University of
HUDDERSFIELD

CFS2160: Software Design and Development

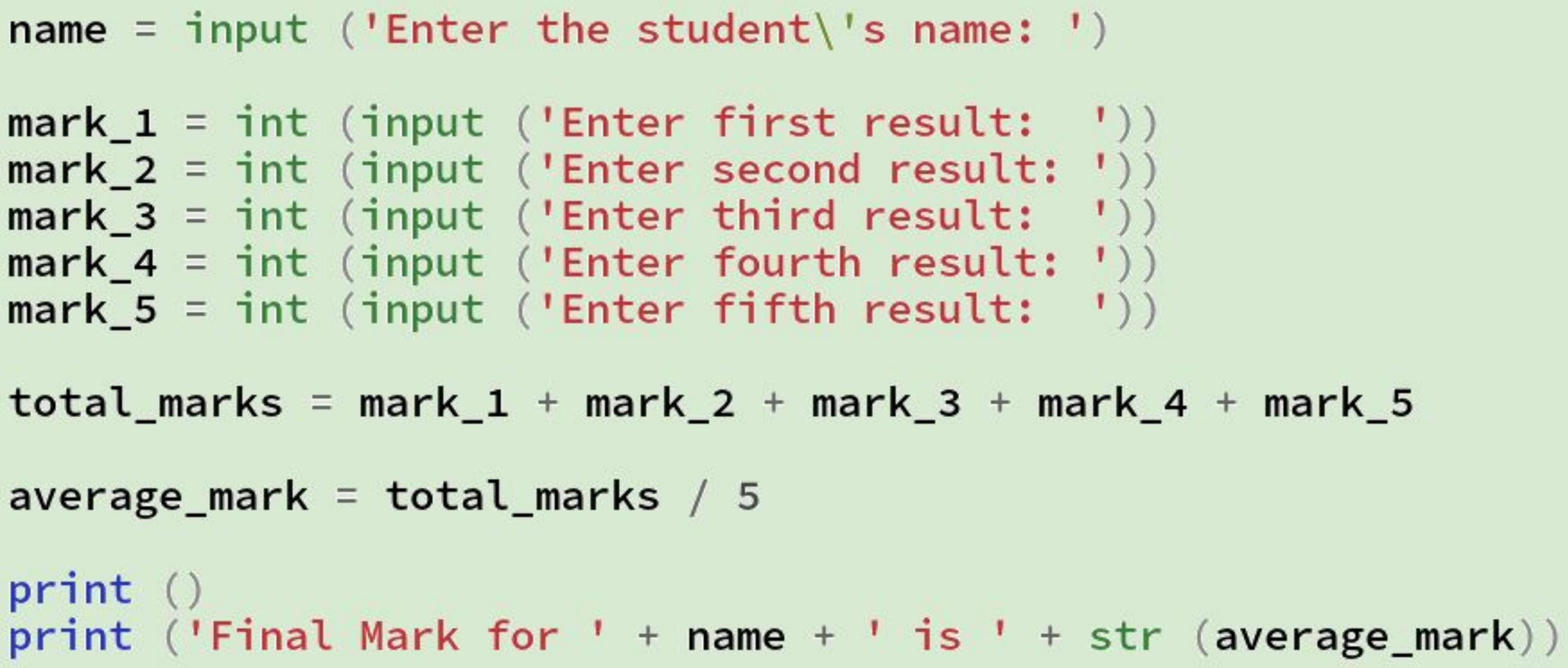
python
powered

05: Repeating Yourself

Do I have to keep telling you?

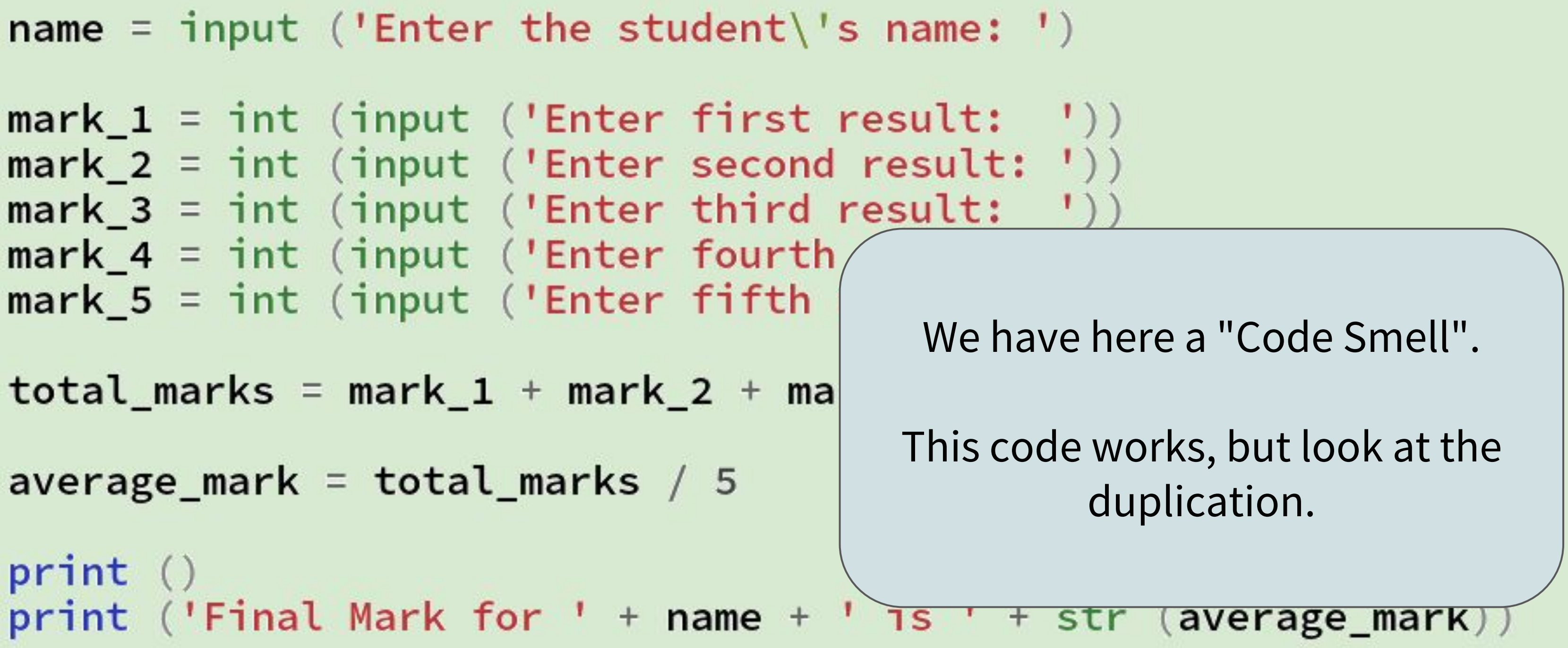
Tony Jenkins
A.Jenkins@hud.ac.uk

Refactoring



```
name = input ('Enter the student\'s name: ')

mark_1 = int (input ('Enter first result:  '))
mark_2 = int (input ('Enter second result: '))
mark_3 = int (input ('Enter third result:  '))
mark_4 = int (input ('Enter fourth result: '))
mark_5 = int (input ('Enter fifth result:  '))

total_marks = mark_1 + mark_2 + mark_3 + mark_4 + mark_5

average_mark = total_marks / 5

print ()
print ('Final Mark for ' + name + ' is ' + str (average_mark))
```

