event.char
            if char =='R' or char=='r':
                col = "red"
            elif char =='G' or char == 'g':
                col = "green"
            elif char =='B' or char == 'b':
                col = "blue"
            else:
                col = event.widget['text'] # the space bar has been pressed over a colour button
        self.canvas.configure(bg = col)
    def quit_program(self, event = None):
        self.parent.destroy()
#main routine
if __name__ == "__main__":
    root = Tk()
    bg_colours = BGButtons(root)
    root.mainloop()
```

Code Listing 2.40



3.1. Files

Data for a program has to come from somewhere. So far in this book, data used to create objects has often been hardcoded within the program e.g.

```
p1 = Person("Ana", "Female", 1987)
```

Alternatively the data has been provided by the user during the program's execution e.g.

```
name = input("Enter the name of the person")
gender = input("What is the gender of the person")
age = eval(input("What is the year of birth of the person"))

p1 = Person(name, gender, age)
```

or from a GUI.

Data might come from an input port (keyboard, mouse, attached sensor), or from the state of an object (animation, GUI), or from some sort of storage device. A commonly used storage option is a text file.

In this chapter, we are going to access data from files and use it first in a text based program, then in a GUI program.

The files to be accessed can be created by any text editor. A likely purpose of reading a data file might be to be able to scroll through it, forwards and backwards, present it and search it using a GUI.

For the programs in this chapter, you will need text files in appropriate formats saved in the same directory as your Python file. If the file you wish to open is not in the same folder as your Python file, you will need to be able to navigate through your directory structure to access the file. We have listed relatively short text file examples for efficiency's sake, but the great advantage of using files is that you can work with large amounts of data.

3.2. Buffers and streams

When you are using a word processor and you open a document file, the text data is transferred from the region in secondary storage where the file is stored into a region in RAM called a buffer. You can think of a buffer as an interface between a program and the file. All interactions with the file (read and write operations) are performed on a buffer. When the buffer is properly closed then the file on hard disk is updated. If we did not use an intermediate buffer in RAM (between the user and the file) then we would have to interact with the file on secondary storage directly. This would be much slower – as accessing secondary storage is slow. We can think of a buffer as a temporary representation of the file which exists during a program's execution.

The read- or writable data within the buffer is called a *stream* because it is a continuous flow (sequence) of data.

To open a file named `"text.txt"`, we apply the function `open`. This function belongs to the `io` module which is automatically available. It will search for and locate the target file in the current working directory and return a text stream (sequence of text data). This is a *readable* `TextIOWrapper` object from a buffer. We will identify the (stream) object with the reference `my_file`.

```
my_file = open("text.txt")
```

Or equivalently: (Where `"r"` means readable)

```
my_file = open("text.txt","r")
```

If the file does not exist, you could include an error message in the code to report the fact.

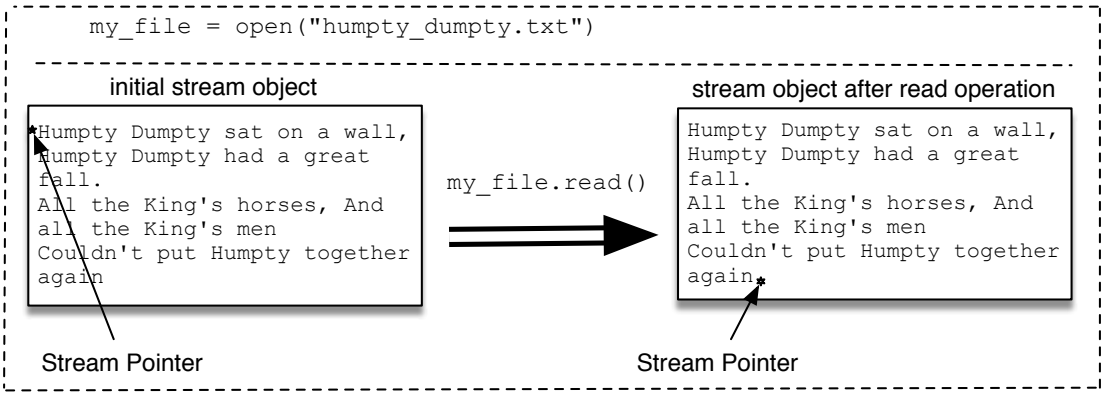
```
try:
    my_file = open("text.txt")
except IOError:
    print("File does not exist")
else:
    # carry on and read the file
```

3.3. Reading from the file

Information is most simply read in from a file as text (a string, or list of strings). If the file exists in the current working directory then we may read its entire string (`str`) content by invoking the `read` method on `my_file`:

```
text = my_file.read()
```

Before `read` is invoked, a pointer is pointing at the beginning of the `TextIOWrapper` object/ stream. After `read` is invoked, this pointer is pointing at the end of the stream and any successive calls to `read` would return the empty `str`.



Task 1

Open the file `"hens.txt"`. Read the file in to a string, print it, replace each occurrence of `"/"` with a space `" "`, then print the new string. (Hint: You can use the string method `replace.)`

Just write this small program in the main routine.

input: hens.txt	Output expected
It/can/boil/an/egg/at/30/paces,/whether/you/want/it/to/or/not,/actually,/so/I've/learned/to/stay/away/from/hens.	It can boil an egg at 30 paces, whether you want it to or not, actually, so I've learned to stay away from hens.

Task 2

Create a function `one_per_line` that takes an input parameter representing a filename, reads in from the file and displays each word on a new line. Remember the newline character is `"\n"`. This function should be called from the main routine.

input: names.txt	Output expected
Abigail Bertha Camelia Daisy Eleanor Fiona Grace Harriet Isobel Jenna Katrina Liana	Abigail Bertha Camelia Daisy Eleanor Fiona Grace Harriet Isobel Jenna Katrina Liana

3.4. Data formatting

Data in files needs to be structured in a known and consistent format so that we can develop an appropriate mechanism to turn data from files into information in our programs. The simplest structure is one data item or set of related items per file line.

To read in lines from a file separately we use the method `readlines()`. This reads each line of the file and stores it as an element in a list. The elements can then be processed individually.

