IT@JCU

CP1401 Module 4
Repetition Structures

JAMES COOK UNIVERSITY
AUSTRALIA

Learning outcomes – you will be able to:

- Choose appropriate repetition structures: definite vs. indefinite
- Write Python code to implement programs using repetition structures: while and for
- Test programs that use repetition with appropriate test values
- Learn to solve problems (decomposition, algorithms) using repetitions

Do this now

- Write an algorithm for a program that asks a game's user to choose their level: 1-6. If they choose outside that, display an error message, otherwise display something like, "Level 5" (whatever level they chose).
- Write appropriate **test data** that effectively tests the possible outcomes

- Remember to start by choosing which decision *pattern* to use
- Would it make a difference if there were 2 levels? 60 levels?

One option: if-elif-else pattern

```
get level
if level < 1
    print invalid level
else if level > 6
    print invalid level
else if level = 1
    print level 1
else if level = 2
    print level 2
…
```

Don't repeat yourself (DRY)

Better option: if-else pattern

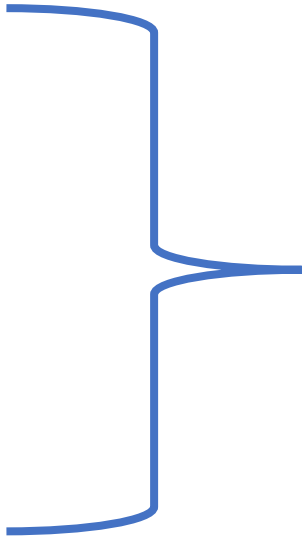
- There are two mutually exclusive outcomes: it's either valid or not

```
get level
if level < 1 or level > 6
    display invalid level
else
    display level
```

- What test data do we need to use?
- What if we want to *keep* asking *until* we get a valid level?...

Get 5 numbers and display their total and average

```
total = 0
get number1
get number2
get number3
get number4
get number5
total = number1 + number2 + number3 + number4 + number5
average = total / 5
display total and average
```



This should make alarm bells ring!
What if there were 5000 numbers?!

Any time you find yourself repeating code, you need to ask yourself if there's a better way...

Algorithms and code should be generalisable:
solve a general class of problem, not a specific one

Without repetition structures...

- We very often need to write code that performs the same kind of task multiple times
- There are disadvantages to duplicating code
  - Makes program large
  - Can be time consuming
  - We may need to make changes in many places
- We sometimes need to do things an indefinite (unknown) number of times
  - This can not be done with sequence only

There are two main kinds of repetition structure

- **Indefinite** iteration – repeat an unknown number of times
  - `while`
- **Definite** iteration – the number of times is known
  - `for`

There are two main kinds of repetition structure

- **`while`** is useful for continuing *while* some condition is True.
  - or *until* some condition is False
    (opposite way of thinking about the same thing)

- **`for`** is useful for iterating through all the elements of a sequence, one at a time.
  - E.g., each number in a list, each line in a file, each character in a string...
  - This includes doing something n times

