University of
South Australia

Problem Solving and Programming

Week 4 – The List data type
– Control structures: for loops
– Problem Solving Process

Python Books

Course Textbook

Gaddis, Tony. 2012, *Starting Out with Python*, 2nd edition, Pearson Education, Inc.

Free Electronic Books

There are a number of good free on-line Python books. I recommend that you look at most and see if there is one that you enjoy reading. I find that some books just put me to sleep, while others I enjoy reading. You may enjoy quite a different style of book to me, so just because I say I like a book does not mean it is the one that is best for you to read.

- The following three books start from scratch - they don't assume you have done any prior programming:
  - The free on-line book "**How to think like a Computer Scientist: Learning with Python**", by Peter Wentworth, Jeffrey Elkner, Allen B. Downey and Chris Meyers, provides a good introduction to programming and the Python language. I recommend that you look at this book.
  - There is an on-line book "**A Byte of Python**" that is quite reasonable. See the home page for the book, or you can go directly to the on-line version for Python 3, or download a PDF copy of the book. This book is used in a number of Python courses at different universities and is another I recommend you look at.
  - Another good on-line book is "**Learning to Program**" by Alan Gauld. You can download the whole book in easy to print PDF format, and this is another book that would be good for you to look at.
- If you have done some programming before, you may like to look at the following:
  - **The Python Tutorial** - this is part of Python's documentation and is updated with each release of Python. This is not strictly an e-Book, but is book-sized.
  - **Dive into Python 3**, by Mark Pilgrim is a good book for those with some programming experience. I recommend you have a look at it. You can download a PDF copy.

UniSA

Problem Solving and Programming

- More on writing programs:
  - The `List` data type
  - Control structures: `for` loops
  - The `range()` function
  - Problem Solving Process

The List Data Type

Data Types - Lists

- Let's look at another data type called a **`list`**.
- A list is a sequence of zero or more objects.
- Lists support the same indexing and slicing as a string.
- The individual values inside a list are called items or elements.
- Lists can hold items/elements of any data type.
- Lists are mutable – meaning we can replace and delete any of their items.
  - Lists can be populated, empty, sorted, and reversed.
  - Individual or multiple items can be inserted, updated, or removed at will.

Data Types - Lists

- How to create and assign Lists.
  - Creating lists is as simple as assigning a value to a variable.
  - Lists are delimited by surrounding square brackets ([ ]).
  - Each element of a list is separated by a comma.

```
>>> animals = ['cat', 'dog', 'elephant', 'tiger']
>>> animals
['cat', 'dog', 'elephant', 'tiger']
```

```
>>> aList = [1, 2, 3, 4]
>>> aList
[1, 2, 3, 4]
```

UniSA

Data Types - Lists

- How to create and assign Lists.

```
>>> anotherList = [1, 'abc', 2, 'def', 3, 'ghi']
>>> anotherList
[1, 'abc', 2, 'def', 3, 'ghi']
```

anotherList

1	'abc'	2	'def'	3	'ghi'
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

index Forward

index Backward

Data Types - Lists

- How to access values in Lists.

```
>>> animals = ['cat', 'dog', 'elephant', 'tiger']
>>> animals[0]
'cat'

>>> animals[1]
'dog'

>>> animals[2]
'elephant'

>>> animals[3]
'tiger'

>>> animals[1:3]
['dog', 'elephant']
```

Data Types - Lists

- How to access values in Lists – more examples…

```
>>> aList = [1, 2, 3, 4]
>>> aList[:3]
[1, 2, 3]
>>> aList[-1]
4

>>> anotherList = [1, 'abc', 2, 'def', 3, 'ghi']
>>> anotherList[2:]
[2, 'def', 3, 'ghi']
>>> anotherList[::2]
[1, 2, 3]
>>> anotherList[1::2]
['abc', 'def', 'ghi']
>>> anotherList[3]
'def'
>>> anotherList[::-1]
['ghi', 3, 'def', 2, 'abc', 1]
```

Data Types - Lists

- List operations:
  - Concatenation (+)
  - Repetition (*)
  - `len()` function
  - `max()` and `min()` functions

For example:

```
>>> myList = [1, 7, 5, 2]
>>> len(myList)
4
>>> max(myList)
7
>>> min(myList)
1
>>> list1 = ['one', 'two', 'three']
>>> list2 = [1, 2, 3]
>>> list1 + list2
['one', 'two', 'three', 1, 2, 3]
>>> list1 * 2
['one', 'two', 'three', 'one', 'two', 'three']
```

