University of HUDDERSFIELD

CFS2160: Software Design and Development

06: Reusing Code

Why reinvent the wheel?

Tony Jenkins
A.Jenkins@hud.ac.uk

Refactored


```
Number_of_Results = 5

results = []

for count in range (Number_of_Results):
    while 1:
        result = int (input ('Enter result #' + str (count + 1) + ': '))
        if result in range (0, 101):
            results.append (result)
            break
        else:
            print ('Invalid. Try again.')

print ('Average is:', sum (results) / len (results))
```

Reuse

This simple program contains a lot of code that could be used in other domains.

Put another way, it solves a number of common problems.

- It reads a number.
- It checks the number is within an allowed range.
- It asks the user to enter again if the number is out of range.
- It stores a collection of related numbers in a list.
- It calculates some useful (and common) statistics.

Reuse

This simple program contains a lot of code that could be used in other domains.

Put another way, it solves a number of common problems.

- It reads a number.
- It checks the number is within an allowed range.
- It asks the user to enter again if the number is out of range.
- It stores a collection of related numbers in a list.
- It calculates some useful (and common) statistics.

We would see identical code in any program that does these things, so it makes sense to *reuse* existing code.

Reuse

This simple program contains a lot of code that could be used in other domains.

Put another way, it solves a number of common problems.

- It reads a number.
- It checks the number is within an allowed range.
- It asks the user to enter again if the nu
- It stores a collection of related numbe
- It calculates some useful (and commo

We would see identical code in any program tha
to *reuse* existing code.

And we saw last week how this program would gradually become more and more cumbersome as we, say, added more validation.

Libraries

When we use a Python library, we are using code written by someone else that solves a common problem.

- `input` deals with reading from the keyboard.
- `print` renders these characters on the screen.
- `random.randint` provides random numbers in a certain range.
- `math.sqrt` finds a square root.

There is a great deal of pre-written Python code that we can use in any program.

Libraries

When we use a Python library, we are using code written by someone else that solves a common problem.

- `input` deals with reading from the keyboard.
- `print` renders these characters on the screen.
- `random.randint` provides random numbers in a certain range.
- `math.sqrt` finds a square root.

A key idea is that we know *what* this code does, and how to use it.

We do not know (or care) how it does what it does.

Libraries

When we use a Python library, we are using code written by someone else that solves a common problem.

- `input` deals with reading from the keyboard.
- `print` renders these characters on the screen.
- `random.randint` provides random numbers in a certain range.
- `math.sqrt` finds a square root.

A **key idea** is that we know *what* this code does, and how to use it.

We do not know (or care) how it does what it does.

Libraries

When we use a Python library, we are using code written by someone else that solves a common problem.

- `input` deals with reading from the keyboard.
- `print` renders these characters on the screen.
- `random.randint` provides random numbers in a certain range.
- `math.sqrt` finds a square root.

Related useful code is grouped together in *modules*.

So `random` is a *module* that contains useful code relating to "randomness". And `math` is a module that contains useful code for mathematical programs.

Functions

A *function* is a reusable chunk of code that can be called from anywhere in a program to do a certain task.

To use a function, we need to know:

1. What it does.
2. What it needs in order to do it.

We do *not* need to know how it does it.

Functions

To use a function, we need to know:

1. What it does.
2. What it needs in order to do it.

We do *not* need to know how it does it.

To use a function from the `math` module we:

- ➢ Study the docs.
- ➢ Import the function.
- ➢ Use the function.

Functions

To use a function, we need to know:

1. What it does.
2. What it needs in order to do it.

We do *not* need to know how it does it.

To use a function from the `math` module we:

- ➢ Study the docs.
- ➢ Import the function.
- ➢ Use the function.

```
>>> from math import sqrt
>>> print (sqrt (2))
1.4142135623730951

>>> from math import fabs
>>> print (fabs (-2.5))
2.5
```

Functions

To use a function, we need to know:

1. What it does.
2. What it needs in order to do it.

We do *not* need to know how it does it.

To use a function from the `math` module we:

- ➢ Study the docs.
- ➢ Import the function.
- ➢ Use the function.

```
>>> import math
>>> print (math.sqrt (2))
1.4142135623730951


>>> print (math.fabs (-2.5))
2.5
```

Alternatively, we can import the whole module, in which case the syntax is slightly different.

Split the Job

Programming is easy.

Programming is all about taking a big task, and breaking it down into smaller tasks.

Split the Job

Programming is easy.

Programming is all about taking a big task, and breaking it down into smaller tasks.