Refactoring

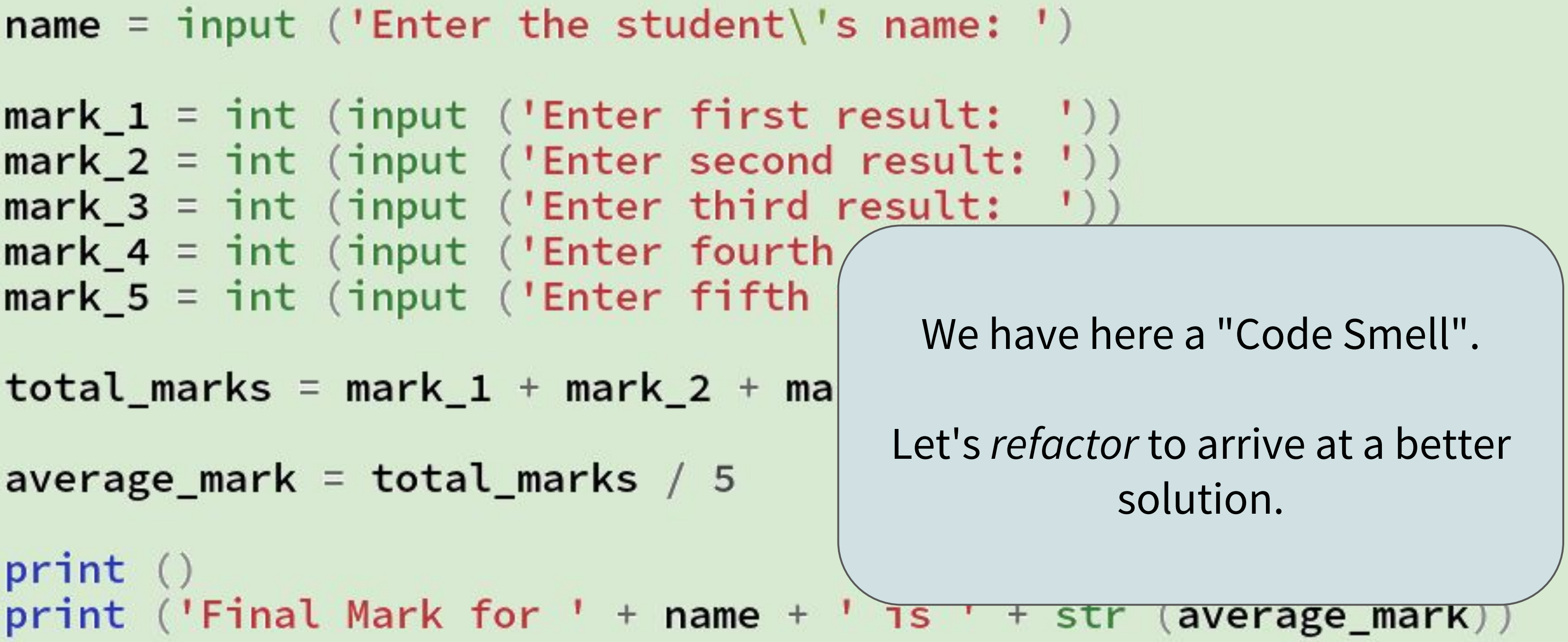


```
name = input ('Enter the student\'s name: ')

mark_1 = int (input ('Enter first result:  '))
mark_2 = int (input ('Enter second result: '))
mark_3 = int (input ('Enter third result:  '))
mark_4 = int (input ('Enter fourth
mark_5 = int (input ('Enter fifth

total_marks = mark_1 + mark_2 + ma

average_mark = total_marks / 5

print ()
print ('Final Mark for ' + name + ' is ' + str (average_mark))
```

We have here a "Code Smell".

This code works, but look at the duplication.

Refactoring

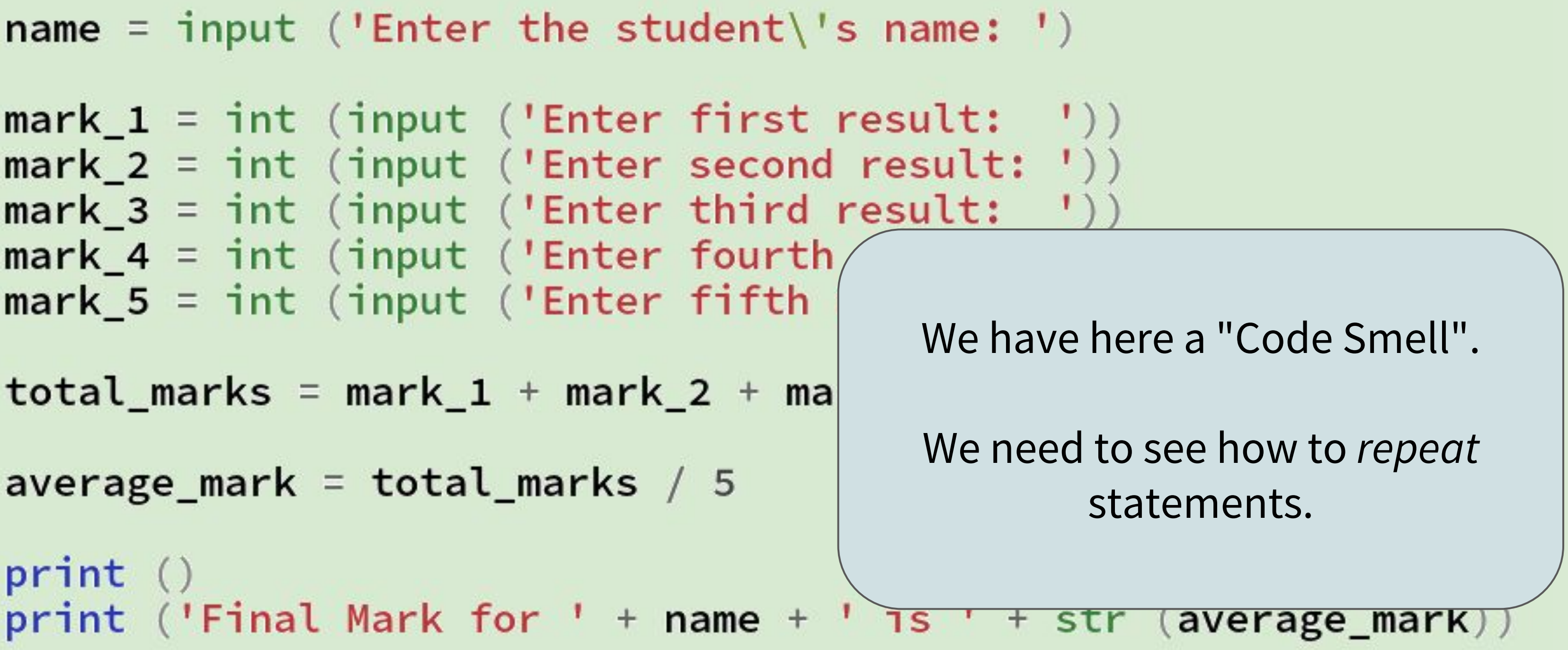


```
name = input ('Enter the student\'s name: ')

mark_1 = int (input ('Enter first result:  '))
mark_2 = int (input ('Enter second result: '))
mark_3 = int (input ('Enter third result:  '))
mark_4 = int (input ('Enter fourth
mark_5 = int (input ('Enter fifth

total_marks = mark_1 + mark_2 + ma

average_mark = total_marks / 5

print ()
print ('Final Mark for ' + name + ' is ' + str (average_mark))
```

We have here a "Code Smell".