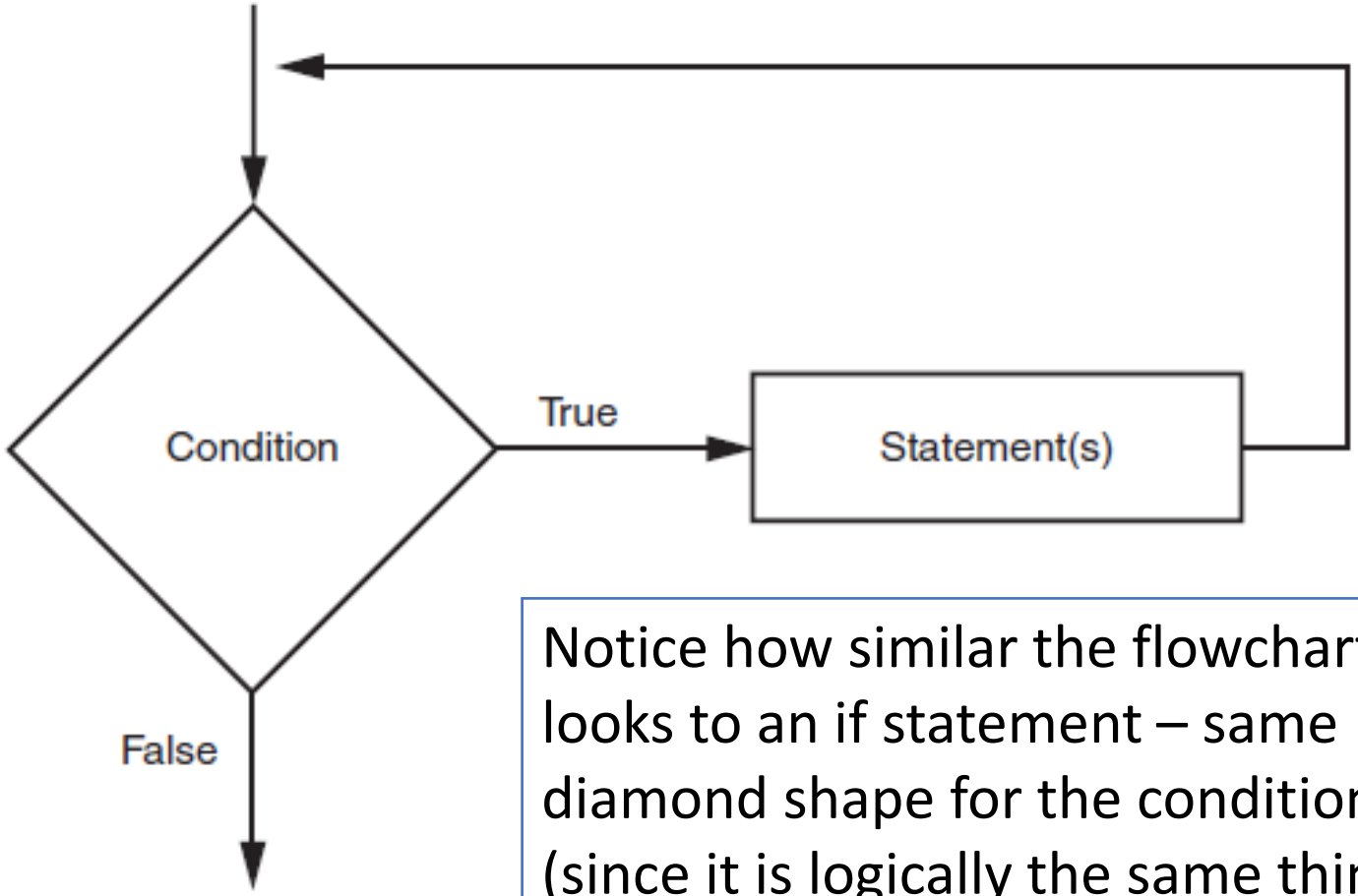
IT@JCU

Use while loops for indefinite repetition

JAMES COOK UNIVERSITY AUSTRALIA

The while loop is similar to an if statement

Figure 5-1 The logic of a `while` loop



Notice how similar the flowchart looks to an if statement – same diamond shape for the condition (since it is logically the same thing).

The while loop is controlled by a condition

- What do we know about conditions?
- For a while loop to *start* executing, the condition must be True
  - Known as a **pretest** loop - the condition is evaluated *before* performing an iteration (body)
  - Will never start if the condition is False to begin with
  - Requires performing some step/s prior to the loop
- For a while loop to *stop* executing, something must happen *inside* the loop to make the condition False

Here's a simple fun while loop example

```
response = input("Do you like me? ").lower()
while response != "yes":
    print("Wrong answer.")
    response = input("Do you like me? ").lower()
print("I'm glad you like me ;)")
```

Deconstructing the while loop in Python

```
while condition:
    statement
    statement
```

- First line includes the keyword `while` followed by a condition, followed by a colon
  - The condition can (only) be true or false
  - First the condition is tested, and if it is True, the indented block statements (body) are executed. Otherwise, body is skipped.

