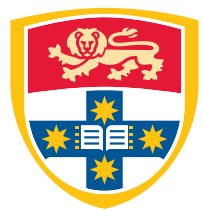
INFO1110 & COMP9001: Introduction to Programming

School of Information Technologies, University of Sydney

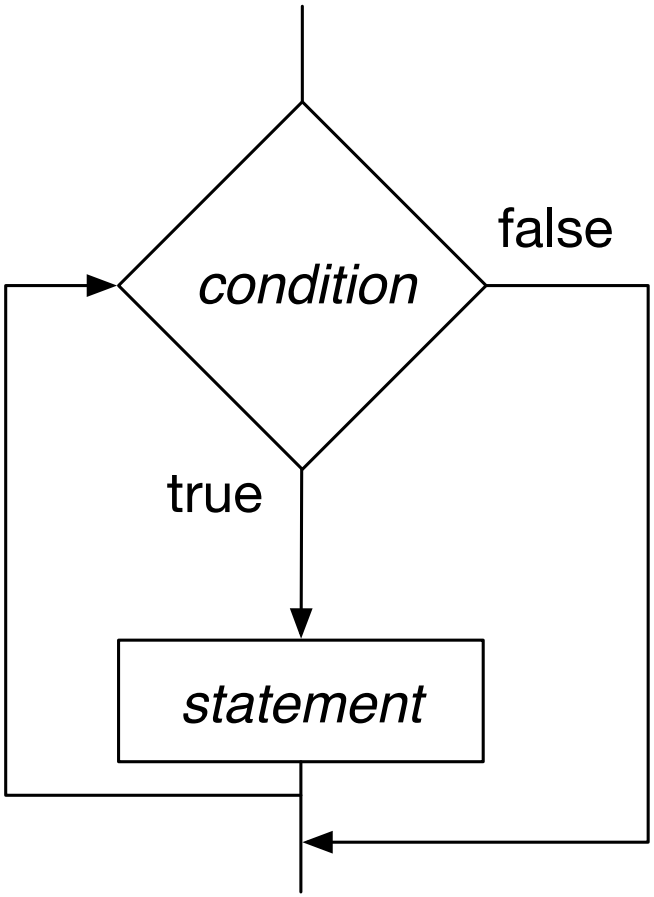


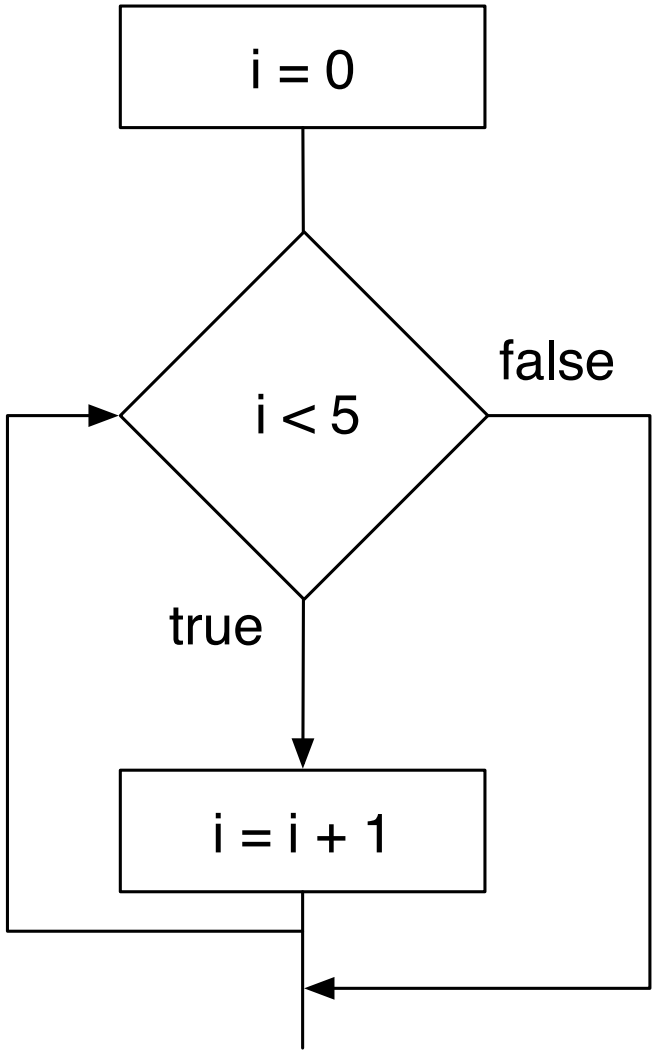