Let's *refactor* to arrive at a better solution.

Refactoring

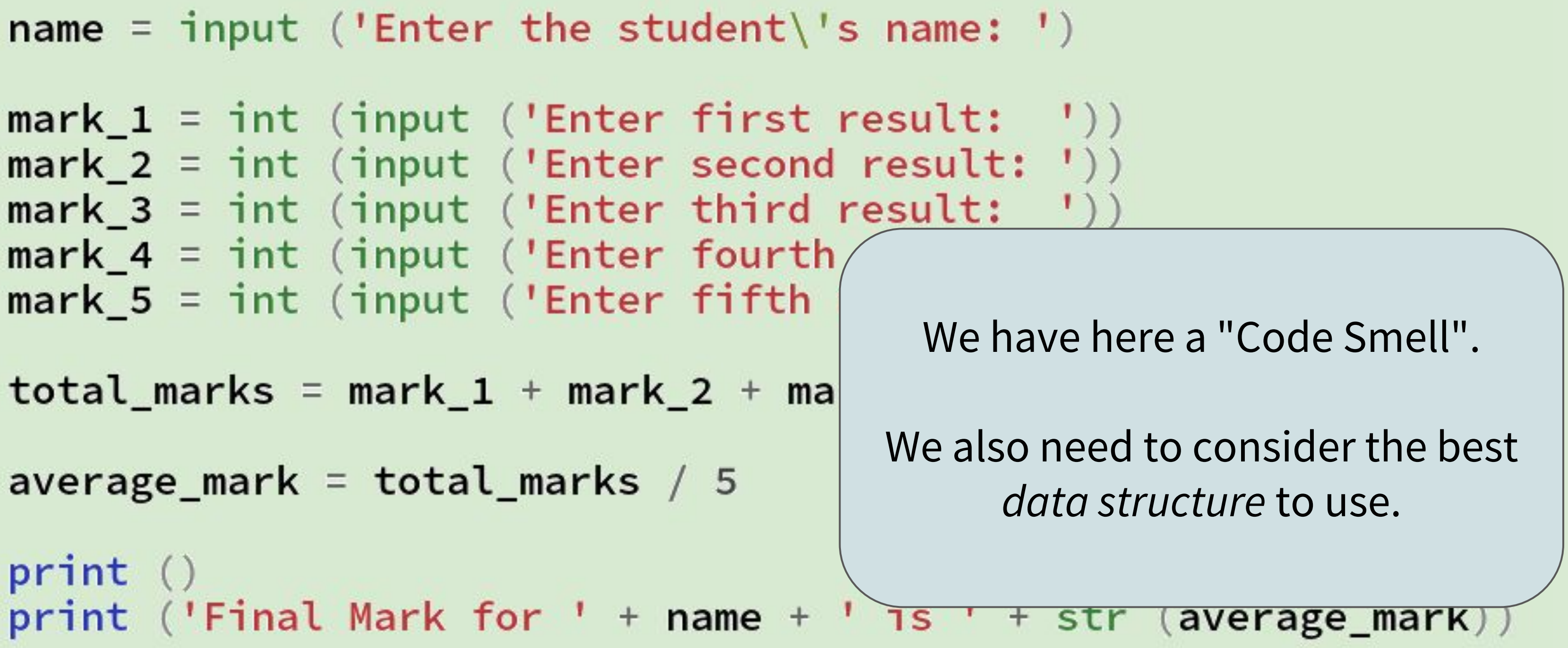


```
name = input ('Enter the student\'s name: ')

mark_1 = int (input ('Enter first result:  '))
mark_2 = int (input ('Enter second result: '))
mark_3 = int (input ('Enter third result:  '))
mark_4 = int (input ('Enter fourth
mark_5 = int (input ('Enter fifth

total_marks = mark_1 + mark_2 + ma

average_mark = total_marks / 5

print ()
print ('Final Mark for ' + name + ' is ' + str (average_mark))
```

We have here a "Code Smell".

We need to see how to *repeat* statements.

Refactoring



```
name = input ('Enter the student\'s name: ')

mark_1 = int (input ('Enter first result:  '))
mark_2 = int (input ('Enter second result: '))
mark_3 = int (input ('Enter third result:  '))
mark_4 = int (input ('Enter fourth
mark_5 = int (input ('Enter fifth

total_marks = mark_1 + mark_2 + ma

average_mark = total_marks / 5

print ()
print ('Final Mark for ' + name + ' is ' + str (average_mark))
```

We have here a "Code Smell".

We also need to consider the best *data structure* to use.

Refactoring

Issues with this program include (but are not limited to):

- We have five almost identical prompts.
  - It would be good to replace them with one prompt, and have the code *repeat.*
- We have five integer variables, used for almost the same thing.
  - We could replace them with a single data structure, like a Tuple.
- The results entered could be out of range.
  - We can detect this, but what do we do about it?

So now we go on to see how to fix these things.

Refactoring

Issues with this program include (but are not limited to):

- We have five almost identical prompts.
  - It would be good to replace them with one prompt, and have the code *repeat.*
- We have five integer variables, used for almost the same thing
  - We could replace them with a single data structur
- The results entered could be out of range.
  - We can detect this, but what do we do about it?

So now we go on to see how to fix these things.

The message here is that it's not enough for a program to work, it must work efficiently.

Aside: Technical Debt

There is a concept in Software Development called "Technical Debt".

It refers to the cost of using a "quick and messy" solution to a coding problem as opposed to using a "better" solution.

The quick solution gets your code working, but you need to make repayments in terms of refactoring effort later on.

And the pressure to deliver often drives a developer down the "quick and messy" route.

Aside: Technical Debt

There is a concept in Software Development called "Technical Debt".

It refers to the cost of using a "quick and messy" solution to a coding problem as opposed to using a "better" solution.

The quick solution gets your code working, but y
of refactoring effort later on.

And the pressure to deliver often drives a develo
e.

This is a massive issue for people working in real software.

The current "Technical Debt" is in the £Bns.

Aside: Technical Debt

There is a concept in Software Development called "Technical Debt".

It refers to the cost of using a "quick and messy" solution to a coding problem as opposed to using a "better" solution.

The quick solution gets your code working, but y s
of refactoring effort later on.

And the pressure to deliver often drives a develo e.

Another "Technical Debt" issue here is how we would modify the program to handle 10 marks, or an arbitrary number.

Refactoring

We'll solve the problems like this:

- The repeated prompt will be one `input` statement, but we'll use it five times.
- The five integers will be held together in one place.
- The results entered will be tested, and the `input` statement will be repeated if the value is out of range.

Refactoring

We'll solve the problems like this:

- The repeated prompt will be one `input` statement, but we'll use it five times.
  - That is, we'll *repeat* it.
- The five integers will be held together in one place.
  - You might think Tuple (which would be a good call), a List would also work.
- The results entered will be tested, and the `input` statement will be repeated if the value is out of range.
  - That word "repeat" again.

So we need to learn about two new things: repetition, and repetition.