Then breaking down the smaller tasks.

Split the Job

Programming is easy.

Programming is all about taking a big task, and breaking it down into smaller tasks.

Then breaking down the smaller tasks.

And so on, until the very small tasks can be completed.

Split the Job

Programming is easy.

Programming is all about taking a big task, and breaking it down into smaller tasks.

Then breaking down the smaller tasks.

And so on, until the very small tasks can be completed.

As a rule of thumb, a "task" usually corresponds to 12 lines of code or fewer, and absolutely never to more than 24.

Why 24?

Using Functions

Sometimes a task is so common that there is a function in the *standard library*.

Some are so common that there is no need to import them explicitly.

We have already met many of these:

```
>>> 'Ni!'.lower ()
'ni!'
>>> len ('Ni!')
3
>>> 'N' in 'Ni!'
True
```

Using Functions

Sometimes a task is so common that there is a function in the *standard library.*

Some are so common that there is no need to import them explicitly.

We have already met many of these:

```
>>> 'Ni!'.lower ()
'ni!'
>>> len ('Ni!')
3
>>> 'N' in 'Ni!'
True
```

Pedantically, there are different things here.
`lower ()` is a *method*.
`len ()` is a *function*.
`in` is an *operator*.
Note the ways they are used.

Using Functions

Sometimes a task is so common that there is a function in the *standard library.*

Some are so common that there is no need to import them explicitly.

We have already met many of these:

```
>>> 'Ni!'.lower ()
'ni!'
>>> len ('Ni!')
3
>>> 'N' in 'Ni!'
True
```

But all are reusable code that we can use wherever we need them. We don't reinvent the wheel.

Using Functions

Sometimes a task is so common that there is a function in the *standard library*.

Some are so common that there is no need to import them explicitly.

We have already met many of these:

```
>>> 'Ni!'.lower ()
'ni!'
>>> len ('Ni!')
3
>>> 'N' in 'Ni!'
True
```

Nor do we know (or care) how they work.
But we have confidence that they do work.

Using Functions

Sometimes a task is so common that there is a function in the *standard library*.

Some are less common, and so need to be imported.

We have already met many of these. Here are two more:

```
>>> from math import ceil, floor
>>> ceil (2.5)
3
>>> floor (2.5)
2
```

Using Functions

Sometimes a task is so common that there is a function in the *standard library*.

Some are less common, and so need to be imported.

And here are two from a different module:

```
>>> from random import randint, choice
>>> randint (0, 10)
7
>>> choice (['Spam', 'Eggs', 'Beans'])
'Spam'
```

Using Functions

Sometimes a task is so common that there is a function in the *standard library.*

Some are less common, and so need to be imported.

And here are two from a different module:

```
>>> from random import randint
>>> randint (0, 10)
7
>>> choice (['Spam', 'Eggs', '
'Spam'
```

Important Note
We do not know (or care) *how* any of these work.
We just need to know what they do, and what they need in order to do it.

Using Functions

Sometimes a task is so common that there is a function in the *standard library.*

Some are less common, and so need to be imported.

And here are two from a different module:

```
>>> from random import randint
>>> randint (0, 10)
7
>>> choice (['Spam', 'Eggs', '
'Spam'
```

Remember that as well as containing useful, tried and tested, functions, modules can also contain relevant constant values.

Writing Functions

Sometimes there is no existing function to do what we need.

In this case we can write our own.

To do this we define:

- What the function does (usually just in a comment).
- What the function *returns*.
- What the function needs in order to do that.

And then we write code to achieve the desired result.

Writing Functions

Suppose we want to find out if a student has passed a test

And suppose that we need to do this several times in our program, and probably in many more programs.

Writing the code over and over is a **bad idea** for all sorts of reasons.

Writing Functions

Suppose we want to find out if a student has passed a test

And suppose that we need to do this several times in our program, and probably in many more programs.

The code itself is not difficult.

```
if student_mark > 40:
  print ('Cool Beans! They pass.')
else:
  print ('Bad News. They fail.')
```

Writing Functions

Suppose we want to find out if a student has passed a test

And suppose that we need to do this several times in our program, and probably in many more programs.

The code itself is not difficult.

Our first step to reusability is to replace the literal value 40 with a constant.

```
PASS_MARK = 40

if student_mark > PASS_MARK:
  print ('Cool Beans! They pass.')
else:
  print ('Bad News. They fail.')
```

Writing Functions

Suppose we want to find out if a student has passed a test

And suppose that we need to do this several times in our program, and probably in many more programs.

The code itself is not difficult.