Data Types - Lists

- List operations:
  - Membership (`in, not in`) – check whether an object is a member of a list:

```
>>> list1 = ['one', 'two', 'three']
>>> 'one' in list1
True
>>> 'four' in list1
False


if 'one' in list1:
    print('Found it!')


Found it!
```

Data Types - Lists

- How to update lists.
  - You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator.

```
>>> aList = [1, 2, 3, 4]
>>> aList[2] = 7
>>> aList
[1, 2, 7, 4]

>>> animals = ['cat', 'dog', 'elephant', 'tiger']
>>> animals[2] = 'lion'
>>> animals
['cat', 'dog', 'lion', 'tiger']
```

Data Types - Lists

- How to update lists – more examples…

```
>>> anotherList = [1, 'abc', 2, 'def', 3, 'ghi']
>>> anotherList[0:3] = [7, 'jkl', 8]
>>> anotherList
[7, 'jkl', 8, 'def', 3, 'ghi']
```

- You can append to the end or add a list to the start using slices:

```
>>> myList = [5,6,7]

>>> myList[:0] = [1,2,3]
>>> myList
[1, 2, 3, 5, 6, 7]

>>> myList[len(myList):] = [8, 9, 10]
>>> myList
[1, 2, 3, 5, 6, 7, 8, 9, 10]
```

UniSA

Data Types - Lists

- You can add elements to a list with the `append()` method.

```
>>> myList
[1, 2, 3, 5, 6, 7, 8, 9, 10]

>>> myList.append(11)
>>> myList
[1, 2, 3, 5, 6, 7, 8, 9, 10, 11]

>>> myList.append('twelve')
>>> myList
[1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 'twelve']
```

Data Types - Lists

- How to remove list elements and lists.

```
>>> myList
[1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 'twelve']
>>> del myList[3]
>>> myList
[1, 2, 3, 6, 7, 8, 9, 10, 11, 'twelve']

>>> del myList[1:4]
>>> myList
[1, 7, 8, 9, 10, 11, 'twelve']

>>> myList[2:4] = []
>>> myList
[1, 7, 10, 11, 'twelve']
```

`del` statement is preferred method of deleting values from a list – it is easier to read than the equivalent setting of a slice to an empty list.

UniSA

Data Types - Lists

- How to remove a value from a list using the `remove()` method.

```
>>> myList
[1, 7, 10, 11, 'twelve']

>>> myList.remove(11)
>>> myList
[1, 7, 10, 'twelve']

>>> myList.remove('twelve')
>>> myList
[1, 7, 10]
```

You can remove a value from a list using the remove() method. This removes the first instance of the given value from the list. It raises an exception if the given value is not in the list. You can either handle the exception (seen later in course) or check first that the value is in the list with the `in` operator.

Data Types - Lists

- Lists can contain other lists.

```
>>> animals = ['cat', 'dog', 'elephant', 'tiger']
>>> aList = [1, 2, 3, 4]
>>> anotherList = [1, 'abc', 2, 'def', 3, 'ghi']

>>> anotherList[0] = aList
>>> anotherList
[[1, 2, 3, 4], 'abc', 2, 'def', 3, 'ghi']

>>> anotherList[0]
[1, 2, 3, 4]
>>> anotherList[0][3]
4

>>> anotherList[5] = animals
>>> anotherList
[[1, 2, 3, 4], 'abc', 2, 'def', 3, ['cat', 'dog', 'elephant',
  'tiger']]

>>> anotherList[5][1]
'dog'
```

UniSA

Data Types - Lists

- Joining two lists together.
  - Don't use `append()` – it will add the second list as a single list element to the end of a list *(unless that's what you want to do)*.
  - Instead use the `extend()` method:

```
>>> myList = [1, 2, 3, 4]
>>> aList = [5, 6, 7, 8]
>>> myList.extend(aList)
>>> myList
[1, 2, 3, 4, 5, 6, 7, 8]
```

versus…

```
>>> myList.append(aList)
>>> myList
[1, 2, 3, 4, 5, 6, 7, 8, [5, 6, 7, 8]]
```