Copyright Warning

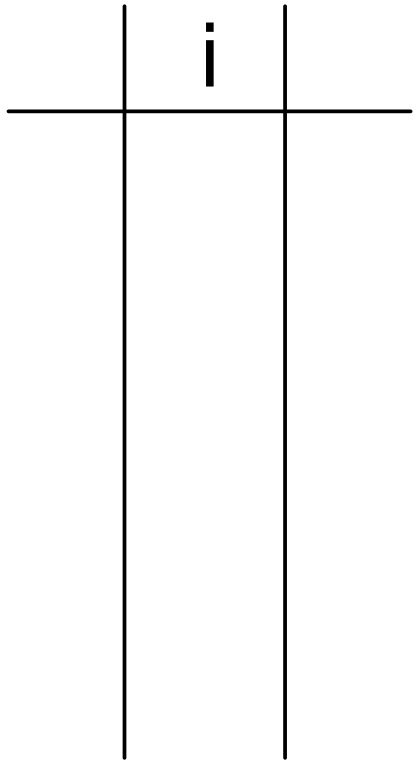
Lecture 8: More on lists and loops

continue; for loop, range, list comprehension

Recall the `while` loop

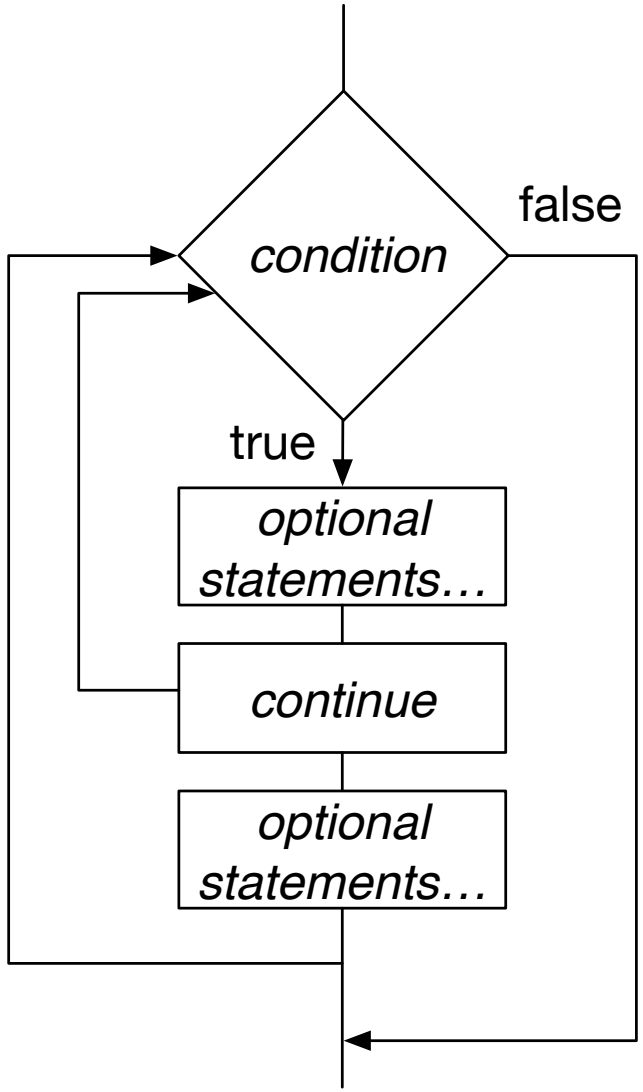