```
my_file = open('titles.txt', 'r')
lines = my_file.readlines()  # lines will be a list of strings
for line in lines:
    print(line)      # print each line
```

Code Listing 3.1

The string function `split()` is very useful for extracting individual items from a string which represents a line from the file. Split by default creates a list of strings separated by white space. White space is the unmarked space between characters which may be made up of tabs or a space character(s) or a carriage return \n.

White space may not be a specific enough separator, since white space may be part of a data item itself:

```
12 Mackeroy Street Auckland New Zealand
```

A comma or semi-colon can serve as an appropriate separator:

```
12 Mackeroy Street, Auckland, New Zealand
```

The split function with a comma as an input parameter i.e. `split(',')` will break the string into a list of substrings separated by commas. As we process every line, we use the function `split` to extract items separated by commas into a list.

```
my_file = open('addresses.txt', 'r')

lines = my_file.readlines()

for line in lines:
    words = line.split(',')
    street = words[0]
    town = words[1]
    country = words[2]
    print(street)
    print(town)
    print(country)
```

Code Listing 3.2

input: addresses.txt	output from Code Listing 3.2
12 Mackeroy Street,Auckland,New Zealand 144 Queens Drive,Dunedin,New Zealand 13 Riley Drive,Perth,Australia	12 Mackeroy Street Auckland New Zealand 144 Queens Drive Dunedin New Zealand 13 Riley Drive Perth Australia

Notice the line between each output set. We can stop this happening by removing all carriage return characters. Include the statement

```
    line = line.replace('\n','')
```

as the first statement of the `for` loop to achieve this.

For a file representing year, population, GDP Billions, separated by commas we could read it like this

```
my_file = open('GDP_info.txt', 'r')

lines = my_file.readlines()

for line in lines:
    line = line.replace('\n','')  # or the carriage return will show up as another line
    words = line.split(',')
    str_form = 'Year: {} Population: {} GDP: {}'
    s = str_form.format(words[0],words[1],words[2])
    print(s)
```

Code Listing 3.3

input: GDP_info.txt	output from Code Listing 3.3
1996,2543466,45.7 2000,2934565,48.3 2004,3124354,49.3	Year: 1996 Population: 2543466 GDP: 45.7 Year: 2000 Population: 2934565 GDP: 48.3 Year: 2004 Population: 3124354 GDP: 49.3

Task 3

Create a program which reads tide data from a file which is delimited by white space. Your job is to list the first low tide beside each date. The file input listing has a header row which describes the data - see this in the input listing.

input: tides.txt	Output expected
Day,Date,High,Low,High,Low Tu,1-Jan-13,5:57,12:09,18:17, We,2-Jan-13,,0:28,6:41,12:53,19:04 Th,3-Jan-13,,1:14,7:28,13:41,19:55 Fr,4-Jan-13,,2:04,8:20,14:33,20:50 Sa,5-Jan-13,,2:57,9:16,15:28,21:47 Su,6-Jan-13,,3:55,10:16,16:25,22:47 Mo,7-Jan-13,,4:56,11:16,17:24,23:46 Tu,8-Jan-13,,5:58,12:15,18:21, We,9-Jan-13,0:44,6:57,13:11,19:18 Th,10-Jan-13,1:41,7:54,14:06,20:14 Fr,11-Jan-13,2:36,8:49,14:59,21:09 Sa,12-Jan-13,3:30,9:42,15:52,22:03 Su,13-Jan-13,4:22,10:34,16:45,22:57 Mo,14-Jan-13,5:14,11:26,17:38,23:50	Date Low 1-Jan-13 12:09 2-Jan-13 0:28 3-Jan-13 1:14 4-Jan-13 2:04 5-Jan-13 2:57 6-Jan-13 3:55 7-Jan-13 4:56 8-Jan-13 5:58 9-Jan-13 6:57 10-Jan-13 7:54 11-Jan-13 8:49 12-Jan-13 9:42 13-Jan-13 10:34 14-Jan-13 11:26

3.5. Data conversion

Since all data is read in from file as a string we may need to convert it to an appropriate type to process it further. For example, to calculate the average of a list of numbers, we need to convert each "number string" into an `int` so we can add them together.

```
myfile = open('numbers.txt')
line = myfile.read()
numbers = line.split()
total = 0
count = len(numbers)
for number in numbers:
    try:
        total += int(number)
    except ValueError:
        print("item \"" + number + "\" not a number")
        count -= 1
print('average is ', format(total/count))
```

Code Listing 3.4

input: numbers.txt	output from Code Listing 3.3
11 4 5 6 3 6 1 4 3	average is 4.777777777777778
11 4 5 6 3 6 a 4 3	item "a" not a number average is 5.25

3.6. Using file data with your GUI programs

Any time you have got data in to your program by a hard-coded list, or by asking the user to type in values, you can now use a file. Code Listing 3.5 shows how easy it is to adapt the MovieRater program from chapter 2 so that instead of having the list of movies hardcoded in the program, it reads them from a file.

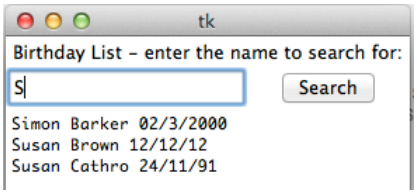
```
# make a list of Movie objects original version, Code Listing 2.39, hardcoded values for movie_names
movie_names = ["The Hobbit", "SkyFall", . . .
                "The Incredibles", "The Avengers", "Thor", "Captain America"]
```