Data Types - Lists

- The `insert(m, value)` method inserts value(s) at the element in the list with index `m`.

```
>>> aList = [5, 6, 7, 8]
>>> aList
[5, 6, 7, 8]

>>> myList = ['one', 'two', 'three', 'four']
>>> myList
['one', 'two', 'three', 'four']

>>> myList.insert(2, aList)
>>> myList
['one', 'two', [5, 6, 7, 8], 'three', 'four']
```

Data Types - Lists

- Other list methods:
  - Reverse the order of the values in the list.

```
>>> myList = [1, 2, 3, 4]
>>> myList
[1, 2, 3, 4]
>>> myList.reverse()
>>> myList
[4, 3, 2, 1]
```

  - Sort the values in a list.

```
>>> myList = [4, 6, 1, 9]
>>> myList
[4, 6, 1, 9]
>>> myList.sort()
>>> myList
[1, 4, 6, 9]
```

Data Types - Lists

- Other list methods:
  - These methods *do not* change the list…
  - These methods return a value.
  - The `count()` method returns the number of times a value occurs in a list.

```
>>> myList = [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> myList
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> myList.count(1)
3
```

  - The `index()` method finds the index of the first instance of a value in a list.

```
>>> myList = [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> myList
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> myList.index(2)
1
```

Built-in Python Functions *(revisited…)*

Function	Description
len(s)	Returns the length (number of items) of s. The argument may be a sequence such as a string or list.
list([iterable])	Returns a list whose items are the same and in the same order as iterable's items. If no argument is given, returns a new empty list, [].
reversed(seq)	Returns a reverse seq.
sorted(iterable)	Returns a new sorted list from the items in iterable.
sum(iterable)	Returns the sum of items in iterable
ord(c)	Given a string of length one, returns an integer representing the value of the string.
chr(i)	Returns a string of one character whose ASCII code is the integer i.
enumerate(seq)	Returns an enumerate object. Useful for obtaining an indexed series.

Control Structures: `for` loops…

Repetition structures *(revisited…)*

- Loops are used when you need to repeat a set of instructions multiple times.
- Python supports two different types of loops:
  - `for` loop
  - `while` loop

- `while` loop
  - Good choice when you need to keep repeating the instructions until a criteria is met.
- `for` loop
  - Good choice when looping over a list of values.

    Different from the while loop, which loops as long as a certain condition is true.

Control Structures: `for` loop

- The `for` loop is very good at looping over a list of values.
- A `for` loop has the following form:

```
for iter_var in iterable:
    for_suite
else:
    else_suite
```

  - The `else` clause is optional.
  - If the individual elements are exhausted, the loop terminates, and if the optional `else` clause is present, its suite is executed.
  - `iter_var` is a variable.
  - `iterable` is a sequence (such as a list or string)

Control Structures: `for` loop

- Traverses through individual elements and terminates when all the items are exhausted.
- The loop is executed once for each element of the list identified in the first line.
- The variable in the for loop takes on the value of each item in the list.

  An example:

```
for k in [1, 3, 7]:
    print(k)
```

output:

```
1
3
7
```

Or…

```
myList = [1, 3, 7]
for k in myList:
    print(k)
```

iteration variable is `k`

UniSA

Control Structures: `for` loop

- Iterating over a sequence – such as a string:
  - Iteration variable will always consist of only a single character (string of length 1).

```
myString = 'someString'
for letter in myString:
    print(letter)
```

Output: ??

```
myString = 'someString'
for letter in myString:
    print(letter, end=' ')
```

Output: ??

Control Structures: `for` loop

- Iterating over a sequence – such as a string:
  - Iteration variable will always consist of only a single character (string of length 1).

```
myString = 'someString'
for letter in myString:
    print(letter)
```

Output:

```
s
o
m
e
S
t
r
i
n
g
```

```
myString = 'someString'
for letter in myString:
    print(letter, end=' ')
```

Output:

```
s o m e S t r i n g
```

Control Structures: `for` loop

- Iterating over a sequence – such as a string:
  - Another example:

```
str1 = 'Over The Top'
new_string = ''

index = 0
for letter in str1:
    if letter.isupper():
        new_string += str1[index]

    index = index + 1

print(new_string)
```

Output:

```
OTT
```

UniSA

Control Structures: `for` loop

- Iterating over a sequence – such as a list:
  - Iteration variable (name) will be the list element that we are on for that particular iteration of the loop.

```
nameList = ['kramer', 'jerry', 'elaine', 'george']
for name in nameList:
    print(name)
```

Output: ??