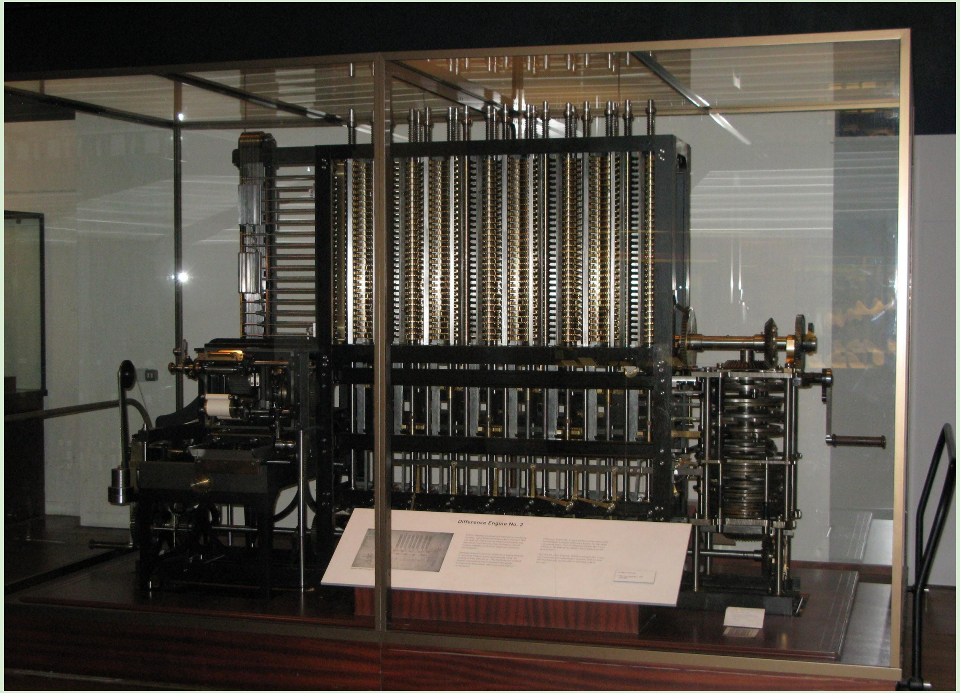
Repetition

One of the original reasons for developing "computers" was to carry out repetitive tasks.

Humans are bad at repetition.

Computers are seriously good at it.

This is the "Difference Engine", used for calculating mathematical tables.



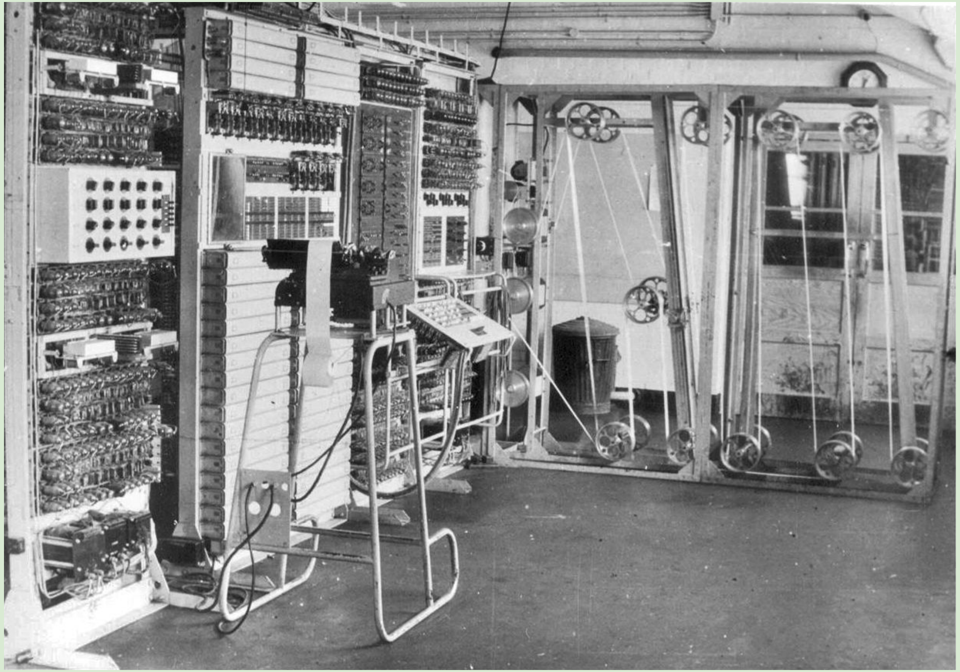
Repetition

One of the original reasons for developing "computers" was to carry out repetitive tasks.

Humans are bad at repetition.

Computers are seriously good at it.

This is Colossus, used to crack codes in the Second World War.



Types of Repetition

We can identify different types of repetition:

- *Infinite* - something is done for ever, and ever, and ever.
- *Determinate* - where it is known in advance how many repetitions are needed.
- *Indeterminate* - where is is not known in advance how many repetitions are needed.

Types of Repetition

We can identify different types of repetition:

- *Infinite* - something is done for ever, and ever, and ever.
- *Determinate* - where it is known in advance how many repetitions are needed.
- *Indeterminate* - where is is not known in advance how many repetitions are needed.

Indeterminate breaks down further:

1. The number of repetitions is unknown, but is at least one.
2. The number of repetitions is unknown, and could be none at all.

Types of Repetition

To take some analogies:

- Do 10 press-ups.
- Run the bath until it is full.
- Open packs of stickers until you get the one with Homer Simpson.
- Take pens from your bag until you find the green one.
- Look for a free PC in Canalside West.
- Bob tells you he's in Canalside West, so go and find him.

Determinate Repetition

Determinate ~~Repetition~~ Loops

These are often called "for" loops, because the keyword `for` is usually used to introduce the statement.

Determinate ~~Repetition~~ Loops

These are often called "for" loops, because the keyword `for` is usually used to introduce the statement.

In Python, `for` takes anything *iterable* and repeats statements for each item in the iterable structure.

Determinate ~~Repetition~~ Loops

These are often called "for" loops, because the keyword `for` is usually used to introduce the statement.

In Python, `for` takes anything *iterable* and repeats statements for each item in the iterable structure.

In this case the *iterable* is a Tuple.

```
>>> for i in ('Eggs', 'Spam', 'Beans'):
...   print (i)
...
Eggs
Spam
Beans
```

Determinate ~~Repetition~~ Loops

These are often called "for" loops, because the keyword `for` is usually used to introduce the statement.

In Python, `for` takes anything *iterable* and repeats statements for each item in the iterable structure.

In this case the *iterable* is a Tuple.

It could equally be a List.

```
>>> for i in ('Eggs', 'Spam', 'Beans'):
...   print (i)
...
Eggs
Spam
Beans

>>> for i in ['Eggs', 'Spam', 'Beans']:
...   print (i)
...
Eggs
Spam
Beans
```

Determinate ~~Repetition~~ Loops

These are often called "for" loops, because the keyword `for` is usually used to introduce the statement.

In Python, `for` takes anything *iterable* and repeats statements for each item in the iterable structure.

In this case the *iterable* is a Tuple.

It could equally be a List.

This repetition is *determinate* because the number of repetitions will be the number of elements in the List (or Tuple).

```
>>> for i in ('Eggs', 'Spam', 'Beans'):
...   print (i)
...
Eggs
Spam
Beans
```

```
>>> for i in ['Eggs', 'Spam', 'Beans']:
...   print (i)
...
Eggs
Spam
Beans
```

Determinate ~~Repetition~~ Loops

These are often called "for" loops, because the keyword `for` is usually used to introduce the statement.

In Python, `for` takes anything *iterable* and repeats state

In th

It co

This

of re

List (or Tuple).

