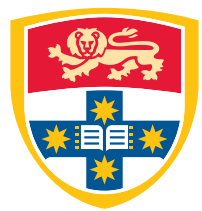
# INFO1110 & COMP9001: Introduction to Programming

School of Information Technologies, University of Sydney



# Copyright Warning

# Week 8: Programming Idioms, Modules

We will cover: Solving common problems

You should read: page 107(slicing), and §§2.2

# Lecture 15: Programming idioms

*Identifying common problems and their solutions*

# Loops revisited

`while` loop is simple and good

only need to consider the condition, when the loop keeps going or stops

other code can contribute to the setup and change in the loop condition

e.g. a counter variable initialised, checked, updated, checked, updated, checked, etc.

```
1 counter = 0 # initialise
2 while counter < 10: # check condition
3     counter = counter + 1  # update something in each iteration
```

Most loops follow a general structure. For loops are not discussed here, nor will it be allowed in certain written assessment questions.

# Programming Idioms

*Common tasks*

# Programming Idioms

There are many small problems to solve in programming

A programming idiom is a commonly used method to solve a small problem

These are all related to search

Find the smallest of two numbers

```
1 if :x < y:
2     # x is the smallest
3 else
4     # y is the smallest
```

# Programming Idioms

Loop exactly $N$ times

```
1  i = 0
2  while i < N:
3      # things to do each time
4      i += 1
```

Creates sequence for values of `i` as `0, 1, 2, 3, ...  N-3, N-2, N-1`

What if we do not know $N$?

# Programming Idioms (cont.)

STOP when you have reached the goal

```
1  N = 1000000
2  i = 0
3  while i < N:
4      # ...
5      if goal is satisfied:
6          break
7      # ...
8      i += 1
```

# Programming Idioms (cont.)

## STOP when you have reached the goal

```
N = 1
i = 0
while i < N:
    # ...
    if goal is satisfied:
        break
    # ...
    if some condition:
        i = N
    # ...
if i == N:
    # ...
```

## STOP when you have reached the goal

```
while True:
    # ...
    if goal is satisfied:
        break
    # ...
```

# Programming Idioms (cont.)

STOP when you have reached the goal

```
while True:
    # ...
    if goal is satisfied in one way:
        break
    # ...
    if goal is satisfied in another way:
        break
    # ...
    if goal is never satisfiable:
        break
    # ...
```

Which one is useful for us?

# Sequence control flow: reverse

Reversing the sequence

Always check your indices

Standard convention where `i` has no dependencies

```
1  i = N - 1
2  while i >= 0:
3      print(i)
4      print(numbers[i])
5
6      i = i - 1
```

Creates sequence for values of `i` as `N-1, N-2, N-3, ... 3, 2, 1, 0`

Prints the values of `i` with first value `N-1` Prints the values in the array in reverse order, last element first, index $N-1$

# Sequence control flow: reverse (cont.)

Standard convention where `i` **has dependencies**

```
1  name_order = [ "1st", "2nd", "3rd", "4th", ... ]
2  i = 0
3  while i < N:
4      print(i)
5
6      # do something with i that must have an ascending sequence
7      print(name_order[i], end=' ')
8
9      print(numbers[N - i - 1])
10
11     i = i + 1
```

Creates sequence for values of `i` as `0, 1, 2, 3, ... N-3, N-2, N-1`

Prints the values of `i` with first value `0`

Prints the values in the array in reverse order, last element first, index $N-1$

# Sequence control flow: reverse (cont.)

Use another variable to store the reversed index to simplify code

```
1  i = 0
2  while i < N:
3      r_index = N - i - 1
4      print(numbers[r_index])
5
6      i = i + 1
```

Use another variable to store the reversed array element to simplify code

```
1  i = 0
2  while i < N:
3      r_number = numbers[N - i - 1]
4      print(r_number)
5
6      i = i + 1
```

# Sequence control flow: subsequences

It is not always necessary to move through all $N$ elements

e.g. print array values starting from element 3 to 7

Questions:

- Terminology: is element the same as index? or is 1st element index 0, hence element 3 is index 2?
- starts from element 3, inclusive. Does it also print element 7, or end at element 7 and not print. Mathematics ranges are helpful as communication of indices
  Includes element 7: $[2,6]$ or $[2,7)$
  Excludes element 7: $[2,5]$ or $[2,6)$
- Is $N$ less than 7? less than 3? What to do?