Control Structures: `for` loop

- Iterating over a sequence – such as a list:
  - Iteration variable (name) will be the list element that we are on for that particular iteration of the loop.

```
nameList = ['kramer', 'jerry', 'elaine', 'george']
for name in nameList:
    print(name)
```

Output:

```
kramer
jerry
elaine
george
```

Control Structures: `for` loop

- Iterating over a sequence – such as a list:
  - Another example:

```
num_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for num in num_list:
    if num % 2 == 0:
        print(num)
```

Output:

```
2
4
6
8
10
```

The `range()` Function

- The `range()` function is used to generate a progression of integer values.

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

- Most often used in for loops. The arguments must be integers. The step value defaults to 1 if omitted. If the start argument is omitted, it defaults to 0.
- When called with one argument, range() will return a list of integers from 0 up to (but not including) the argument.
- If used with two arguments, the list of integers it returns is from the first argument up to (but not including) the second argument.
- Can specify a different increment using step.

The `range()` Function

- Examples:

```
>>>list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>>list(range(3, 7))
[3, 4, 5, 6]

>>>list(range(0,10,2))
[0, 2, 4, 6, 8]

for n in range(10):
    print(n)
```

output:

```
0
1
2
3
4
5
6
7
8
9
```

UniSA

Control Structures: `for` loop

- Iterating using an index – list example:
  - Rather than iterating through the elements themselves, we are iterating through the indices of the list.
  - To do so, we combine the `range()` and `len()` functions as follows:

```
nameList = ['kramer', 'jerry', 'elaine', 'george']
for nameIndex in range(len(nameList)):
    print(nameIndex, nameList[nameIndex])
```

Output: ??

Control Structures: `for` loop

- Iterating using an index – list example:
  - Rather than iterating through the elements themselves, we are iterating through the indices of the list.
  - To do so, we combine the `range()` and `len()` functions as follows:

```
nameList = ['kramer', 'jerry', 'elaine', 'george']
for nameIndex in range(len(nameList)):
    print(nameIndex, nameList[nameIndex])
```

Output:

```
0 kramer
1 jerry
2 elaine
3 george
```

UniSA

Control Structures: `for` loop

- Iterating using an index – string example:
  - You can access the individual characters in a string with an index. Each character in a string has an index that specifies its position in the string.
  - To do so, we combine the `range()` and `len()` functions as follows:

```
myString = 'someString'
for index in range(len(myString)):
    print(index, myString[index])
```

Output: ??

Control Structures: `for` loop

- Iterating using an index – string example:
  - You can access the individual characters in a string with an index. Each character in a string has an index that specifies its position in the string.
  - To do so, we combine the `range()` and `len()` functions as follows:

```
myString = 'someString'
for index in range(len(myString)):
    print(index, myString[index])
```

Output:

```
0 s
1 o
2 m
3 e
4 S
5 t
6 r
7 i
8 n
9 g
```

Control Structures: `for` loop

- The for loop is similar to the while loop, but when you only need to iterate over items in a list, using a for loop requires less code.
- You can make a while loop that acts the same way as a for loop by adding extra code:

  For example:

```
nameList = ['kramer', 'jerry', 'elaine', 'george']
for name in nameList:
    print(name)
```

Or…

```
nameList = ['kramer', 'jerry', 'elaine', 'george']
for nameIndex in range(len(nameList)):
    print(nameList[nameIndex])
```

Versus…

```
nameList = ['kramer', 'jerry', 'elaine', 'george']
index = 0
while (index < len(nameList)):
    print(nameList[index])
    index = index + 1
```

UniSA

Problem Solving Strategy

UniSA

Reference

- Robertson, A. *Simple Program Design*. 5th edition. 2006. Thomson.

Problem Solving Strategy

- Moving from problems to code…

- Use a systematic problem solving strategy:

1. Define the problem.
2. Develop an algorithm.
3. Test the algorithm for correctness.
4. Implement algorithm in chosen programming language (Python).
5. Test and verify program.