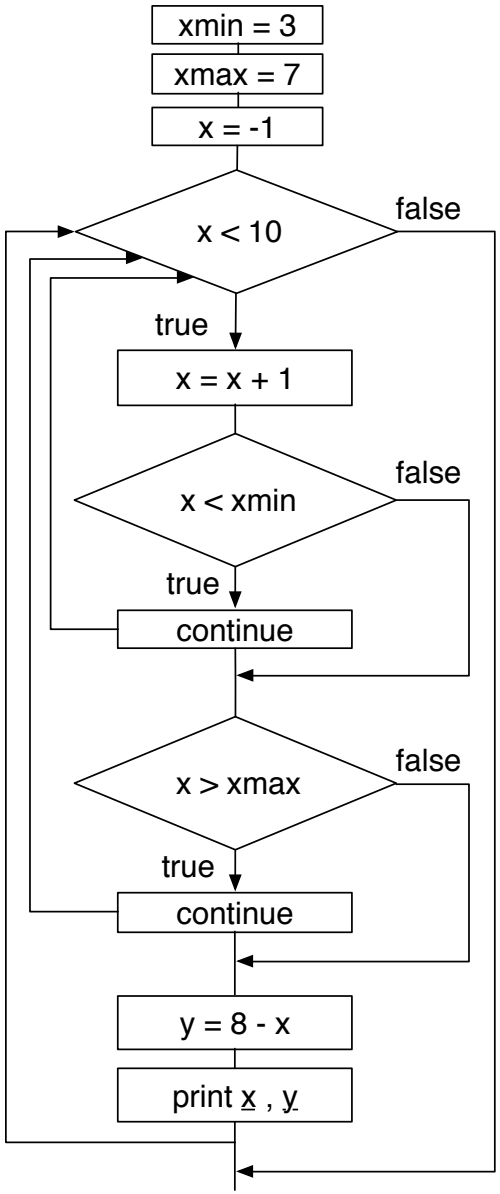


continue

- `continue` is a special reserved word
- When used in loops, it means "skip the rest of this iteration and go on to the next one"
- It is only allowed inside the body of a loop.