Notice again how the *indentation* is very important in these examples.

```
>>> for i in ('Eggs', 'Spam', 'Beans'):
...   print (i)
...
Eggs
Spam
Beans

>>> for i in ['Eggs', 'Spam', 'Beans']:
...   print (i)
...
Eggs
Spam
Beans
```

Ranges

The built-in function `range` generates a sequence of integers.

This sequence can be used for iteration.

Ranges

The built-in function `range` generates a sequence of integers.

This sequence can be used for iteration.

```
>>> for i in range (1, 3):
...   print (i)
...
1
2

>>> for i in range (4):
...   print (i)
...
0
1
2
3
```

Ranges

The built-in function range generates a sequence of integers.

This sequence can be used for iteration.

```
>>> for i in range (1, 3):
...   print (i)
...
1
2

>>> for i in range (4):
...   print ('Ni!')
...
Ni!
Ni!
Ni!
Ni!
```

Ranges

The built-in function `range` generates a sequence of integers.

This sequence can be used for iteration.

It works like this:

```
range ([start,] stop [, step])
```

https://docs.python.org/3.6/library/functions.html#func-range

```
>>> for i in range (1, 3):
...   print (i)
...
1
2

>>> for i in range (4):
...   print ('Ni!')
...
Ni!
Ni!
Ni!
Ni!
```

Ranges

The built-in function `range` generates a sequence of integers.

This sequence can be used for iteration.

It works like this:

```
   range ([start,] stop [, step])
```

https://docs.python.org/3.6/library/functions.html#func-range

```
>>> for i in range (1, 3):
...   print (i)
...
1
2

>
```

Note
Precisely what `range` does *behind the scenes* changed between Python 2 and Python 3. Be sure to read the right docs!

Iterables

Many data types apart from lists in Python are *iterable*.

Iterables

Many data types apart from lists in Python are *iterable*.

Strings:

```
>>> for c in 'Ni!':
...   print (c)
...
N
i
!
```

Iterables

Many data types apart from lists in Python are *iterable*.

Dictionaries:

```
>>> k = {'Robin':'Yes', 'Galahad':'No'}

>>> for c in k:
...   print (c)
...
Robin
Galahad

>>> for c in k:
...   print (k [c])
...
Yes
No
```

Iterables

Many data types apart from lists in Python are *iterable*.

Dictionaries:

Dictionaries allow us to manage "Key-Value" pairs.

Full details (along with handy examples) are in the docs:

https://docs.python.org/3.6/tutorial/datastructures.html#dictionaries

```
>>> k = {'Robin':'Yes', 'Galahad':'No'}

>>> for c in k:
...   print (c)
...
Robin
Galahad

>>> for c in k:
...   print (k [c])
...
Yes
No
```

Indeterminate Loops

In our results program we want to iterate *while* some condition is `True`.

Hence these are often referred to as "While Loops".

Python makes no distinction between the two subtly different types. Some languages (C, C++, Java, Pascal) do.

The Python rationale is that you don't really need two kinds: in Python there should be one, and ideally only one, way to do it.

Indeterminate Loops

In our results program we want to iterate while some condition is `True`.

Hence these are often referred to as "While Loops".

Python makes no distinction between the two subtly different types. Some languages (C, C++, Java, Pascal) do.

The Python rationale is that you don't really need two kinds: in Python there should be one, and ideally only one, way to do it.

```
colour = 'green'
choice = ''

while choice != colour:
  choice = input ('Favourite Colour: ')

print ('You may pass!')
```

Indeterminate Loops

In our results program we want to iterate while some condition is True.

Hence these are often referred to as "While Loops".

Python makes no distinction between the two

```
colour = 'green'
choice = ''

while choice != colour:
  choice = input ('Favourite Colour: ')

print ('You may pass!')
```

Indentation is again important.

The input is inside the loop, the print is outside.

Indeterminate Loops

In our results program we want to iterate while some condition is `True`.

Hence these are often referred to as "While Loops".

Python makes no distinction between the two subtly different types. Some languages (C, C++, Java, Pascal) do.

The Python rationale is that you don't really need two kinds: in Python there should be one, and ideally only one, way to do it.

```
colour = 'green'
choice = ''

while choice != colour:
  choice = input ('Favourite Colour: ')

  if choice != colour:
    print (Ni!')

print ('You may pass!')
```

Indeterminate Loops

This code works, and would be a common "recipe" in some languages.

More Pythonic is to make use of an infinite loop.

This will mean that the condition is only tested the once.

```
colour = 'green'
choice = ''

while choice != colour:
  choice = input ('Favourite Colour: ')

  if choice != colour:
    print (Ni!')

print ('You may pass!')
```

Indeterminate Loops

This code works, and would be a common "recipe" in some languages.

More ***Pythonic*** is to make use of an infinite loop.

This will mean that the condition is only tested the once.

```
colour = 'green'
choice = ''

while choice != colour:
  choice = input ('Favourite Colour: ')

  if choice != colour:
    print (Ni!')

print ('You may pass!')
```

Indeterminate Loops

This code works, and would be a common "recipe" in some languages.

More Pythonic is to make use of an infinite loop.

This will mean that the condition is only tested the once.

```
colour = 'green'


while 1:
  choice = input ('Favourite Colour: ')

  if choice == colour:
    break
  else:
    print (Ni!')

print ('You may pass!')
```

Infinite Loops

An infinite loop will never end.

Which implies that there is nothing inside the loop that will alter the value of the condition.

Infinite Loops

An infinite loop will never end.

Which implies that there is nothing inside the loop that will alter the value of the condition.

```
# Print 'Ni!' 10 times.

count = 1

while count < 10:
  print ('Ni!')
```

Infinite Loops

An infinite loop will never end.

Which implies that there is nothing inside the loop that will alter the value of the condition.

This is often a *bug*, when the programmer has forgotten to include code that might alter the condition.

```
# Print 'Ni!' 10 times.

count = 1

while count < 10:
  print ('Ni!')
  count += 1
```

Loops

An infinite loop will never end.

Which implies that there is nothing inside the loop that will alter the value of the condition.

This is often a *bug*, when the programmer has forgotten to include code that might alter the condition.

There's actually still a bug in that loop. Can you see it?

```
# Print 'Ni!' 10 times.

count = 1

while count < 10:
  print ('Ni!')
  count += 1
```

Loops

An infinite loop will never end.

Which implies that there is nothing inside the loop that will alter the value of the condition.