```
# make a list of Movie objects using a text file - assumes there is one movie title per line in the file
my_file = open("movies.txt")
movie_names = my_file.readlines()
```

3.7. Exercises

Task 4: Birthday List

Using a text file containing a list of names and birthdays, set up a GUI with a label to prompt the user, an entry field where the 'search text' is written, a button to search for names which match the search string and a ScrolledText widget to insert the search results. Implement the callback function to search for names and birthdays.



Task 5: Multi choice Quiz

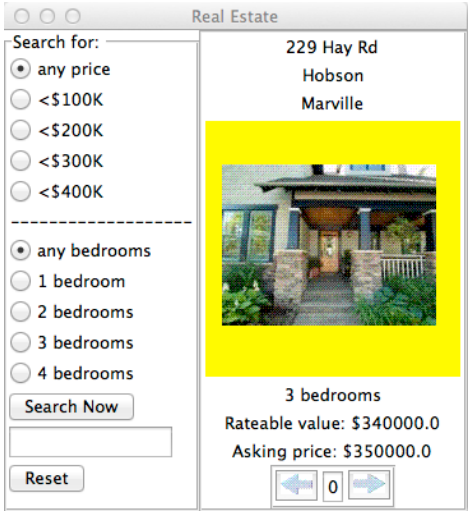
Adapt your multi choice quiz program so it obtains its data from a file rather than has it hard-coded in the program.

Task 6: Real Estate

Produce a program that allows the user to scroll through a list of house images.

Design your own specifications for a Real Estate GUI, which accesses a text file with information about houses, including the correct paths to image file names.

The program should provide search functions for the user. One possible "look" is shown:



3.8. Writing to a file (optional)

Instead of reading from the file, we may wish to write to the file.

Open

First we must open the file in a mode which allows the file to be written to.

This can be achieved by changing the read/write mode.

If `"text.txt"` does not exist, then `"w"` creates a new file. If it does exist, it will be replaced (any data in it will be lost - be careful).

`"r+"` allows both reading and writing to an existing file. Data will be appended.

```
my_file = open("text.txt", "r+")
```

`"a+"` will create the file if it does not exist already. If it does exist, it will append the new data

```
my_file = open("text.txt", "a+")
```

`"w+"` is similar to `"a+"` but replaces the data rather than appends it (any existing data will be lost - be careful).

```
my_file = open("text.txt", "w+")
```

Write

Once the file is open, we can write string data to the file using the method `write`.

```
my_file.write('The cat ')
my_file.write(' sat on the hat')
```

Close

When we are finished with the buffer, we must close the buffer to ensure the changes have been made to the file.

```
my_file.close()
```


Appendix

Most of this Workbook is practical, focused on the basic facts and plenty of exercises. But to really understand what is going on you do need to know about certain terms and the underlying structure of Python.

This Appendix sets out some relevant background information. You don't have to read it, but we recommend that you do. If you have a question that arises while progressing through the Workbook, you may find the answer here. You'll also learn about a few built in functions that are useful for learning more about how Python works. Much of it should be revision of things you already know. This Appendix should probably be re-read a couple of times as you progress, it might make more sense each time.

Basics

Python is a **programming language**, with specified syntax and semantics. You can write a Python program on a bit of paper. Python also refers to the tools that you need to install on a computer to write and run Python programs. There are many different versions of Python, but the essential elements are an **interpreter** (a program which runs programs written in the Python language) and the **standard libraries** (which contain the various resources used by programs).

The libraries contain **modules** (which may be grouped together into **packages**).

Modules are files containing Python code (file names end in ".py"), which may include **definitions** (e.g. of **functions** or **classes**) and other statements.

Some of the resources in the libraries are automatically available in all Python programs – these are called "**built in**" resources. They include built in **functions** (e.g. print, sum, id, help), built in **constants** (e.g. True, False, None), built in **types / classes** (e.g. int, list, string).

Other resources in the libraries can be used in a Python program by **importing** the module that contains them. (Importing a module will automatically run any "top-level" code, i.e. code that isn't part of a function or class, more of this later).

Everything is an object

Python programmers like to say that "everything is an object". The number `2` is an object, the list `[2, 4, 6]` is an object, a string is an object, the function `print` is an object. Even a module is an object. Everything is an object – information stored (starting) at a particular address in the computer's memory. We can find that address using the `id` function[4].

```
>>> id(2)
4296853152
>>> id([2, 4, 6])
4319203984
>>> id("Hello")
4336382336
>>> id(print)
4297858368
```

As well as being able to inspect its address in memory, technically speaking everything is an object in that we can assign objects to a variable (or pass them as an argument to a function). In this example we assign a built in function (reference to an object) to our own variable:

[4] All example code in the Appendix is typed directly to the interpreter prompt `>>>` with output following on the next line. To convert to a saved program most function calls like `id` would need to be in a print statement, e.g. `print( id(2) )` .


```
>>> myPrint = print   # myPrint is assigned the value of print, i.e. they refer to the same object
>>> myPrint("Hello world")
Hello world
>>> id(print)
4297858368
>>> id(myPrint)      # as expected myPrint refers to the same object as print, they are aliases
4297858368
```

Objects and object oriented programming

The above technical definition isn't what people usually mean when they talk about objects and object oriented programming (OOP). Generally speaking, an **object** is a way of organising data and the methods that operate on the data into a coherent whole. Objects can be thought of as "entities" or "agents" having **state** (or "properties" – the values of its data) and **behaviour** (the actions performed by its methods), as described at the start of the chapter.

The Python object model / type system

When we talk about objects in this book we mean objects in the general sense just described. **Objects** are made from **classes**. They may have **instance variables** and **methods** (functions), these are called the **attributes** of the class. The values of the variables represent the **state** of the object, and the methods represent the **behaviour** of an object (as above).

Python uses the terms **type** and **class** almost interchangeably[5]. The type of something is the class that it is made from. To say that `2` is of **type** `int` means that the object `2` is made from the **class** `int` (which is itself an object). In the examples below the function type tells us the class that something is made from, and recall that id tells us the address of an object:

```
>>> id(2)         # 2 is an object (at this address)
4296853152
>>> type(2)          # 2 is made from the class int
<class 'int'>
>>> id(int)          # int an object (at this address)
4296606080
>>> type(int)        # int is made from the class type
<class 'type'>
>>> id(type)         # type is an object (at this address)
4296623904
>>> type(type)       # type is made from the class type – itself – tricky! – don't worry about it
<class 'type'>

>>> type(print)      # don't forget that some functions are "built in"
<class 'builtin_function_or_method'>
```

There are various kinds of objects. The two main ones are[6]:

[5] Useful reference:
http://www.cafepy.com/article/python_types_and_objects/python_types_and_objects.pdf

[6] Other kinds of objects include **fundamental objects** and **method objects**. There are two fundamental objects, type and object. The relationship between them is complex. Type is called a "metaclass" and is the type of all types (including itself). Object is the base of all types, where base is the hierarchical parent / superclass (this has to do with object oriented **inheritance** which is beyond the scope of Year 13). You might be relieved to hear that we don't need to worry about fundamental objects in this Workbook! Method objects are objects representing methods and built in functions, we don't need to worry about this either.

(1) **Type objects (class objects):** There is one and only one type object for each class in the program. This includes built in types / classes, the ones that we import, and the ones that we write for ourselves. Type objects serve as a kind of "blueprint" for creating instance objects.

A class is defined by a particular class definition (a piece of code), e.g. the built in class list (which is defined in the libraries), or the classes that we write. A class is also the type of any instances that we create, e.g. `list` is the type of `[2, 3, 4]`. Thus the terms class and type are used almost interchangeably, and type objects are often called class objects. In the code examples in this section you will see that the type function returns a class, e.g. `type(2)` returns `<class 'int'>`.

(2) **Instance objects**: We make instance objects out of class objects, we can make as many as we like (including none). Instance objects are the "typical" objects of OOP, the ones we usually make to represent aspects of the task or its solution (the "entities" or "agents" discussed above).

We can make instance objects:

- by writing them literally in the program code (e.g. `2` is an instance of `int`, `[4, 5, 6]` is an instance of `list`, and `"Time for lunch"` is an instance of `str`),
- by calling the constructor (`__init__` method) for the class as shown in Chapter 1,
- (or by subclassing type objects to make new type objects, this related to **inheritance** which is beyond the scope of Year 13).

We can call methods on objects, as shown in Chapter 1. We can even call methods on objects written literally in the program code:

```
>>> [4, 5, 6].__len__()       # call __len__ method on a list to get its length
3
>>> 3.14.__add__(2)           # call __add__ method on a float to add a value[7]
5.14
>>> (1).__add__(3)        # put an int in ( ) or the . is read as a decimal point & method call fails
4
```

Object names and attributes

We can refer to objects using their names / identifiers, but the names aren't part of the objects themselves (names live in the **namespaces** that define the rules of **scope** for the language). We can have more than one name for the same object (see aliases in Section 1.13).

Recall that the **instance variables** and **methods** are the **attributes** of an object. We can use the dir function to get a list of attributes from any object. The integer `2` has heaps of them!

```
>>> dir(2)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__',
'__floordiv__', '__format__', '__ge__', '__getattribute__',
'__getnewargs__', '__gt__', '__hash__', '__index__', '__init__', '__int__',
'__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__',
```

[7] The actual result printed is 5.140000000000001. Arithmetic with real values is always approximate in programming languages, because we're trying to represent an infinite range of values with finite computer memory. This becomes very important in some applications, like engineering, where we need to structure our calculations to reduce errors, not increase them!


```
'__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__',
'__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__',
'__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__',
'__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__',
'__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__',
'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator',
'real', 'to_bytes']
```

Trying `dir(int)` would get the same list, in this case the attributes of the type / class object are the same as the instance object. (Note that lots of attribute names start and end with single or double underscores, the examples above are doubles, more of this below.) We can find out about any attribute by using the `help` function[8].

```
>>> help(int.__add__)
Help on wrapper_descriptor:

__add__(...)
    x.__add__(y) <==> x+y
```

In this case it tells us that `__add__` is a method with behaviour equivalent to the `+` operator (we used `__add__` in an example just above). Try calling `help` on other attributes, e.g. `__abs__` and `bit_length`. Try `help(int)` – there's plenty to read! Try `help(print)`. (You can even try `help(help)`, but it isn't very – helpful.)

An example

Assume the following module, containing a class definition and some other statements. The class declares one class variable (see Section 1.14) `x`, and one instance variable, `y`.

```
class MyClass:
    x = 5

    def __init__(self, y):
        """ Initialise an instance variable """
        self.y = y

    def printMe(self):
        """ Prints the values of x and y"""
        print("MyClass.x is:", MyClass.x, " self.y is:", self.y)


print("The class variable exists before we make any instances")
print("MyClass.x is:", MyClass.x)
print("The instances share the class variable and have their own instance
variables")
mc1 = MyClass(10)     # make an instance with self.y = 10
mc2 = MyClass(20)     # make an instance with self.y = 20
mc1.printMe()
mc2.printMe()
print("\nLooking at the attributes of the class object")
print( dir(MyClass) )
print("\nLooking at attributes of an instance object")
print( dir(mc1) )
```

Code Listing A.1

If we run this module / program, the following output is produced:

[8] See also http://www.rafekettler.com/magicmethods.html


```
The class variable exists before we make any instances
MyClass.x is: 5
The instances share the class variable and have their own instance variables
MyClass.x is: 5  self.y is: 10
MyClass.x is: 5  self.y is: 20

Looking at the attributes of the class object
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
'__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'printMe', 'x']