Our first step to reusability is to replace the literal value 40 with a constant.

But we might still have to duplicate the code; if we wanted to use a different message, for example.

```
PASS_MARK = 40

if student_mark > PASS_MARK:
  print ('Cool Beans! They pass.')
else:
  print ('Bad News. They fail.')
```

Writing Functions

Suppose we want to find out if a student has passed a test

And suppose that we need to do this several times in our program, and probably in many more programs.

The code itself is not difficult.

Our first step to reusability is to replace the literal value 40 with a constant.

And the code has a bug, and we would really prefer to have to fix the bug in only one place.

```
PASS_MARK = 40

if student_mark >= PASS_MARK:
  print ('Cool Beans! They pass.')
else:
  print ('Bad News. They fail.')
```

Writing Functions

Suppose we want to find out if a student has passed a test

And suppose that we need to do this several times in our program, and probably in many more programs.

The code itself is not difficult.

Our first step to reusability is to replace the literal value 40 with a constant.

And the code has a bug, and we would really prefer to have to fix the bug in only one place.

```
PASS_MARK = 40

if student_mark >= PASS_MARK:
  print ('Cool Beans! They pass.')
else:
  print ('Bad News. They fail.')
```

The solution is to *encapsulate* this little bit of *business logic* into a function.

Writing Functions

Suppose we want to find out if a student has passed a test

And suppose that we need to do this several times in our program, and probably in many more programs.

The code itself is not difficult.

Our first step to reusability is to replace the literal value 40 with a constant.

And the code has a bug, and we would really prefer to have to fix the bug in only one place.

```
PASS_MARK = 40

if student_mark >= PASS_MARK:
  print ('Cool Beans! They pass.')
else:
  print ('Bad News. They fail.')
```

The function gets used wherever the logic is needed.
And if the rules change, only one small bit of code needs to be changed.

Writing Functions

Suppose we want to find out if a student has passed a test

And suppose that we need to do this several times in our program, and probably in many more programs.

The code itself is not difficult.

Our first step to reusability is to replace the literal value 40 with a constant.

And the code has a bug, and we would really prefer to have to fix the bug in only one place.

```
PASS_MARK = 40

if student_mark >= PASS_MARK:
  print ('Cool Beans! They pass.')
else:
  print ('Bad News. They fail.')
```

The code needed is self-contained.
So this idea could also be used to split a large program between developers.

Pass or Fail?

What the function does.

What the function needs.

What the function returns.

Pass or Fail?

What the function does.

Given a mark and a pass mark, the function determines whether the student has passed or failed.

What the function needs.

A pass mark (integer) and a mark (integer).

What the function returns.

`True` if the student passes, `False` otherwise.

Pass or Fail?

What the function does.

Given a mark and a pass mark, the function determines whether the student has passed or failed.

What the function needs.

A pass mark (integer) and a mark (integer, 0 to 100).

What the function returns.

`True` if the student passes, `False` otherwise.

Pass or Fail?

What the function does.

Given a mark and a pass mark, the function determines whether the student has passed or failed.

What the function needs.

A pass mark (integer) and a mark (integer, 0 to 100).

What the function returns.

`True` if the student passes, `False` otherwise.

```
def pass_or_fail (pass_mark, grade):
```

```
  if grade >= pass_mark:
    return True
  else:
    return False
```

Pass or Fail?

What the function does.

Given a mark and a pass mark, the function determines whether the student has passed or failed.

What the function needs.

A pass mark (integer) and a mark (integer, 0 to 100).

What the function returns.

`True` if the student passes, `False` otherwise.

```
def pass_or_fail (pass_mark, grade):

  """
     Determine pass or fail on a test.
     Parameters: pass mark and grade.
     Returns: True if passed, False
              Otherwise.
  """

  if grade >= pass_mark:
    return True
  else:
    return False
```

Pass or Fail?

What the function does.

Given a mark and a pass mark, the function determines whether the student has passed or failed.

What the function needs.

A pass mark (integer) and a mark (integer, 0 to 100).

What the function returns.

`True` if the student passes, `False` otherwise.

```
def pass_or_fail (pass_mark, grade):

  """
     Determine pass or fail on a test.
     Parameters: pass mark and grade.
     Returns: True if passed, False
              Otherwise.
  """

  return grade >= pass_mark
```

What Grade?

What the function does.

Given a mark, the function determines what grade the student has on the test.

What the function needs.

A mark (on a scale of 0 to 100).

What the function returns.