# Sequence control flow: subsequences (cont.)

Let narrow the problem to print values at indices `2, 3, 4, 5, 6`

Version 1 - inclusive index range $[2, 6]$

```
1  i = 2
2  while i <= 6:
3      print(numbers[i])
4
5      i = i + 1
```

# Sequence control flow: subsequences (cont.)

Same. print values at indices `2, 3, 4, 5, 6`

Version 2 - exclusive index range $[2, 7)$

```
1  i = 2
2  while i < 7:
3      print(numbers[i])
4
5      i = i + 1
```

# Sequence control flow: subsequences (cont.)

Same. print values at indices `2, 3, 4, 5, 6`

Version 3 - we know it is 5 times, make a variable for offset from beginning

```
1  start = 2
2  i = 0
3  while i < 5:
4      print(numbers[i + start])
5
6      i = i + 1
```

# Sequence control flow: subsequences (cont.)

Same. print values at indices `2, 3, 4, 5, 6`

Version 4 - make a variable for start and end

```
1  start = 2
2  end = 7
3  i = start
4  while i < end:
5      print(numbers[i])
6
7      i = i + 1
```

# Sequence control flow: subsequences (cont.)

Same. print values at indices `2, 3, 4, 5, 6`

Version 5 - move through the sequence but keep `i` nice and tidy [0,N-1]

```
1  start = 2
2  end = 7
3  N = end - start
4  i = 0
5  while i < N:
6      print(numbers[i + start])
7
8      i = i + 1
```

How easy is it to identify the number of iterations among the five versions?

# Sequence control flow: subsequences (cont.)

Obligatory python slice :

Used in indices of array operators, return a new collection that is a subsequence (not necessarily smaller)

```
1 numbers = [ 0, 1, 2, 3, 4, 5, 6 ]
2 subnum = numbers[1:3]
3 print(subnum)
```

```
[1, 2]
```

Read as `array[start:end]`, where the indices are [*start*, *end*)

Convention of reading sequences as `[0,N-1]` reigns supreme!

# Sequence control flow: subsequences (cont.)

## Other tricks with slicing

Copy sequence all values starting from index *start* up to and including $N-1$

```
1 numbers = [ 0, 1, 2, 3, 4, 5, 6 ]
2 subnum = numbers[3:] # [3, N )
```

Copy sequence all values starting from index 0 up to and including *end* $-1$

```
1 numbers = [ 0, 1, 2, 3, 4, 5, 6 ]
2 subnum = numbers[:3] # [ 0, 3 )
```

Copy entire sequence starting from index 0 up to and including $N-1$

```
1 numbers = [ 0, 1, 2, 3, 4, 5, 6 ]
2 subnum = numbers[:] # [ 0, N - 1 )
```

# Sequence control flow: even/odd

Executing instructions for every 2nd number

Even and odd numbers

```
1  i = 0
2  while i < N:
3      if i % 2 == 0:
4          # do something when even sequence
5      else:
6          # do something when odd sequence
7
8      i += 1
```

# Sequence control flow: even/odd (cont.)

## Even and odd numbers: Another way

```
1  i = 0
2  is_even = True
3  while i < N:
4      if is_even:
5          # do something when even sequence
6      else:
7          # do something when odd sequence
8
9      is_even = not is_even
10     i += 1
```

# Sequence control flow: even/odd (cont.)

## Even and odd numbers: Another way

```
1  i = 0
2  counter = 0
3  while i < N:
4      if counter == 0:
5          # do something when even sequence
6      else:
7          # do something when odd sequence
8
9      counter += 1
10     if counter == 2:
11         counter = 0
12
13     i += 1
```

## Even and odd numbers: Another way using -1...how?

# Sequence control flow: even/odd (cont.)

## Only even

```
1  i = 0
2  while i < N:
3      # do something when even sequence
4
5      i += 2
```

## Only odd

```
1  i = 1
2  while i < N:
3      # do something when odd sequence
4
5      i += 2
```

# Sequence control flow: even/odd (cont.)

You can play with the two adjacent elements, the odd and even in many ways and in many orderings. Consider these.

all odds, then all evens

all evens, then all odds

interleaved reverse, with the first element being always even, then odd, then even...? i.e. N, N-1, N-2, N-3

Write the code to print this sequence for N=10:
`0, 9, 1, 8, 2, 7, 3, 6, 4, 5`

Similar to the previous sequence, write the code to make the sequence for $N > 0$:
`0, N-1, 1, N-2, 2, N-3, 3, N-4, 4, N-5, 5, N-6, 6, N-7, 7...`