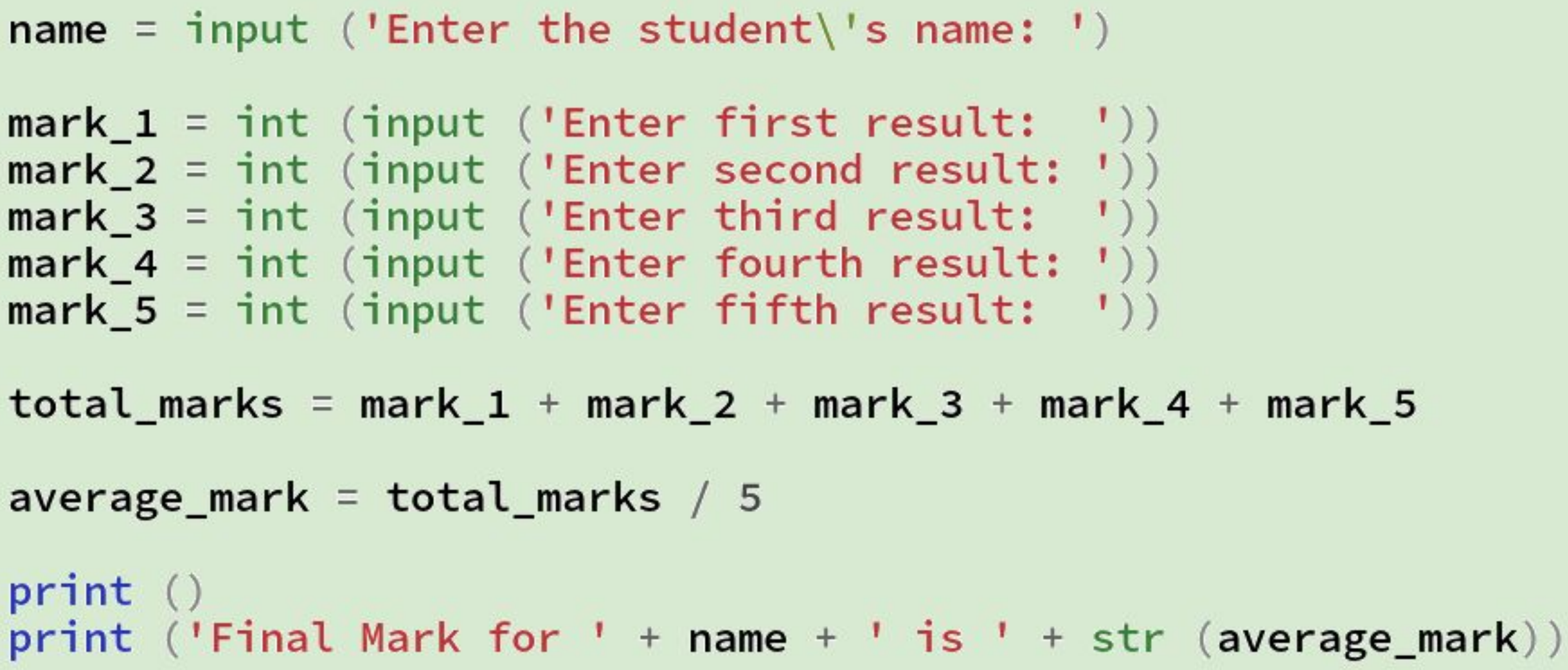
This is often a *bug*, when the programmer has forgotten to include code that might alter the condition.

There's actually still a bug in that loop. Can you see it?

Or maybe the "bug" is in the comment?

```
# Print 'Ni!' 10 times.

count = 1

while count < 10:
  print ('Ni!')
  count += 1
```

Back to Refactoring



```
name = input ('Enter the student\'s name: ')

mark_1 = int (input ('Enter first result:  '))
mark_2 = int (input ('Enter second result: '))
mark_3 = int (input ('Enter third result:  '))
mark_4 = int (input ('Enter fourth result: '))
mark_5 = int (input ('Enter fifth result:  '))

total_marks = mark_1 + mark_2 + mark_3 + mark_4 + mark_5

average_mark = total_marks / 5

print ()
print ('Final Mark for ' + name + ' is ' + str (average_mark))
```

Reading the Results

We can eliminate the five separate integers by reading the results into a better data structure.

As part of this we can just repeat the prompt.

We can also use a loop to make sure that the result entered is in the correct range.

And because we have met `range`, we can do it in a rather neat way.

Let's start with the prompt.

Reading the Results

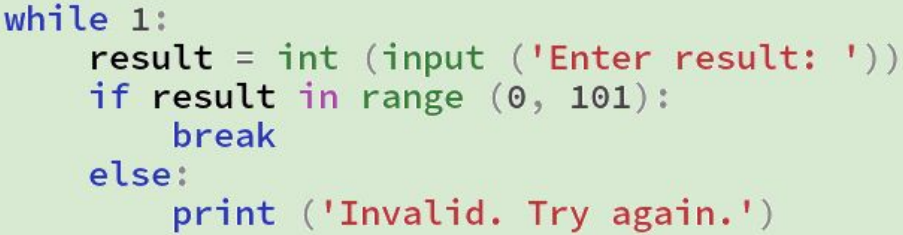
We can eliminate the five separate integers by reading the results into a better data structure.

As part of this we can just repeat the prompt.

We can also use a loop to make sure that the result entered is in the correct range.

And because we have met `range`, we can do it in a rather neat way.

Let's start with the prompt.



```
while 1:
    result = int (input ('Enter result: '))
    if result in range (0, 101):
        break
    else:
        print ('Invalid. Try again.')
```

Reading the Results

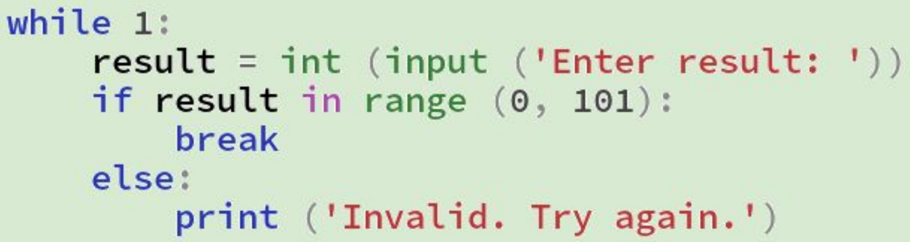
We can eliminate the five separate integers by reading the results into a better data structure.

As part of this we can just repeat the prompt.

We can also use a loop to make sure that the result

a

Remember
Programming is all about patterns and recipes.
This is a pattern you will meet again and again.

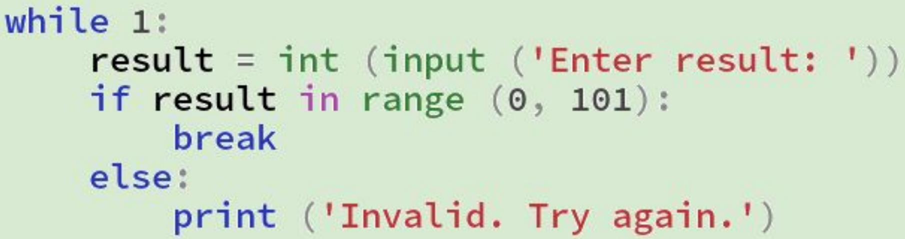


```
while 1:
    result = int (input ('Enter result: '))
    if result in range (0, 101):
        break
    else:
        print ('Invalid. Try again.')
```

Reading the Results

We can eliminate the five separate integers by reading the results into a better data structure.

So now, we can create an empty Tuple, and stick the new result on it each time.



```
while 1:
    result = int (input ('Enter result: '))
    if result in range (0, 101):
        break
    else:
        print ('Invalid. Try again.')
```

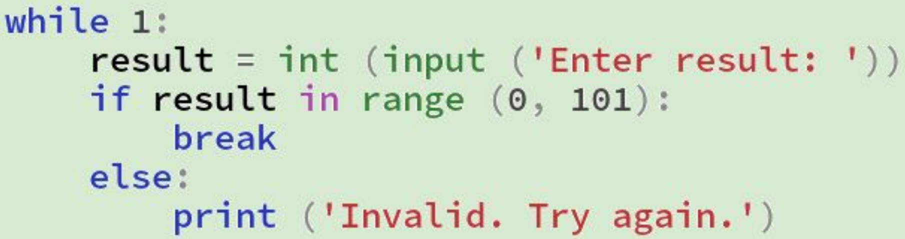
Reading the Results

We can eliminate the five separate integers by reading the results into a better data structure.

