Classifying a Parcel

CityParcels is a parcel delivery company which provides a variety of delivery options. The com- pany wishes to have a program that will correctly classify the parcel type and cost.

Parcel Classes

The parcel carrier provides the following parcel types:

Type	Max weight	Max length	Max width	Max height
Letter	0.1kg	240mm	165mm	5mm
Large letter	0.75kg	353mm	250mm	25mm
Small parcel	2kg	450mm	350mm	80mm
Small parcel	2kg	350mm	250mm	160mm
Medium parcel	20kg	610mm	460mm	460mm
Large parcel				

Notice there are a couple of oddities about this table. These are not mistakes. The information has been acurately copied from an an actual pricing table. The row for "Large parcel" is empty and there are two rows which say "Small parcel". What does this mean? There is a fairly clear interpretation but it does require consideration which goes a bit beyond what is in the raw data of the table.

Postage Pricing

The cost of sending a parcel is given by a basic cost for each parcel type, plus for every 0.1kg the parcel is over the basic weight an additional fee is charged. The details are given in the following table:

Type	Price	Basic Weight	Price per 0.1kg
Letter	£1.72	0.1kg	--
Large letter	£2.03	0.1kg	£0.10
Small parcel	£4.30	1kg	£0.40
Medium parcel	£6.75	10kg	£0.90
Large parcel	£15	10kg	£1.56

A suggested solution using a *declarative* representation of parcel types

Weakness of hard coding the table into complex conditional statements

Several years ago, the above problem was used as part of an introductory undergraduate Python course. Most students did write code that gave correct solutions. However, quite a few had code that was correct for some inputs and incorrect for others. When I looked at the submitted code, I saw that nearly all students had tackled the problem in the same way. What they had done is coded the table in terms of a long and complex sequence of `if`, `elif`, `else` statements, with all the numbers embedded directly in the code. Although in most cases the code did work, it was difficult to understand and, in in cases where there were errors, they were hard to find. Also, it was apparent that if the information needed to be updated, especially if new categories were added, the required code modification would be somewhat complex and error prone.

A *declarative* approach

Declarative programming means that, where possible, a programmer will specify code in terms of static information representing facts, rather than in terms of the dynamic flow of a program. The advantage is that declarative code is easier to understand, alter and modify.

Of course to get the functionality we want, we will usually still need to write some algorithmmic code involving conditionals and loops that will operate on our declaratively stated information. However, because the declarative information is kept separate from this, it often means that the algorithmic code also becomes simpler and easier to understand. And it nearly always means that the code is easier to modify or generalise.

In [7]:
```
## Store all parcel info in a dictionary
## This associates each parcel type name with the details for that type.
## These details are in turn stored as a dictionary.
## So we have a dictionary containing dictionaries.

PARCEL_TYPE_INFO = {
        "letter":
                  { "print_string" : "letter",
                    "max_weight"   : 0.1,
                    "max_length"   : 240,
                    "max_width"    : 165,
                    "max_height"   :   5,
                    "price"        : 1.72,
                    "base_weight"  : 0.1,
                    "price_per01"  : 0
                  },

        "large letter":
                  {"print_string" : "large letter",
                    "max_weight"   : 0.75,
                    "max_length"   : 353,
                    "max_width"    : 250,
                    "max_height"   :  25,
                    "price"        : 2.03,
                    "base_weight"  : 0.1,
                    "price_per01"  : 0.10
                  },

        "small parcel A":
                  { "print_string" : "small parcel",
                    "max_weight"   : 2,
                    "max_length"   : 450,
                    "max_width"    : 350,
                    "max_height"   :  80,
                    "price"        : 4.30,
                    "base_weight"  : 1,
                    "price_per01"  : 0.40
                  },

        "small parcel B":
                  { "print_string" : "small parcel",
                    "max_weight"   : 2,
                    "max_length"   : 350,
                    "max_width"    : 250,
                    "max_height"   : 160,
                    "price"        : 4.30,
                    "base_weight"  : 1,
                    "price_per01"  : 0.40
                  },

        "medium parcel":
                  { "print_string" : "medium parcel",
                    "max_weight"   : 20,
                    "max_length"   : 610,
                    "max_width"    : 460,
                    "max_height"   : 460,
                    "price"        : 6.75,
                    "base_weight"  : 10,
                    "price_per01"  : 0.90
                  },

        "large parcel":
                  { "print_string" : "large parcel",
                    "max_weight"   : float("inf"),
                    "max_length"   : float("inf"),
                    "max_width"    : float("inf"),
                    "max_height"   : float("inf"),
                    "price"        : 15,
                    "base_weight"  : 10,
                    "price_per01"  : 1.56
                  }
        }
```

In [17]:
```
## Create a small dictionary of the details of a parcel
## by taking input from a user
def get_parcel_details_input():
    parcel_details = {}
    for key in ["weight", "length", "width", "height"]:
        i = input( "Enter parcel " + key + ": " )
        measure = float(i)
        parcel_details[ key ] = measure
    return parcel_details

## Given the parcel details (in the form of a dictionary)
## We find the parcel type by looping through the types, until we find
## one that falls within the given limits.
## Here we are *assuming* that the types are arranged from low to high
## cost, so we pick the first that matches.
def find_parcel_type( parcel_details ):
    for parcel_type in ["letter", "large letter", "small parcel A",
                        "small parcel B", "medium parcel", "large parcel"]:
        type_details = PARCEL_TYPE_INFO[parcel_type]
        if ( parcel_details["weight"] <= type_details["max_weight"] and
             parcel_details["length"] <= type_details["max_length"] and
             parcel_details["width"]  <= type_details["max_width"] and
             parcel_details["height"] <= type_details["max_height"]
           ):
            return parcel_type
    ## Should never get here because large parcel has no limits
    print( "ERROR: parcel classification failure !!!")


def calculate_parcel_price( parcel_details ):
    parcel_type = find_parcel_type( parcel_details )
    base_price  = PARCEL_TYPE_INFO[parcel_type]["price"]
    base_weight = PARCEL_TYPE_INFO[parcel_type]["base_weight"]
    price_per01 = PARCEL_TYPE_INFO[parcel_type]["price_per01"]
    weight = parcel_details["weight"]
    if weight > base_weight:
        return base_price + ((weight - base_weight)/0.1) * price_per01
    else:
        return base_price

def pounds_str( val ):
    return "£{:.2f}".format(val)

def display_and_return_price_of_user_input_parcel():
    parcel_details = get_parcel_details_input()
    parcel_type = find_parcel_type( parcel_details )
    print( "This is a " + PARCEL_TYPE_INFO[parcel_type]["print_string"] +
"." )
    price = calculate_parcel_price( parcel_details )
    print( "The price to send this parcel is:", pounds_str(price) )
    return price


def multiple_parcel_classification_and_pricing():
    print( "** Parcel Classification and Pricing Program **" )
    total_price = 0
    while True:
        print() # add blank line before each parcel entry
        total_price += display_and_return_price_of_user_input_parcel()
        response = None
        while not response in ["y", "n"]:
            response = input( "\nWould you like to price up another one? (y/
n) " )
        if response == "n":
            break
    print( "\nThe total price for sending your parcels is:", pounds_str(tota
l_price) )
```

```
In [ ]: # Run the interactive parcel pricing function:
        multiple_parcel_classification_and_pricing()
```

Question

Suppose I want to send a heavy cube shaped object that measures `100*100*100mm` and weights `2kg` .

How should I package this object in order to minimise the postage I must pay?