# Sequence control flow: every nth

Executing instructions for every *n*th number

Special sequence: every 7th number OR every 3rd number

```
# 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
# ^        ^        ^  ^     ^              ^        ^   ^
i = 0
while i < N:
    if (i % 7 == 0) or (i % 3 == 0):
        # do something

    i += 1
```

# Sequence control flow: every nth (cont.)

Do one thing every 7th number, otherwise a different thing every 3rd number

```
1  i = 0
2  while i < N:
3        if i % 7 == 0:
4         # do something when 0, 7, 14, 21
5        elif i % 3 == 0:
6         #do something when 3, 6, 9, 12
7         # WILL NOT HAPPEN when i == 0
8
9      i += 1
```

# Sequence control flow: every nth (cont.)

What about this?

```
1  i = 0
2  while i < N:
3      if i == 0:
4              ...statement block X...
5              continue
6      if i >= 54 and i < 60:
7              ...statement block Y...
8      else:
9              ...statement block Z...
10
11     i += 1
```

What is the execution order of blocks `X`, `Y` and `Z`?

# Idioms with arrays

Programs tend to work with same data but different size. e.g. sum of $N$ numbers, where $N$ could be small, or really big

The data is stored in some kind of memory, could be on disk as a file, or could be in main memory

To solve the problem, we have to search. We need to **visit** each number that is somewhere in memory/file

An array is the most typical way to represent $N$ pieces of data, where each piece is the same data type (e.g. Integer)

The idiom is to scan the entire array and *do something* with each value

# Idiom: computation over all values

Search for: The sum of N numbers[1]

```
1  def calculate_total(numbers):
2      N = len(numbers)
3
4      i = 0
5      total = 0
6      while i < N:
7          total += numbers[i]
8
9          i += 1
10
11     return total
```

[1] python has a built function called `sum`, we will use the word total

# Idiom: search for a specific value

Search for: does the number 42 exist in the array? Yes/No

```
1  def contains_42(numbers):
2      N = len(numbers)
3      i = 0
4      found = False
5      while i < N:
6          if numbers[i] == 42:
7              found = True
8
9          i += 1
10
11     return found
```

# Idiom: search for a specific value (cont.)

Has our search ended?

```
1  def contains_42(numbers):
2      N = len(numbers)
3      i = 0
4      found = False
5      while i < N:
6          if numbers[i] == 42:
7              found = True
8              break
9
10         i += 1
11
12     return found
```

# Idiom: search for a specific value (cont.)

Do we have all the answers we need for our search?

```
1  def contains_42(numbers):
2      N = len(numbers)
3      i = 0
4      while i < N:
5          if numbers[i] == 42:
6              return True
7
8          i += 1
9
10     return False
```

# Idiom: search for a specific value (cont.)

Is there another way to detect if the search succeeded or failed?

```
1  def contains_42(numbers):
2      N = len(numbers)
3      i = 0
4      while i < N:
5          if numbers[i] == 42:
6              break
7
8          i += 1
9
10     return ?
```

# Idiom: search for a specific value (cont.)

We have made many versions, is this only going to work for 42?
yes

Can we make it work for any value?
yes!

```
1  def contains_value(numbers, value):
2      N = len(numbers)
3      i = 0
4      while i < N:
5          if numbers[i] == value:
6              return True
7
8          i += 1
9
10     return False
```

# Idiom: search for a specific value

Search for: The number of occurrences of a value - counting

```
1  def count_number_of_42(numbers):
2      N = len(numbers)
3      count = 0
4      i = 0
5      while i < N:
6          if numbers[i] == 42:
7              count += 1
8
9          i += 1
10
11     return count
```

# Idiom: search for a specific value (cont.)

Easily modified to work for any value

```
1  def count_numbers_with_value(numbers, value):
2      N = len(numbers)
3      count = 0
4      i = 0
5      while i < N:
6          if numbers[i] == value:
7              count += 1
8
9          i += 1
10
11     return count
```

# Idiom: search for a specific value (cont.)

How do you know the search failed?

We have two pieces of information

#1

```
1    if count == 0:
2        # did not find any occurrences of value
```

and #2?