A grade (a String).

```
def grade (mark):

  """
     Determine grade on a test.
     Parameters: numeric mark.
     Returns: Corresponding grade.
  """

  if mark >= 70:
    return 'D'
  elif mark >= 60:
    return 'M'
  elif mark >= 50:
    return 'P'
  else:
    return 'F'
```

What Grade?

What the function does.

Given a mark, the function determines what grade the student has on the test.

What the function needs.

A mark (on a scale of 0 to 100).

What

A gra

Code Smell!

What does this function return if the mark is 150?

```
def grade (mark):

  """
     Determine grade on a test.
     Parameters: numeric mark.
     Returns: Corresponding grade.
  """

  if mark >= 70:
    return 'D'
  elif mark >= 60:
    return 'M'
  elif mark >= 50:
    return 'P'
  else:
    return 'F'
```

What Grade?

What the function does.

Given a mark, the function determines what grade the student has on the test.

What the function needs.

A mark (on a scale of 0 to 100).

What

A gra

Code Smell!

What does this function return if the mark is 23.45?

```
def grade (mark):

  """
     Determine grade on a test.
     Parameters: numeric mark.
     Returns: Corresponding grade.
  """

  if mark >= 70:
    return 'D'
  elif mark >= 60:
    return 'M'
  elif mark >= 50:
    return 'P'
  else:
    return 'F'
```

What Grade?

What the function does.

Given a mark, the function determines what grade the student has on the test.

What the function needs.

A mark (on a scale of 0 to 100).

Wha

A gra

Code Smell!

What does this function return if the mark is "Pass"?

```
def grade (mark):

  """
     Determine grade on a test.
     Parameters: numeric mark.
     Returns: Corresponding grade.
  """

  if mark >= 70:
    return 'D'
  elif mark >= 60:
    return 'M'
  elif mark >= 50:
    return 'P'
  else:
    return 'F'
```

What Grade?

The *parameter* passed to the function can be any type - this not checked.

So errors can occur if the value passed is not the expected type.

In this code there are two possible errors:

- `mark` is not an integer.
- `mark` is an integer, but is out of range.

```
def grade (mark):

  """
     Determine grade on a test.
     Parameters: numeric mark.
     Returns: Corresponding grade.
  """

  if mark >= 70:
    return 'D'
  elif mark >= 60:
    return 'M'
  elif mark >= 50:
    return 'P'
  else:
    return 'F'
```

What Grade?

The *parameter* passed to the function can be any type - this not checked.

So errors can occur if the value passed is not the expected type.

In this code there are two possible errors:

- `mark` is not an integer.
- `mark` is an integer, but is out of range.

We need to check for both of these, and if one is detected, we need to *raise* the problem.

```
def grade (mark):

  """
     Determine grade on a test.
     Parameters: numeric mark.
     Returns: Corresponding grade.
  """

  if mark >= 70:
    return 'D'
  elif mark >= 60:
    return 'M'
  elif mark >= 50:
    return 'P'
  else:
    return 'F'
```

What Grade?

The *parameter* passed to the function can be any type - this not checked.

So errors can occur if the value passed is not the expected type.

In this code there are two possible errors:

- `mark` is not an integer.
- `mark` is an integer, but is out of range.

We need to check for both of these, and if one is detected, we need to *raise* the problem.

```
def grade (mark):

  """
     Determine grade on a test.
     Parameters: numeric mark.
     Returns: Corresponding grade.
  """
```

These are two different error types.

The first is a "Type Error".
The second is a "Value Error".

What Grade?

We can easily detect if the mark is out of range with a simple conditional statement.

```
def grade (mark):

  if mark not in range (0, 101):
    # Reject It.
    pass

  if mark >= 70:
    return 'D'
  elif mark >= 60:
    return 'M'
  elif mark >= 50:
    return 'P'
  else:
    return 'F'
```

What Grade?

We can easily detect if the mark is out of range with a simple conditional statement.

But, what to do?

- We could print an error message, but there might be no-one to read it.
- We could return some other letter, but that could be confusing.
- We could cause the program to crash, but that would be extreme.

```
def grade (mark):

  if mark not in range (0, 101):
    # Reject It.
    pass

  if mark >= 70:
    return 'D'
  elif mark >= 60:
    return 'M'
  elif mark >= 50:
    return 'P'
  else:
    return 'F'
```

What Grade?

We can easily detect if the mark is out of range with a simple conditional statement.

But, what to do?

We *throw an exception*, and trust whatever called the function to handle this and pick up the pieces.

```
def grade (mark):

  if mark not in range (0, 101):
    raise ValueError ('Invalid Mark')


  if mark >= 70:
    return 'D'
  elif mark >= 60:
    return 'M'
  elif mark >= 50:
    return 'P'
  else:
    return 'F'
```

What Grade?