Looking at attributes of an instance object
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
'__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'printMe', 'x', 'y']
```

The first part of output is self explanatory, the class variable `x` is shared between all instances. like `_num_people` in Figure 1.20.

Understanding the attributes is trickier – objects get lots of attributes automatically (via inheritance, which is beyond the scope of Year 13). It is worth noticing that the list of attributes for the class object `MyClass` ends with the attributes we have written, the method `printMe` and the variable `x`. Compare that with the attributes of the instance object `mc1`, that list ends with the same attributes, and also `y`, which is an instance variable (and so not part of the class object).

Mutable and immutable objects

Instance objects are either **mutable** or **immutable**. If the state of an object can be changed by operations it is **mutable**. If it can't it is **immutable**. When it looks like we're changing the state of an immutable object we aren't, we're replacing it with a different object.

As we saw in Section 1.13 the data type `list` is mutable, and type `int` is not. Given `x = 1` we cannot change the state of the `int` object `1`, so an operation like `x += 5` replaces the reference stored in `x` with a reference to a different object, the `int` value `6`.

`list` is mutable
an operation on `y`
changes the state of
the object it refers to:

```
>>> y = [1, 2]
>>> id(y)
4358002448
>>> y += [3]
>>> y
[1, 2, 3]
>>> id(y)
4358002448
```

ids are the same

`int` is immutable
an operation on `x`
changes the reference
to a different object:

```
>>> x = 2
>>> id(x)
4296853152
>>> x += 3
>>> x
5
>>> id(x)
4296853248
```

ids are different

`str` is immutable
an operation on `z`
changes the reference
to a different object:

```
>>> z = "hi"
>>> id(z)
4300064392
>>> z += "there"
>>> z
'hithere'
>>> id(z)
4358035080
```

ids are different

More on modules and main

All code belongs to a **module**. When we write files the module is usually based on the name of the python file (module name from file name.py). Code that we write in the interpreter shell belongs to a special module called the **main module** (with the name `__main__`). Using the `dir` function we can find out the attributes of the main module. (Reset the interpreter shell – ^F6 in IDLE – to clear any other stuff you maybe have been working with recently.)

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
```

Let's keep exploring.

```
>>> __name__
'__main__'
>>> __builtins__
<module 'builtins' (built-in)>
```

The `__name__` attribute holds the name of the module, and `__builtins__` is a module containing all of the built-in functions, constants and types / classes (see above). If you have plenty of time on your hands, try `help(__builtins__)` – heaps to read!

In fact, anywhere a Python program "starts" becomes the main module. If we run a Python module / file (e.g. when we F5 a file in IDLE) that module is the main module. All other modules (anything imported) have their names determined by their file names.

For example, consider two very simple modules! The first is in the file hello.py:

```
print("Hello, this is module", __name__)
```

The second is in the file greetings.py (these two files must be in the same directory):

```
import hello
print("Greetings, this is module", __name__)
```

Open the module `hello` (file hello.py) and run it. The output is:

```
Hello, this is module __main__
```

Because we are starting execution in this module, it is the main module. But look what happens when we select and run the module `greetings` (file greetings.py). The output is:

```
Hello, this is module hello
Greetings, this is module __main__
```

There are two things to notice here! (1) Because we are starting execution in this module, greetings.py is the main module (hello has a name based on its file hello.py). (2) The first thing that happens in greetings.py is that it imports hello.py and thus runs the top-level code, creating the first line of output (before proceeding with the print statement that creates the second line of output). Remember that top-level code in a module (code that isn't part of a function or class) is run when the module is imported!

It is often the case that we write lots of useful modules, but when we import them into programs (perhaps only wanting to use parts of them) we *don't want* the top-level code to run. The solution is to use an if statement to check the module name. If the name is `__main__`

then we run the code, otherwise the module is being imported and we don't run the code. Replace hello.py with the following version[9]:

```
if __name__ == "__main__":
    print("Hello, this is module", __name__)
```

Select and run hello.py, the output is:

```
Hello, this is module __main__
```

Now select and run the (unchanged) greetings.py, the output is:

```
Greetings, this is module __main__
```

Just one line of output this time, because the print statement in the imported module hello did not run (because hello was not the main module).

This is the reason why Python programmers (e.g. most of the examples in this workbook) put this test at the start of their main routine / "top-level" code…

```
if __name__ == "__main__":
    # main routine runs only when if is true
```

… so that the main routine doesn't run if the module is being imported, it only runs if we are "starting" a Python program here. (Just for fun try `>>> import this`)

Access to data

As we have already seen (Section 1.3) Python allows **direct access** to instance variables from "outside" the class, for example from the main routine, as shown in Code Listing A.2.

```
class Person:

    def __init__(self, age):
        # An instance variable to hold the person's age
        self.age = age

#main routine
if __name__ == "__main__":
    boris = Person(15)          # construct an instance and set age
    print(boris.age)            # direct access to get age  ==> 15
    boris.age = 16              # direct access to set age
    print(boris.age)            # direct access to get age  ==> 16
```

Code Listing A.2

Output:

```
15
16
```

This direct access is usually considered bad design – access to instance variables should be "managed" by the class to make sure that they can only be used in well designed ways. (This is a difficult requirement to illustrate in our small programs, but it does become really important for complex programs and especially for teams of programmers).

The alternative to direct access is some form of managed access – ensuring that the data and methods of our objects are protected from being used or overwritten in an undesirable way (see also other OO concepts **like information hiding**). Languages vary in the strength

[9] As you're experimenting and changing the files, don't forget to save them before running or the changes will not be picked up.

of their support for managed access. With strong management data is like prized treasure being held in a castle, surrounded by a moat. If people want to see it or change it they need to cross the right drawbridge. Python, on the other hand, has very weak support. There is no moat. Sometimes, if they feel strongly, Python programmers might put up a mildly worded warning sign. This fits with Python's general philosophy: "Let's all treat each other as grown-ups." Critics maintain that such a philosophy can only end badly.