# Idiom: search for a specific value

Search for: The minimum of $N$ numbers

```
1  def find_minimum(numbers):
2      N = len(numbers)
3      i = 0
4      min = 1000000
5      while i < N:
6          if min > numbers[i]:
7              min = numbers[i]
8
9          i += 1
10
11     return min
```

Think about the input cases. There are at least two situations this will fail. What are they?

# Idiom: search for a specific value (cont.)

Search failed? can we guarantee with this?

```
1 if min == 1000000:
2     # no minimum found
```

We need a bigger number, the largest possible...

# Idiom: search for a specific value (cont.)

```
1  def find_minimum(numbers):
2      N = len(numbers)
3      min = sys.float_info.max # largest possible number value
4      i = 0
5      while i < N:
6          if min > numbers[i]:
7              min = numbers[i]
8
9          i += 1
10
11     return min
```

An improvement, can we still say this is always correct?

```
1  if min == sys.float_info.max:
2      # no minimum found
```

... are you sure?

# Idiom: search for a specific value (cont.)

It would make sense to start defining what happens where there is no minimum. e.g. an empty list.

We have some design decisions here:

- return `None`
- return the largest possible value
- throw an exception
- change the function prototype and return an error code

# Idiom: search for a specific value (cont.)

```
1  def find_minimum(numbers):
2      if numbers is None: # or (not (numbers is <expected type>))
3          # do special thing
4          # return ...
5          # raise ...
6      N = len(numbers)
7      if N == 0:
8          # do special thing
9          # return ...
10         # raise ...
11
12     min = sys.float_info.max # largest possible number value
13     i = 0
14     while i < N:
15         if min > numbers[i]:
16             min = numbers[i]
17
18         i += 1
19
20     return min # always correct, even if largest
```

# Idiom: search for a specific value

Search for: The minimum of N *Objects*

What does minimum mean if it is not a number? e.g. for a String Object

What if we don't know the largest possible value in advance?

First solve this for numbers

```
1 def is_a_bigger_than_b(a, b):
2     '''Returns True if the value of a is bigger than value b'''
3     if a > b:
4         return True
5     else
6         return False
```

# Idiom: search for a specific value (cont.)

```
1  def is_a_bigger_than_b(a, b):
2      '''Returns True if the value of a is bigger than value b'''
3
4  def find_minimum(numbers):
5      N = len(numbers)
6      if N == 0:
7          # do special thing
8
9      # guaranteed at least one element from here
10     min = numbers[0]
11     i = 1     # start from 1
12
13     while i < N:
14         # we have some way of asking if min > numbers[i]
15         if is_a_bigger_than_b(min, numbers[i]) == True:
16             min = numbers[i]
17
18         i += 1
19
20     return min
```

# Idiom: search for a specific value (cont.)

Apply to any object

```
1  def find_minimum(things):
2      N = len(things)
3      if N == 0:
4          # do special thing
5
6      # guaranteed at least one element from here
7      min = things[0]
8      i = 1     # start from 1
9
10     while i < N:
11         # we have some way of asking if min > things[i]
12         if is_a_bigger_than_b(min, things[i]) == True:
13             min = things[i]
14
15         i += 1
16
17     return min
```

# Idiom: search for a specific value (cont.)

Define a way to measure if one object value is bigger than another. There only needs to be a yes/no answer.

Consider strings. What makes one string bigger than another?

Create several discriminators.

# Idiom: general idea with arrays

- A search task
- input is some array
- there is some criteria for the search
- there is a failed search outcome that needs to be considered

```
1  input: array[N]
2  output: result
3
4  LOOP over each element of the array
5      if search criteria:
6          update result
7
8  if search failed criteria
9      error handling/reporting
10
11 return result
```

# Solve these small problems

Each problem can be solve using the general form of an idiom.

Given an array of integers. Increment each value by one.

Given an array of integers. For any value that is less than 5, set the value to 5.

Given an array of integers. Create a new array of integers that contains no negative values.

# Solve these small problems (cont.)

Given an array of Strings. Create a single new String that is the concatenation of each String in the array.

Do the previous one, in reverse concatenation order.

Write a function to return subsequence of a list that is equivalent to the slice operation in python. The function accepts a list, and two parameters `start` and `end`. An exception of ValueError is thrown on any of the following conditions. 1) the indices are outside the bounds of the list's size, 2) value end is less than start, 3) the list is `None`, 4) the list is empty.

Generate the first $N$ numbers based on the sequence `2, 4, 6, 9, 12, 14, 17, 20, 22, 25, 28, 30, 33, 36, 38, 41, 44, 46, 49, 52...`