So now, we can create an empty Tuple, and stick the new result on it each time.

Really Important Note

Before doing this, we would check the code over there works!



```
while 1:
    result = int (input ('Enter result: '))
    if result in range (0, 101):
        break
    else:
        print ('Invalid. Try again.')
```

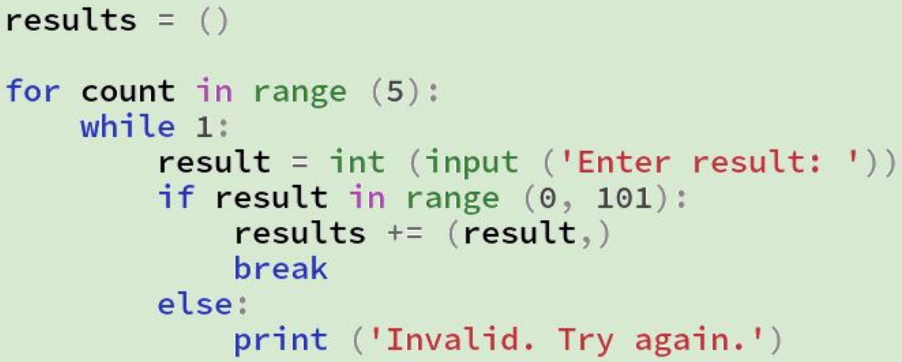
Reading the Results

We can eliminate the five separate integers by reading the results into a better data structure.

So now, we can create an empty Tuple, and stick the new result on it each time.

Really Important Note

Before doing this, we would check the code over there works!



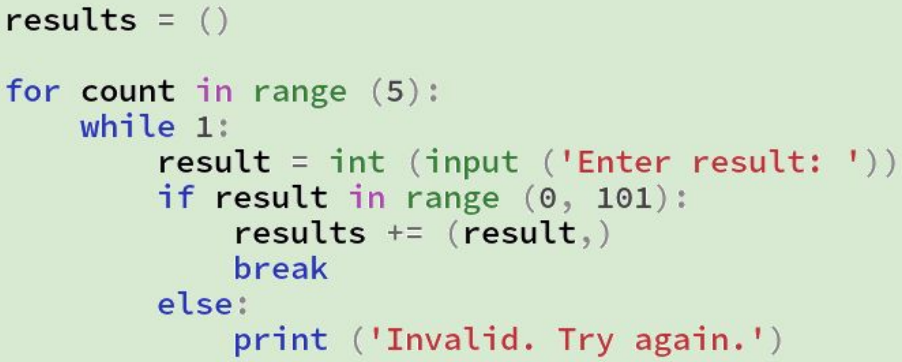
```
results = ()

for count in range (5):
    while 1:
        result = int (input ('Enter result: '))
        if result in range (0, 101):
            results += (result,)
            break
        else:
            print ('Invalid. Try again.')
```

Reading the Results

We can eliminate the five separate integers by reading the results into a better data structure.

So now, we can create an empty Tuple, and stick the new result on it each time.

This code is still assuming that the user is entering an integer. For the moment we are going to assume they are well behaved!



```
results = ()

for count in range (5):
    while 1:
        result = int (input ('Enter result: '))
        if result in range (0, 101):
            results += (result,)
            break
        else:
            print ('Invalid. Try again.')
```

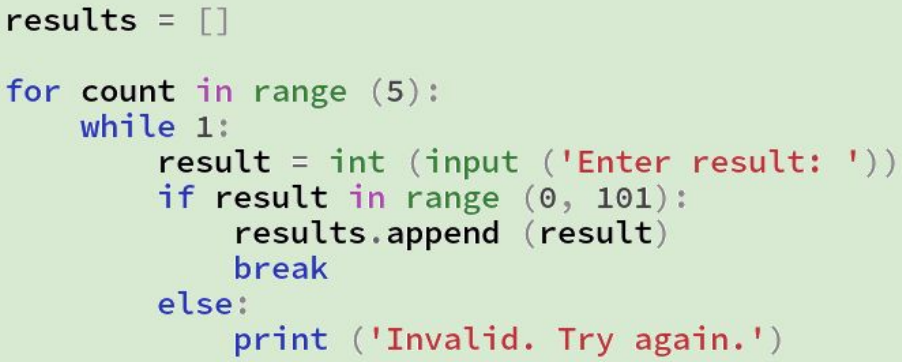
Reading the Results

We can eliminate the five separate integers by reading the results into a better data structure.

So now, we can create an empty Tuple, and stick the new result on it each time.

Now, I would have done this with a List, but either will work in this case.

See the difference?



```
results = []

for count in range (5):
    while 1:
        result = int (input ('Enter result: '))
        if result in range (0, 101):
            results.append (result)
            break
        else:
            print ('Invalid. Try again.')
```

Doing the Calculations

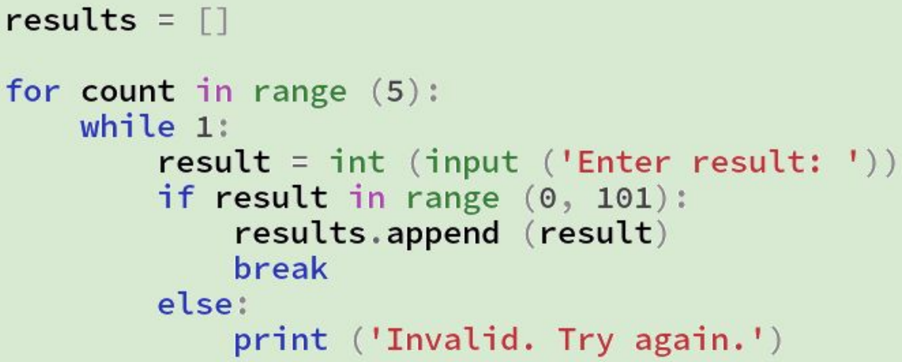
All that remains is to find the required statistic.

Because we have the numbers in a List, we can use some handy functions:

➢ `sum` will give the total of the results in the list.

➢ `len` will give the length of the list.

(We don't need `len`, but if we do use it our code will work for any number of results: let's keep the Technical Debt down!)



```
results = []

for count in range (5):
    while 1:
        result = int (input ('Enter result: '))
        if result in range (0, 101):
            results.append (result)
            break
        else:
            print ('Invalid. Try again.')
```

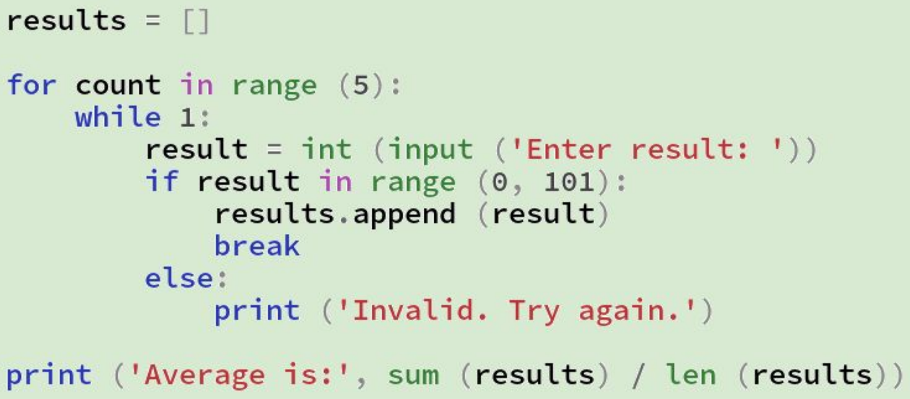
Doing the Calculations

All that remains is to find the required statistic.