Tracing how while loops execute

```
statements (before)
while condition:
    statements (body)
statements (after)
```

- Program gets to the while and evaluates the condition
- If the condition is True, then all of the statements in the body are executed
- When all the statements in the loop body are executed, control flows back to the start of the loop and the condition is checked again
- If the condition is (still) True, all of the statements in the body are executed (again)
- …
- When the condition is False, control exits the loop (skips the body) and goes to the next section of code (after)

Do this now (play it with someone first!)

Write a program that asks the user to guess a secret number between 1 and 10 and keeps asking until they guess the secret.

- Use a CONSTANT for the secret number
- Just use "? " as the prompt for now

Here's our guessing game while loop

```
SECRET = 6
guess = int(input("? "))
while guess != SECRET:
    print("Guess again!")
    guess = int(input("? "))
print("You got it!")
```

← "priming read"

← meaningful (readable) condition

← same as priming read

- **This is THE standard while loop format we want you to learn.**
- Note that the line before the loop header (condition) is the same as the last line of the body

We've just learned another important pattern

- This while loop pattern is very (very) common
- Learn this and the other common patterns,
  then when you need to do something, first check if you already know a pattern that you can reuse
  - Don't reinvent the wheel or do weird things

- We've written up a lot of these common patterns:
  https://github.com/CP1404/Starter/wiki/Programming-Patterns

while True?

Which one of these is easier to read (and write)?

ANTI-PATTERN!

```
while True:
    guess = int(input("? "))
    if guess == SECRET:
        break
    else:
        print("Guess again!")
print("You got it!")
```

```
guess = int(input("? "))
while guess != SECRET:
    print("Guess again!")
    guess = int(input("? "))
print("You got it!")
```

If you have to write an if statement to break out of a loop, that should probably just be your normal loop condition