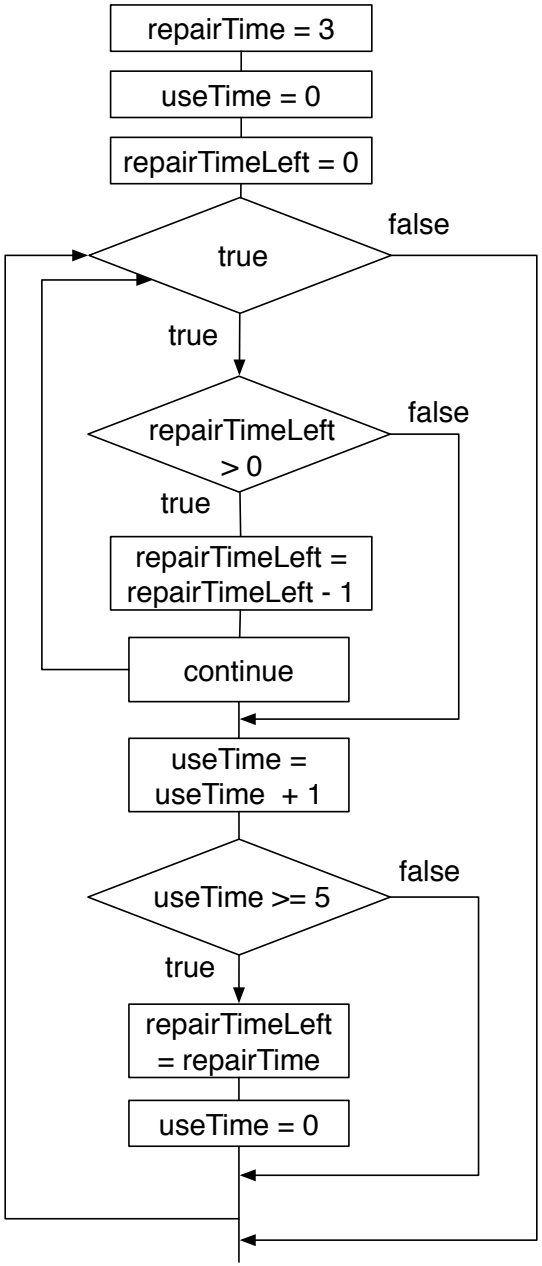


Example with `continue`



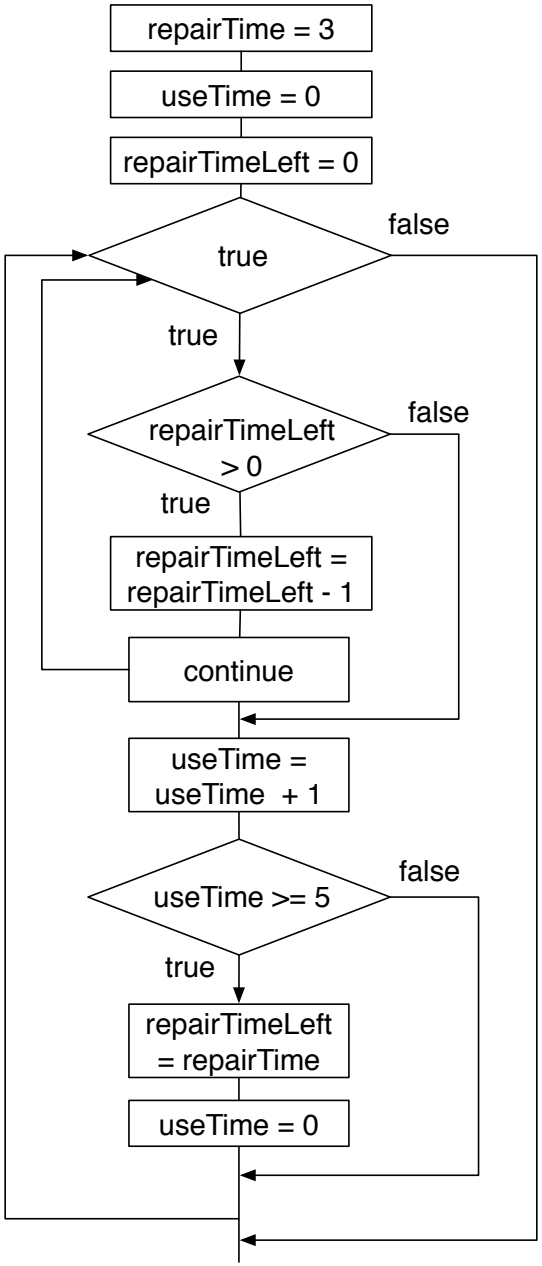
x	y

Example with `continue`



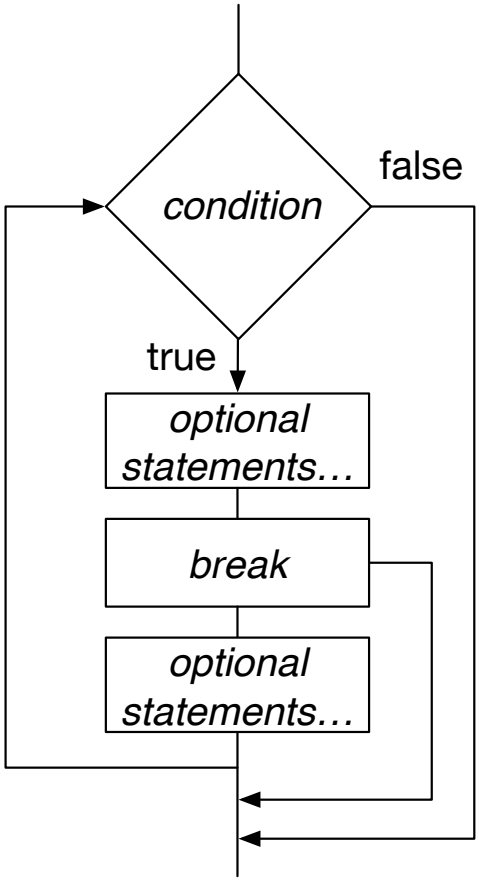
useTime	repairTime	repairTimeLeft

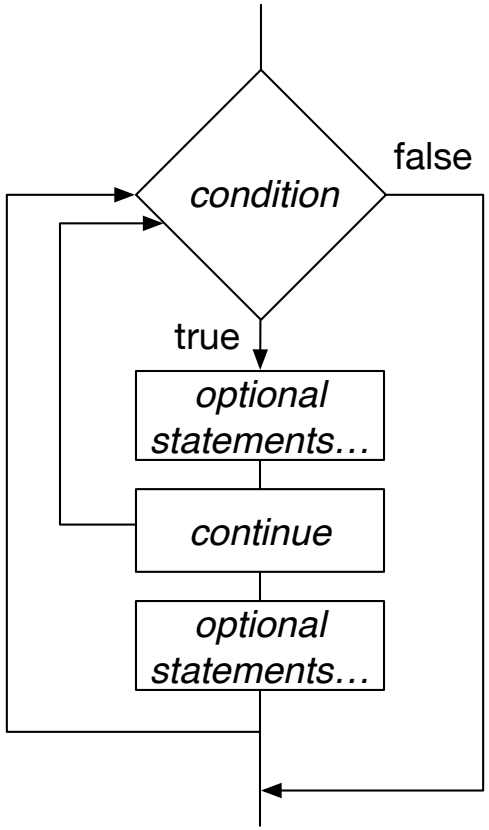
Example with `continue`



useTime	repairTime	repairTimeLeft

`while` loop with `break` and `continue`





Control flow within loops

- if the `break` statement is executed then exit the loop
- if the `continue` statement is executed then operation of the loop goes back to the beginning of the loop

Code trace: `break` and `continue` — example

```
1  x = 4
2  y = 20
3  while x < y:
4      x += 1
5      if x % 2 == 0:
6          continue
7      print("x = " + str(x))
8      if x == 13:
9          break
10 print("All done!")
```

```
1  x = 4
2  y = 20
3  while x < y:
4      x += 1
5      if x % 2 == 0:
6          continue
7      print("x = " + str(x))
8      if x == 13:
9          break
10 print("All done!")
```

```
~> python Continuing.py
x = 5
x = 7
x = 9
x = 11
x = 13
All done!
```

What would it print out if the increment of `x` were moved just after line 5 before line 6?

For loops

A favoured method of iteration

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 # initilise
2 counter = 0
3 while counter < 10:
4     # do something in each iteration
5     ...
6     # move toward the end of loop condition
7     counter = counter + 1;
```

Most loops follow a general structure, can be shortened as a for loop

for

The easy version:

Syntax:

`for` *variable* in *collection* :

    statement

- First, create a variable to initialise with the first item of the collection
- while all items in the collection have not been visited
  - execute the statement
  - assign the variable to the *next* element in the collection

for (cont.)

Example: Denomination of notes in Australia (2019)

For loop

```
1 notes = [ 50, 20, 10, 5 ]
2 for v in notes:
3     print(v)
```

Equivalence with a while loop

```
1 notes = [ 50, 20, 10, 5 ]
2 index = 0
3 while index < len(notes):
4     v = notes[index]
5     print(v)
6     index += 1
```

For loop

This is the most complex loop, so perhaps it's surprising that it's also one of the most commonly seen, though is a reasonably compact way of describing some very sophisticated behaviour.

They perform all the management to setup a variable at the beginning and on each iteration, update the variable for you to use in the loop

range

Common sequences can be constructed using `range`

```
1 for num in range(5):
2     print("the value of num is " + str(num) )
```

```
the value of num is 0
the value of num is 1
the value of num is 2
the value of num is 3
the value of num is 4
```

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

range (cont.)

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

Useful for any counting of iterations to perform.