In OO languages that support strong management (like Java) there are usually rules that specify the **visibility** of data (where it can be seen and used). Instance data is made **private** and can only be accessed via methods. By convention, methods which retrieve pieces of data have identifiers prefixed with `get` e.g. `get_x`, `get_height`. Programmers call these methods getters or accessors. By convention, methods which change pieces of data have identifiers prefixed with `set` e.g. `set_x`, `set_height`. Programmers call these methods setters or mutators. Setter methods will usually have a parameter which describes the new value for the data.

The point of `get` and `set` methods is to include code that manages access appropriately. For example, a class representing a person might have an instance variable representing age. If we change the age there might be all sorts of checks that we want to do (is the person now old enough to go to school, or drive a car, or retire, and so on).

A get method might make sure that the information is completely up-to-date before it goes out e.g. a class returning age might want to recalculate it using today's date, in case a birthday has happened since the value was stored.

With direct access there is no way to enforce these checks.

Here's a version of the above program that uses get and set methods for the instance variable:

```
class Person:

    def __init__(self, age):
        # an instance variable to hold the person's age
        self._age = age

    def get_age(self):
        # any code needed to calculate, check or manage the data returned
        return self._age

    def set_age(self, age):
        # any code needed to check or manage the change to this or any related data
        self._age = age

#main routine
if __name__ == "__main__":
    boris = Person(15)        # construct an instance and set age
    print(boris.get_age())    # use getter to get age ==> 15
    boris.set_age(16)         # use setter to set age
    print(boris.get_age())    # use getter to get age ==> 16
```

Code Listing A.3

Output:

```
15
16
```

If the underscore `_age` made the variable **private** then the only way to access it from outside the class would be via the get and set methods. But Python doesn't believe in visibility (there is no private), so the underscore *doesn't make any practical difference at all*, it just **suggests** to programmers that they shouldn't use direct access for this variable.

Try this - A1

Write a class that stores a person's name and age. It should have a method that displays whether a person is eligible to vote but discourages programmers from accessing or changing the age variable from outside the class. (Some of us are sensitive on the subject and don't want to tell!) Include the following doctest:

```
    def display(self):
        """
        >>> c1 = Citizen("Richard", 17)
        >>> c1.display()
        Name : Richard, Age : 17 cannot vote
        >>> c2 = Citizen("Chris", 18)
        >>> c2.display()
        Name : Chris, Age : 18 can vote
        """
```

The property function

Most Python programmers don't use get and set methods, preferring to use other mechanisms to manage access to data (there is a bit of "friendly" debate about this!).

Another approach to management is to use the `property` function. The `property` function in turn uses things that are basically get and set methods, but it hides the method calls, so the syntax looks just like simple direct access.

Code Listing A.4 follows on from the previous examples. In this case `_age` should never be greater than 100, so we want to perform a check whenever setting the value of `_age`. The use of a `property` function allows us to (almost) enforce this by ensuring that the variable can only be used via methods.

```
class Person:

    def __init__(self, age_in):
        self._age = age_in        # (Note 2) – if an age_in greater than 100 is sent the constructor will fail!

    # the following methods cannot return or change _age directly
    # without calling themselves (setting up an infinite loop - bad!)
    # so they use another variable _checked_age which is based on _age

    def get_age(self):
        return "I am " + str(self._checked_age)

    def set_age(self, age_in): # performs a check so that _checked_age cannot be set to greater than 100
        if age_in <= 100:
            self._checked_age = age_in

    # if we include the following line of code then _age can only be used via methods
    #   - any attempt to access the value of _age will call get_age instead
    #   - any attempt to change the value of _age will call set_age instead (including from __init__ )

    _age = property(get_age, set_age)

#main routine
if __name__ == "__main__":
    boris = Person(15)            # construct an instance and set age
    print(boris._age)             # ==> "I am 15"  (Note 1)
    boris._age = 16               # changes age to 16 (Note 2)
    print(boris._age)             # ==> "I am 16"  (Note 1)
    boris._age = 200              # fails the check, age is unchanged (Note 2)
    print(boris._age)             # ==> "I am 16"  (Note 1)
    boris._checked_age = 300      # bypasses the check - possible but silly!
    print(boris._age)             # ==> "I am 300"  (Note 1)
    pania = Person(101)   # construct another instance and attempt to set age
    print(pania._age)     # run time error because the constructor failed the property check)
```

Code Listing A.4

Output:

```
I am 15
I am 16
I am 16
I am 300
AttributeError: 'Person' object has no attribute '_age'
```

In the main routine it looks like we have direct access to the instance variable `_age`. In fact we have indirect access via the `get_age` and `set_age` methods (and another attribute called `_checked_age`) but we never explicitly called the methods as we did in Code Listing A.3. So:

Note 1 these lines look like they are accessing `_age` directly, but they are really getting the output of the `get_age` method

Note 2 these lines look like they are changing the `_age` variable directly, but they are really passing input to the `set_age` method

Code Listing A.5 is another example, which uses a property function with just a getter method to return a value which is derived from two different instance variables. (A second input to the property function, the specification of the setter method as in Code Listing A.4 is optional.)