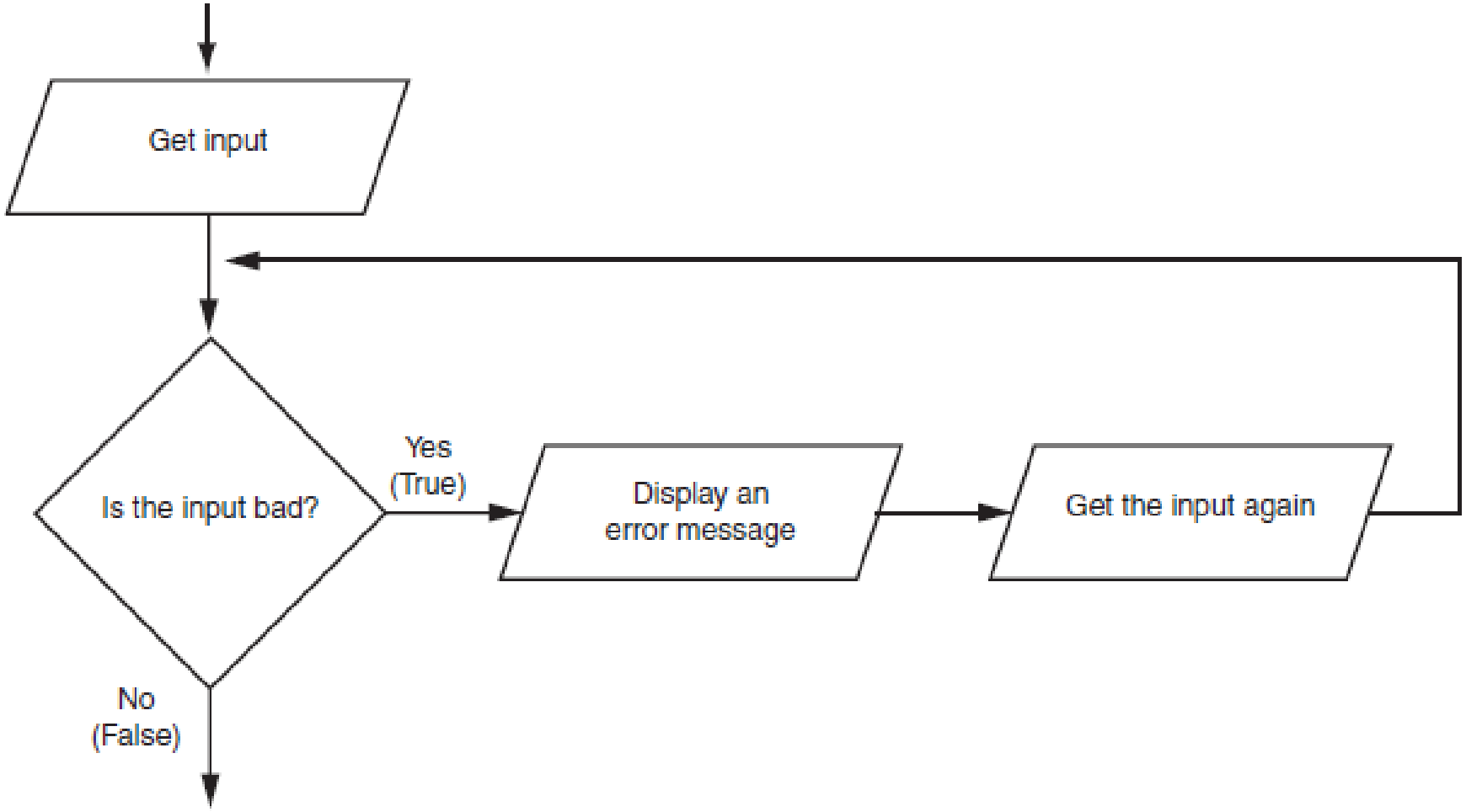
- The 'overhead' of re-writing the priming read is **not** worth the loss of readability and the potential issues of while True and break – especially for larger code (break might be hard to find)

Input validation loops follow the common pattern

- Input validation: inspecting input before it is processed by the program
- If input is invalid, prompt user to enter correct input again
- Commonly accomplished using a while loop, which repeats for as long as the input is bad
  - If input is bad, display error message and get input again (loop)
  - If input is good, skip the body of the loop and continue on (use the input)

Input validation loops

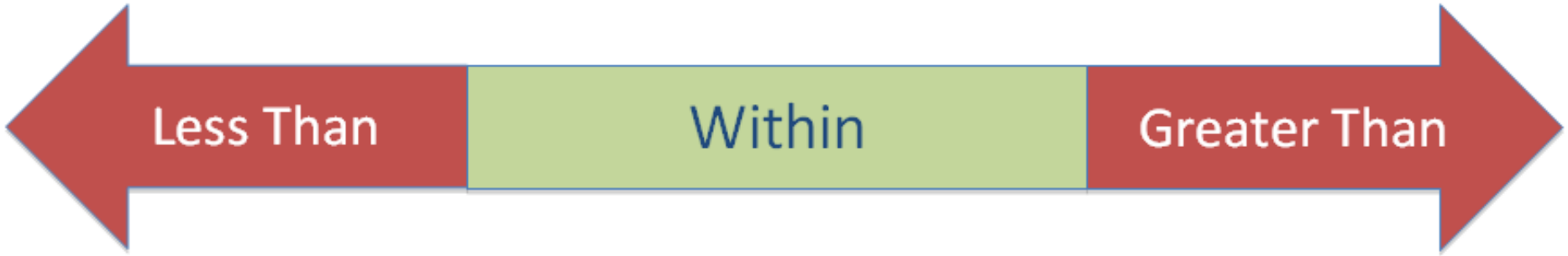
Figure 5-8 Logic containing an input validation loop



The same conditions for **if** apply for **while** (Remember this?)