We can easily detect if the mark is out of range with a simple conditional statement.

But, what to do?

We throw an exception, and trust whatever called the function to handle this and pick up the pieces.

We do this because it is the most general response. We are not relying on there being someone reading messages.

(In real life, the exception could be processed to print a message, flash a light, send a text …)

```
def grade (mark):

  if mark not in range (0, 101):
    raise ValueError ('Invalid Mark')


  if mark >= 70:
    return 'D'
  elif mark >= 60:
    return 'M'
  elif mark >= 50:
    return 'P'
  else:
    return 'F'
```

What Grade?

To deal with the case that `mark` is not an integer, we need to be able to check what type it is.

```
def grade (mark):
  if mark >= 70:
    return 'D'
  elif mark >= 60:
    return 'M'
  elif mark >= 50:
    return 'P'
  else:
    return 'F'
```

What Grade?

To deal with the case that `mark` is not an integer, we need to be able to check what type it is.

In some cases, the code will fail anyway if the type is not what we expect.

- In this code a string would cause a `TypeError`.
- So would a list, and a tuple.
- But, a `float` (and, oddly, a `bool`) would be OK.

```
def grade (mark):
  if mark >= 70:
    return 'D'
  elif mark >= 60:
    return 'M'
  elif mark >= 50:
    return 'P'
  else:
    return 'F'
```

What Grade?

To deal with the case that `mark` is not an integer, we need to be able to check what type it is.

In some cases, the code will fail anyway if the type is not what we expect.

- In this code a string would cause a `TypeError`.
- So would a list, and a tuple.
- But, a `float` (and, oddly, a `bool`) would be OK.

So here we do need to find out what type `mark` is.

```
def grade (mark):
    if mark >= 70:
      return 'D'
    elif mark >= 60:
      return 'M'
    elif mark >= 50:
      return 'P'
    else:
      return 'F'
```

Checking Type

This is not really very Pythonic, but needs must sometimes.

`isinstance` is a function that will tell us whether a variable is currently of a particular type.

```
>>> x = 1
>>> isinstance (x, bool)
False
>>> isinstance (x, int)
True

>>> y = 1.0
>>> isinstance (y, int)
False
>>> isinstance (y, float)
True
```

What Grade?

To deal with the case that `mark` is not an integer, we need to be able to check what type it is.

```
def grade (mark):

  if not isinstance (x, int):
    # Reject It.
    pass

  if mark >= 70:
    return 'D'
  elif mark >= 60:
    return 'M'
  elif mark >= 50:
    return 'P'
  else:
    return 'F'
```

What Grade?

To deal with the case that `mark` is not an integer, we need to be able to check what type it is.

It makes sense to use the same approach as if a string had been entered, and to use a `TypeError` to report (*raise*) the problem.

```
def grade (mark):

  if not isinstance (x, int):
    raise TypeError ('Invalid Type')


  if mark >= 70:
    return 'D'
  elif mark >= 60:
    return 'M'
  elif mark >= 50:
    return 'P'
  else:
    return 'F'
```

Functions in Programs

Functions are usually defined in a program at the top.

They can then be used in the code below.

Functions can, of course, use other functions …

Functions in Programs

Functions are usually defined in a program at the top.

The code in these functions is *not executed*. This is just defining what the function is.

It is processed, and syntax errors will cause problems, but it is not executed.

```
def some_function ():
  pass

def some_other_function ():
  pass

def yet_another_function ():
  pass
```

Functions in Programs

Functions are usually defined in a program at the top.

The code in these functions is *not executed*. This is just defining what the function is.

It is processed, and syntax errors will cause problems, but it is not executed.

The main program comes underneath, and is often marked out like this:

```
def some_function ():
  pass

def some_other_function ():
  pass

def yet_another_function ():
  pass

if __name__ == '__main__':
  pass
```

PyCharm Demo and Question Time

Jobs

By next week, you should:

- Have read up to the end of Unit 5 in the book.
  - Worked through the examples.
- Be all up to date with practicals.

Next week we move on to Java.

Overleaf is "Hello World", in Java.

The Java IDE is IntelliJ. It is on the Lab PCs. Use it to run this program.

Java "Hello World"


```
public class HelloWorld {

    public HelloWorld () {
        System.out.println ("Hello World");
    }

    public static void main (String[] args) {
        new HelloWorld ();
    }
}
```


python
powered