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    # this method can't directly return _full_name
    # it returns a value (based on other instance variables) which can be treated as if it was _full_name
    def get_full_name(self):
        return "Full: " + self.first_name + " " + self.last_name

    # if we include the following line of code then _full_name can only be used via methods
    # in this case there is no set method in the property so there is no way to set _full_name,
    # but that's fine because its value is always derived from other values
    _full_name = property(get_full_name)

#main routine
if __name__ == "__main__":
   joe = Person("Joe", "Bloggs");
   print(joe.first_name);      # direct access to first_name
   print(joe.last_name);       # direct access to last_name
   print(joe._full_name);      # indirect access to _full_name via get_full_name method
```

Code Listing A.5

Output:

```
Joe
Bloggs
Full: Joe Bloggs
```

For further details on the property function see:
http://www.packtpub.com/article/python-when-to-use-object-oriented-programming

Try this - A2

Go back to the original cake class and consider the instance variable `num_layers`. If someone wished to increase the number of layers a cake object has after instantiation we would need to scale the ingredients accordingly. We also want to ensure that the number of layers stayed within a sensible range (say 1 – 6).

Write a property function for `num_layers` and methods for this task (including a new instance variable called `_checked_value`). The setter method should check to see if the input is in range, if so update the checked value and call `set_ingredients` (which will then use `self.num_layers` as usual, but because of the property this is really the checked value.) The getter method should just return the checked value.

What will be the output for the following main routine?

```
if __name__ == "__main__":
    c1 = Cake()
    print("Layers: ", c1.num_layers)
    print("Butter: ", c1._butter)
    c1.num_layers = 3
    print("Layers: ", c1.num_layers)
    print("Butter: ", c1._butter)
    c1.num_layers = 8
    print("Layers: ", c1.num_layers)
    print("Butter: ", c1._butter)
```

Try this - A3

In the `Person` class from Chapter 1, Task 5, write a `get_email_address` and a `set_email_address` method. The `set_email_address` method should take a parameter representing an email address and ensure that it contains the @ symbol. If it does, an instance variable `_email_address` should be created and assigned this value. If it does not, the instance variable `_email_address` should be created and assigned the value "invalid". Use the properties function to manage and check the email address variable.

In the `print_info` method, if a person's email is valid then print it (e.g. Ursula in the doctest below), if it is invalid don't print it (e.g. Percy). Include the following doctest:

```
    def print_info(self):
        """
        >>> p1 = Person('Ursula', 1972, False)
        >>> p1.email = "Ursula@yahoo.com"
        >>> p1.print_info()
        Ursula will be 40 this year. She is female.
        Email: Ursula@yahoo.com
        >>> p2 = Person('Percy', 1974, True)
        >>> p2.email = "PercyATyahoo.com"
        >>> p2.print_info()
        Percy will be 38 this year. He is male.
        """
```

In the `main` routine, try setting a badly formed email address for one `Person`, then a correctly formed email address for another.

```
Tom will be 30 this year. He is male.
Penny will be 17 this year. She is female.
Email: petronella@hotmail.com
```

Underscores in names

So what's with all the underscores in Python names? In brief:

A single leading underscore e.g. `_name` indicates by convention that this attribute should be thought of as private (not to be accessed from outside the class, see below).

Double leading underscores e.g. `__name` is a stronger version of the above called "name mangling". The `__name` is treated as `_classname_name`, i.e. explicitly an attribute of the current class, so as to avoid confusion with variables in other classes that may have the same name.

Double leading and trailing underscores e.g. `__name__` are used to indicate that the name is a defined part of the language, to make sure that names written by the programmer don't accidentally duplicate / replace such names. We can replace these built-in attributes if we want to though, we do it when we write our own `__init__` (or `__str__`) methods and the like (then we are **overriding** default methods which have been **inherited** – OO terminology which is beyond the scope of Year 13).

A single trailing underscore e.g. `name_` is used by convention to avoid a clash with a Python keyword

There are other rules too – check out the Python style guide – and follow it!:
http://www.python.org/dev/peps/pep-0008/

Namespaces

When we give different things the same names, working out what a particular name means at a particular place in the code can be tricky! A namespace is a mapping from names to objects, or in other words a way of keeping track of what variables mean.

From http://www. diveintopython.net/html_processing/locals_and_globals.html

> "At any particular point in a Python program, there are several namespaces available. Each function has its own namespace, called the local namespace, which keeps track of the function's variables, including function arguments and locally defined variables. Each module has its own namespace, called the global namespace, which keeps track of the module's variables, including functions, classes, any other imported modules, and module-level variables and constants. And there is the built-in namespace, accessible from any module, which holds built-in functions and exceptions."

We won't go in to this in any more detail – just be careful when using the same name for different things in different functions, methods and classes.

The __str__ method

The `__str__` method can be used to provide information about an object in the form of a string. The most useful thing about it is the "shorthand" way of calling it in a `print` statement, it looks like just "printing out" the object (the variable that points to the object). Here's a very simple example. We haven't written our own `__str__` method so the default is used:

```
class Test:
    def __init__(self):
        self.x = 1

#main routine
if __name__ == "__main__":
    t = Test()
    print(t)                # shorthand way to call __str__ on the object
    print(t.__str__())      # the usual way to call a method. Equivalent to previous line.
    print(id(t))            # id prints the same address in decimal
```

Code Listing A.6

The first two lines of output are from the `__str__` method, the last line from the `id` method:

```
<__main__.Test object at 0x10273bd50>
<__main__.Test object at 0x10273bd50>
4336106832
```


As we can see the default `__str__` method returns a string describing the object's type (`Test` in the module `__main__` – see the Appendix) and its address. The address is in **hexadecimal** format (as indicated by 0x), where the hex number 10273bd50 is 4336106832 in decimal (as returned by the `id` function).

The default `__str__` method isn't very useful, so it's usual to write our own (**overriding** the **inherited** default). If we add the following method to the Test:

```
    def __str__(self):
        return "Hi it's me"
```

Then the output would be:

```
Hi it's me
Hi it's me
4336106832
```

Still not very useful? We usually write a `__str__` method (in Java the equivalent is the toString method) to return a useful description of the object. Consider the class `Cake` from Code Listing 1.2, and add the following method:

```
def __str__(self):
    return "{} cups flour \n{} eggs \n{} cups sugar \n{} tsp bSoda \n
     {} g butter \n{} layers".format(self._flour, self._eggs,
     self._sugar, self._bSoda, self._butter, self.num_layers)
```

If we make an instance of the class and (shorthand) call the `__str__` method on it:

```
c1 = Cake(2)
print(c1)
```

The output is a useful summary:

```
2 cups flour
1 eggs
1 cups sugar
1 tsp bSoda
200 g butter
1 layers
```

We have tended to write various "print", "show" or "display" methods for our classes, but any of the class definitions you have made as you worked through this chapter could have a `__str__` method included to do a similar task in a very convenient way.

Try this - A4

Write a `__str__` method that returns meaningful information about the `Person` class from Task 5 or Task 18. In the `main` routine create a `Person` object with appropriate information and display it in a `print` statement. Include the doctest:

```
    def __str__(self):
        """
        >>> p1 = Person('Percy', 1974, True)
        >>> print(p1)
        Name:Percy birth year:1974 isMale:True.
        """
```


Class variables

So far we have been using instance variables, which are able to hold different values in each instance of the class. In contrast, **class** variables are shared by all instances of the class, like a public resource that every instance can access and modify. A class variable is declared outside of any method, and by convention appears at the top of the class definition, directly after the class header. Figure 1.19 below shows 3 objects instantiated from the same class. They have individual instance variables, but all share the class variable.

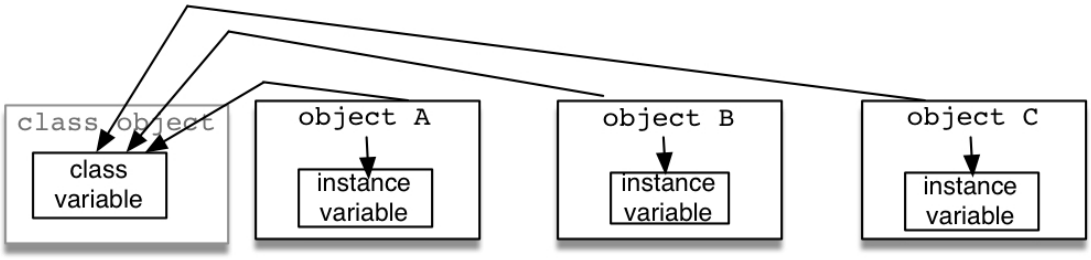


Figure A.0

The `Ant` class in code listing A.7 has an (x, y) position, and a shared `num_ants` class variable that exists once for all `Ant` objects. Each time the `__init__` method is called (each time an `Ant` instance is created) the class variable `num_ants` will increase by 1.

```
class Ant:

    #initialise the class variable giving the total number of ants
    num_ants = 0

    def __init__(self, x, y):
        """Initialise the x, y coordinates of this ant"""
        self.x = x
        self.y = y
        Ant.num_ants += 1     # update the class variable that is shared by all instances

#main routine
if __name__ == "__main__":
    a1 = Ant(0, 0)
    a2 = Ant(0, 1)
    print(Ant.num_ants)       # this will print 2
```

Code Listing A.7 The `Ant` class using class variables

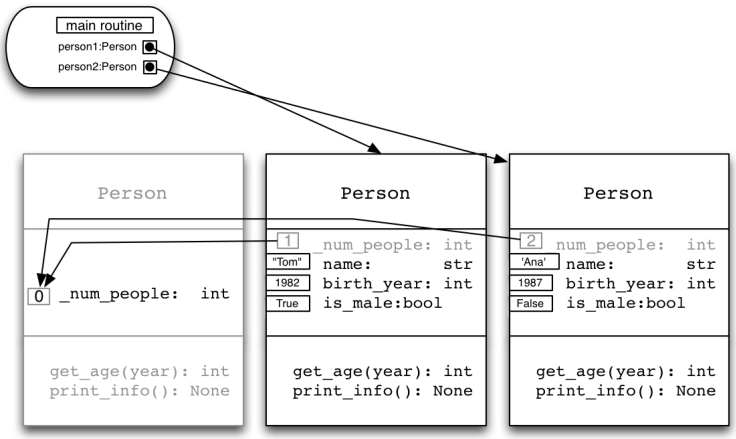
The use of class variables is not always necessary since the "public resource" may have a more logical or intuitive place in another class. We should question this code and decide whether or not it is better that this information belongs elsewhere in the program, perhaps in a `hive` object which keeps a list of individual `Ant` objects?

Try this - A5

In the `Person` class from Task 5, assume `Person` now needs to keep track of the number of `Person` objects that have been created using the class variable `_num_people` -- as in Figure A1.

Perhaps each person is part of a group of friends who are aware of how many people there are in their group. Note the grey object box on the left, which represents the class object. It is here that the class variable `_num_people` is set to 0.

Starting with your `Person` class, add the class variable `_num_people`. Every time a new `Person` object is created, increment `_num_people` and include this as information displayed by `print_info`.



.

Figure A1

Resources

From python.org

Language documentation (for version 3.3.0)
http://docs.python.org/release/3.3.0/

The tutorial
http://docs.python.org/release/3.3.0/tutorial/index.html

The language reference (technical material)
http://docs.python.org/release/3.3.0/reference/index.html

The Python standard library (technical material)
http://docs.python.org/release/3.3.0/library/index.html

Glossary of terms
http://docs.python.org/release/3.3.0/glossary.html

The style guide (useful for assessing the quality of Python code!)
http://www.python.org/dev/peps/pep-0008/

Other

How to Think Like a Computer Scientist – Python (superb tutorial with interactive examples!)
http://interactivepython.org/courselib/static/thinkcspy/index.html

Dive in to Python (popular text book available in various formats)
http://www.diveintopython.net/

The Beginners Guide to Python
http://wiki.python.org/moin/BeginnersGuide

Wikipedia (overview, history, heaps of references)
http://en.wikipedia.org/wiki/Python_(programming_language)