Because we have the numbers in a List, we can use some handy functions:

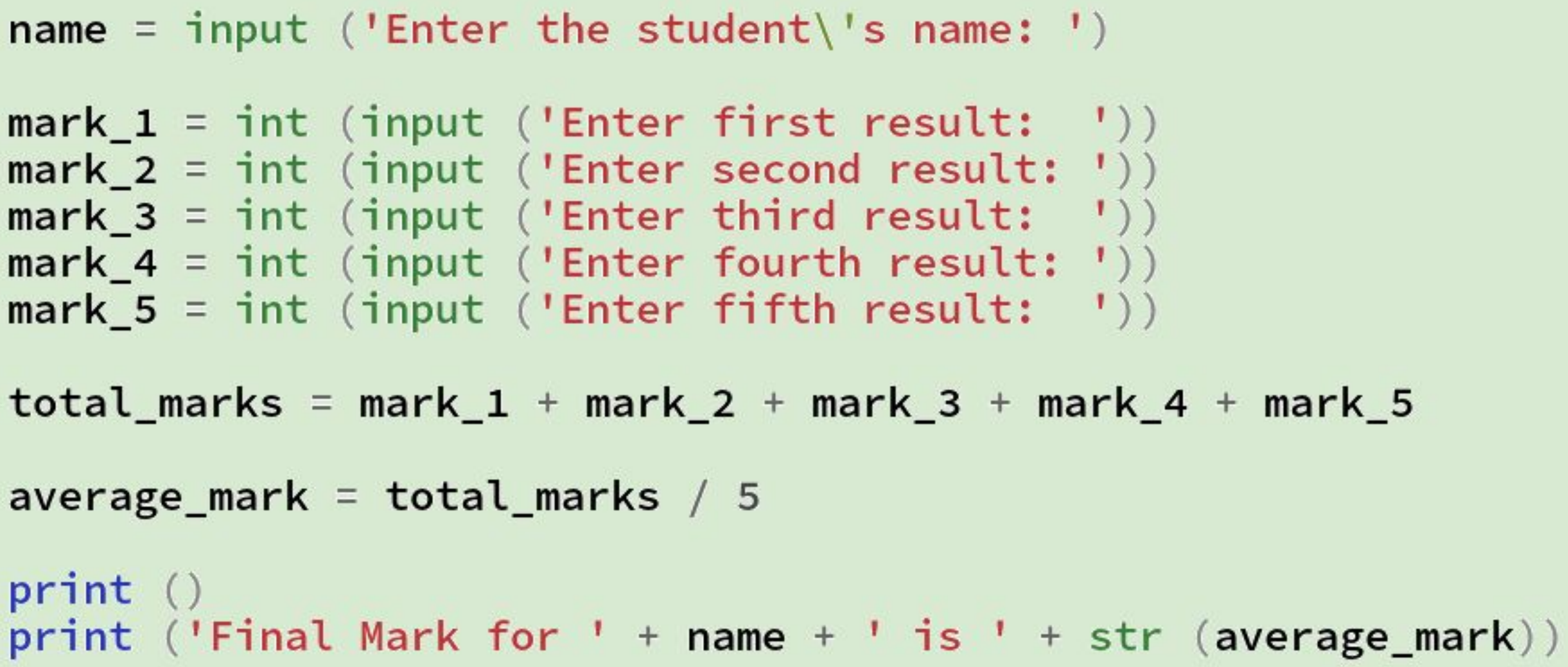
- `sum` will give the total of the results in the list.
- `len` will give the length of the list.

(We don't need `len`, but if we do use it our code will work for any number of results: let's keep the Technical Debt down!)



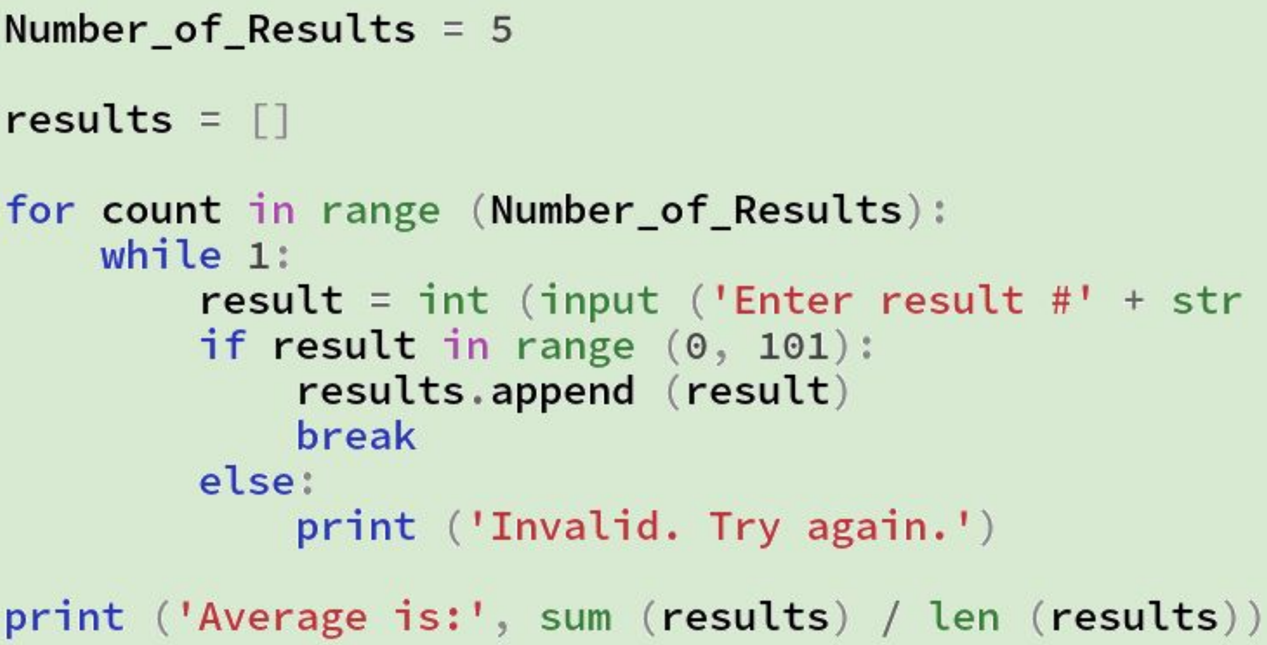
```
results = []

for count in range (5):
    while 1:
        result = int (input ('Enter result: '))
        if result in range (0, 101):
            results.append (result)
            break
        else:
            print ('Invalid. Try again.')

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

Refactoring



```
name = input ('Enter the student\'s name: ')

mark_1 = int (input ('Enter first result:  '))
mark_2 = int (input ('Enter second result: '))
mark_3 = int (input ('Enter third result:  '))
mark_4 = int (input ('Enter fourth result: '))
mark_5 = int (input ('Enter fifth result:  '))

total_marks = mark_1 + mark_2 + mark_3 + mark_4 + mark_5

average_mark = total_marks / 5

print ()
print ('Final Mark for ' + name + ' is ' + str (average_mark))
```

Refactored



```python
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))
```

PyCharm Demo and Question Time

Jobs

By next week, you should:

- Have read up to the end of Unit 4 in the book.
  - Worked through the examples.
- Be all up to date with practicals.

We've actually now "done programming".

What you need most now is practice.