Problem Solving Strategy

- **1. Define the Problem.**
  - If you don't have a clear understanding of the problem, it is unlikely that you'll be able to solve it.
  - Steps to help you understand the problem:
    - Carefully read the problem until you understand what is required.
    - Divide the problem into:
      - Input
      - Output
      - Processing - List of steps/actions needed to produce the output
  - Underline words identifying the inputs and outputs.
    - Look for nouns
  - CAPITALISE words identifying processing actions.
    - Look for verbs

Problem Solving Strategy

- **1. Define the Problem.**
  - Strategy:
    - What is the desired outcome?
    - What do I need to know first to reach this goal?
    - What steps do I need to take? (look for verbs, in order)

Problem Solving Strategy

- **2. Develop an Algorithm.**
  - List the detailed set of instructions you need to perform to solve the problem.
  - The solution to any computing problem involves executing a series of actions in a specific order.
  - What is an algorithm?
    - An algorithm is like a recipe, a set of instructions that are:
      - Detailed
      - Precise
      - Ordered
    - A procedure for solving a problem in terms of:
      - The actions to be executed, and
      - The order in which these actions are to be executed.
  - The goal of an algorithm is to complete some task.
  - An algorithm is written in simple English.

Problem Solving Strategy

- **2. Develop an Algorithm.**
  - Correctly specifying the order in which the actions are to be executed is important.
  - An Exercise:
    - Write an algorithm to get out of bed and arrive at Uni for a lecture…

Problem Solving Strategy

- **2. Develop an Algorithm.**
  - Write an algorithm to get out of bed and arrive at uni for a lecture…

    Get out of bed
    Eat breakfast
    Take off pyjamas
    Take a shower
    Brush teeth
    Get dressed
    Drive to Uni
  - Suppose that the steps are performed in a slightly different order… trouble!!

    Get out of bed
    Take off pyjamas
    Get dressed
    Take a shower
    Brush teeth
    Eat breakfast
    Drive to Uni

Problem Solving Strategy

- **2. Develop an Algorithm.**
  - Design a solution and develop into an algorithm…
    - You may like to work backwards:

      What outputs are required?
      What data do you need to be able to calculate the outputs?
      Where can you get that data? (user inputs, constants, derived values, other programs).
  - You may use pseudocode to draft your solution…

Problem Solving Tool

- What is Pseudocode?
  - Pseudocode is an informal language that helps you develop algorithms.
  - Pseudocode is a tool used to plan your program before you start coding.
  - It helps you “think out” a program before attempting to write it in a programming language such as Python.
  - Pseudocode is a verbal description of your plan.
    - Similar to programming code.
    - Design an algorithm in pseudocode.
    - Structured, formalized, condensed English.
    - Should not resemble any particular programming language (ignore syntax).
  - You may use pseudocode in order to help you solve your problems.
  - You can design solutions without knowing a programming language.
  - Intended to help you create better computer programs.

Problem Solving Tool

- 3 Control Structures
  - Sequence
  - Selection
  - Repetition

- Pseudocode uses common words and keywords to symbolise these operations.

  For example:

Pseudocode:

```
IF time is greater than 7
  print 'time to go home'


k = 0
WHILE k is less than 3
  k = k + 1
  print k to screen
```

Python code:

```
if time > 7:
   print('Time to go home')


k = 0
while k < 3:
   k = k + 1
   print(k)
```

UniSA

Problem Solving Tool

- For example *(continued…)*:

Pseudocode:

```
k = 0
WHILE k < 3
   k = k + 1
   print k to screen
```

```
FOR k in 1 to 3
   print k to screen
```

Python:

```
k = 0
while k < 3:
    k = k + 1
    print(k)
```

```
for k in [1, 2, 3]:
   print(k)
```

UniSA

Problem Solving Strategy

- **2. Develop an Algorithm.**
  - Write down a set of steps to solve the problem.
  - Take the processing steps from step 1 (defining the problem).
  - Take the 3 basic control constructs:
    - Sequence
    - Selection
    - Repetition
  - Determine how the processing will be performed.
    - Sometimes a trial and error process.
  - Each processing step relates to **1 or more steps** in the algorithm.
  - Once an algorithm is fully developed and tested, implementing it in a particular programming language is relatively trivial.

Problem Solving Strategy

- **3. Test the algorithm for correctness.**
  - To detect errors early.
  - To make sure the algorithm is correct.

    i.e. produces the correct results.

  - Hand 'execute' your algorithm:
    - Work through your test cases and see if your algorithm handles these and gives the right answer.
    - Look for ambiguous or missing steps.
    - Look for steps that do a lot – these may need to be broken down further.

Problem Solving Strategy

- **3. Test the algorithm for correctness.**
  - To test if your approach to solving the problem will work, you need test cases where you know what the result or output should be.
  - You need one or more situations that are 'typical' and where the algorithm should work and you are able to specify what the algorithm should do.