```
1 for num in range(10, 5, -1):
2     print("the value of num is " + str(num) )
```

```
the value of num is 10
the value of num is 9
the value of num is 8
the value of num is 7
the value of num is 6
```

enumerate

enumerate will return us a new collection where the number sequence is associated with each element.
Recall `zip`. If we have a sequence of the indices parallel to the collection

```
1 indices = [ 0, 1, 2, 3 ]
2 notes = [ 50, 20, 10, 5 ]
3 new_collection = zip(indices, notes)
4 print( list(new_collection) )
```

```
[(0, 50), (1, 20), (2, 10), (3, 5)]
```

`enumerate` is producing the same as the above result.

```
1 new_collection = enumerate(notes)
2 print( list(new_collection) )
```

```
[(0, 50), (1, 20), (2, 10), (3, 5)]
```

Thus enumerate is working to build the sequence of indices automatically for you and then zipping them with the collection values.

`for` can unpack more than one variable

`for` is really:

Syntax:

`for` *target list* in *expression list* :

    statement

I need the index!

```
1 notes = [ 50, 20, 10, 5 ]
2 index = 0
3 while index < len(notes):
4     v = notes[index]
5     print("Note {} is located at position {}".format(v, index) )
6     index += 1
```

```
1 notes = [ 50, 20, 10, 5 ]
2 for (index, v) in enumerate(notes):
3     print("Note {} is located at position {}".format(v, index) )
```

`for` can unpack more than one variable (cont.)

```
coords = [  (1, 2, 3) , (4, 5, 6) ]
for (x, y, z) in coords:
    print(x)
    print(y)
    print(z)
```

Know your collections and their structure

```
records = [ ("Tom", 23, "Cheerful"), \
            ("Sandy", 27, "Happy"), \
            ("Alejandro", "Psyched!") ]
for (name, age, mood) in records:
    print("{} is {} years old and is {}".format(name, age, mood))
```

```
ValueError: need more than 2 values to unpack
```

Compact the `for` loop even more?

It may be necessary to transform a sequence from input to output.

Example: multiply all numbers by 10

```
1 numbers_input = [ 5, 3, 7, 3, 6, 8 ]
2 numbers_output = [0] * len(numbers_input)
3 index = 0
4 for num_in in numbers_input:
5     numbers_output[index] = num_in * 10
6     index += 1
```

Compact the `for` loop even more? (cont.)

Example: for every string with spaces, only include the first word

```
strings_input = [ "Researchers at Tufts University",
                  "are testing tooth-mounted RFID chips",
                  "which sense and transmit data",
                  "on what goes in your mouth." ]
strings_output = [""] * len(strings_input)
index = 0
for str_in in strings_input:
    if " " in str_in:
        strings_output[index] = str_in.split(" ")[0]
    else:
        strings_output[index] = str_in
    index += 1
```

There are many ways. Some messier than others

Compact the `for` loop even more? (cont.)

List comprehension - construct a list of transformed items from an existing list

```
1 numbers_input = [ 5, 3, 7, 3, 6, 8 ]
2 numbers_output = [ num * 10 for num in numbers_input ]
```

```
1 numbers_input = [ 5, 3, 7, 3, 6, 8 ]
2 print( [ num for num in numbers_input if num > 4 ] )
```

```
[5, 7, 6, 8]
```

```
1 numbers_input = [ 5, 3, 7, 3, 6, 8 ]
2 print( [ num if num > 4 else 0 for num in numbers_input ] )
```

```
[5, 0, 7, 0, 6, 8]
```

Compact the `for` loop even more? (cont.)

List comprehensions are super compact!

```
1 strings_input = [ "Researchers at Tufts University",
2                   "are testing tooth-mounted RFID chips",
3                   "which sense and transmit data",
4                   "on what goes in your mouth." ]
5 strings_output = [ str_in.split(" ")[0] \
6    if " " in str_in else str_in for str_in in strings_input ]
```

These are not to be abused. Readable and understandable code is a priority.

`for` vs `while` Which do I choose?

How many iterations do you need? if you know the answer to this question in advance, a for loop is suitable

When does the loop end in these examples?

- A list of names to search through
- the loop for a web server
- the loop for a video game
- the loop for a movie playing
- get a customers order
- sort a collection of books by name