1. Choose 2 simple sets of valid input values.
   - Select test data based on the requirements, not your algorithm
   - As you're not using a computer to calculate, keep values simple: 10, 20, 30 is easier than 3.75 2.89 and 5.31
2. Determine expected results.
3. Step through algorithm with first test data set.
4. Repeat with other test data set.
5. Check that results from steps 3 and 4 match expected results.

Problem Solving Strategy

- **3. Test the algorithm for correctness.**
  - Look for boundaries
    - Are there input ranges where there is no solution or the algorithm will not work?
    - Should the algorithm work differently for different ranges of inputs?
    - Look for special cases.
    - These are often associated with a boundary, such as first or last value of an input.

Problem Solving Strategy

- **4. Implement algorithm in chosen programming language.**
  - Create a Python solution.
    - Convert the algorithm into a Python solution.
  - Include comments in the solution.
    - Helps others follow your work (as well as yourself).
  - Generally better to do this incrementally – in small steps where you execute and test the code you have just added. This is important!

Problem Solving Strategy

- **4. Implement algorithm in chosen programming language.**
  - Follow good coding standards as you go.

    This includes:
    - Use of sensible and meaningful variable names.
    - Use of consistent indentation.
    - Leave some 'white space' to improve readability (i.e. blank lines, appropriate spacing between operators, etc).
    - Comment interesting / significant bits of code.
    - Use of comments to describe functions.

  - Tidy up your code as you go – keep it readable and take out redundant test code.

Problem Solving Strategy

- **5. Test and verify the solution / program.**
  - Compare to the hand solution / algorithm test.
  - Do your answers make sense?
  - Do they match your sample calculations?
  - Is your answer what was asked for?

Problem Solving Strategy

- Here's an example:

  Given a wall which has a width of 5 metres and a height of 10 metres, write a program that computes and displays the amount of paint required to paint the wall. Paint coverage is 10 square metres per litre.

Problem Solving Strategy

- 1. Define the problem

  Given a wall which has a width of 5 metres and a height of 10 metres, write a program that COMPUTES and DISPLAYS the amount of paint required to paint the wall. Paint coverage is 10 square metres per litre.

  You may like to restate the problem in your own words…

  Find the amount of paint required to paint a wall.

Problem Solving Strategy

- 1. Define the problem
  - Inputs

    Wall width width = 5 m

    Wall height height = 10 m

  - Output

    Quantity of paint (quantity in litres)

  - Processing

    Compute the area of a wall

    Compute amount of paint required

    Display amount of paint required

Problem Solving Strategy

- 2. Develop an algorithm.
  - The area of a rectangle can be calculated using the following formula:

    area = width x height

  - Compute the area of the wall

    area = width x height

    = 5 m x 10 m

    = 50 m^2

  - Compute the amount of paint required

    quantity = area / 10

    = 50 m^2 / 10 m^2

    = 5 litres of paint required

Problem Solving Strategy

- 2. Develop an algorithm.
  - List the detailed set of instructions you need to perform to solve the problem.
    - The hand solution (if you have one) will help you do this.
    - The processing actions identified in step 1 will also help you do this.
  - Thus, the algorithm (detailed set of instructions) is as follows:

```
Set width to 10
Set height to 5
Compute area of wall  area = width x height
Compute quantity of paint  quantity = area / 10
Display the quantity of paint to the screen
```

Problem Solving Strategy

- 3. Test the algorithm for correctness.

- 4. Develop a Python solution.
  - Convert the algorithm into a Python solution.

```
width = 10
height = 5
area = width * height
quantity = area / 10
print('Quantity of paint required:', quantity)
```

Problem Solving Strategy

- 5. Test and verify the solution / program.
  - Compare to the hand solution (if you have one) or the testing from step 3.
  - Do your answers make sense?
  - Do they match your sample calculations?
  - Is your answer what was asked for?

Problem Solving Strategy

- Summary:
  - Analyse then Design then Test then Implement then Test.
  - Pseudocode is structured English.
    - Don't need to worry about syntax.
    - Focus on the most important aspect – design.
    - Good way to focus on solving the problem without getting stuck on the syntax/details of a programming language.
  - Let the problem statement guide you to the solution.
    - Identify the nouns (often input/output).
    - Identify the verbs (often the processing steps) in order.
  - Pseudocode can be translated into nearly any programming language.
  - Break up larger problems into a set of smaller ones.

End of Week 4