- To determine if a value is **within** a range, use the **and** operator and appropriate relational operators
  - E.g., `(age >= 0) and (age <= 120)`

- To determine if a value is **outside** a range, use the **or** operator and *opposite* relational operators
  - E.g., `(age < 0) or (age > 120)`



Back to our game level input... with validation loop

```
level = int(input("Level: "))
while level < 1 or level > 6:
    print("Invalid level")
    level = int(input("Level: "))
print("Level", level)
```

"keep going until they get it right"
(same as)
"keep going while they get it wrong"

```
get level
if level < 1 or level > 6
    print invalid level
else
    print level
```

Notice:

- this follows the standard while loop pattern
- it uses the same kind of condition we use with "if"
- the 'good' result/path happens *after* the loop - no else!

while loops (standard pattern) are good for menus

```
display menu
get choice
while choice != quit option
    if choice == first option
        do first task
    else if choice == second option
        do second task
    ...
    else
        display invalid input error message
    display menu
    get choice
do final thing, if needed
```

Sometimes you stop while loops with sentinels

- A sentinel is a special value that marks loop's end condition
- When the condition matches a sentinel, the loop terminates
- The sentinel value must not be an allowed regular value
- Example: keep getting incomes until a negative value is entered; keep doing a menu until the quit value is entered

- What would be a good sentinel value when asking for age values?

Do this now

- Write a program to ask the user for their age; continuing until they enter a valid age (between 0 and 120 inclusive). Then tell them if they are an adult or a minor.

IT@JCU

Use for loops for definite repetition

JAMES COOK UNIVERSITY AUSTRALIA

Here are some sample for loop examples

```
for subject in ["CP1401", "CP1404", "CP2406"]:
    print("I like", subject)
```

```
I like CP1401
I like CP1404
I like CP2406
```

```
for number in [2, 4, -8, 99]:
    print(number, end=" ")
```

```
2 4 -8 99
```

```
for character in "Python":
    print(character, end="-")
```

```
P-y-t-h-o-n-
```

The first two use lists, which we will learn later in the subject…

Deconstructing the for loop in Python

```
for item in sequence:
    statement
    statement
```

- We are getting used to the colon and the indent now…
- A sequence can be anything that is *iterable*:
  - list, string, function that returns an iterable sequence…
- The "item" is each element of the sequence, which gets stored in the "target variable" for each loop through the body

for loops commonly iterate through a **range**

- The range function represents a sequence of integers

```
range(start, end, step)
```

  - the **start** of the range. Assumed to be 0 if not provided (default value).
  - the **end** of the range, but *not inclusive* (up to but not including the number). Required.
  - the **step** of the range. Assumed to be 1 if not provided.
- If only one argument is given, defaults are used (start=0, step=1)

```
range(5) == range(0, 5) == range(0, 5, 1)
```

for loops commonly iterate through a **range**

```
for i in range(1, 5):
    print(i)
```

- This range is the sequence 1, 2, 3, 4
- The loop assigns each of the values in the sequence, one at a time, to the variable `i`
  - `i` is a common name for an index, integer, counter, number in sequence
  - Do NOT use `i` for anything else (e.g., a name, income, age, distance…)
- To do something n times, just use (default start & step):

```
for i in range(n): ...
```

Tracing how for loops execute

```
statements (before)
for item in sequence:
    statements (body)
statements (after)
```

- Program gets to the for and puts the next item in the sequence into the target variable
- All of the statements in the body (usually something to do with item) are executed
- When all the statements in the loop body are executed, control flows back to the start of the loop and the next item is unpacked
- All the statements in the body are executed (again)
- …
- When there are no more items, control exits the loop and goes to the next section of code (after)

Do this now

- Write a program to ask the user for their name and age, then print their name the same number of times as their age.
  Example:

```
Name: Lindsay
Age: 3
Lindsay
Lindsay
Lindsay
```

IT@JCU

Accumulation Nested Loops Avoiding Problems

JAMES COOK UNIVERSITY AUSTRALIA

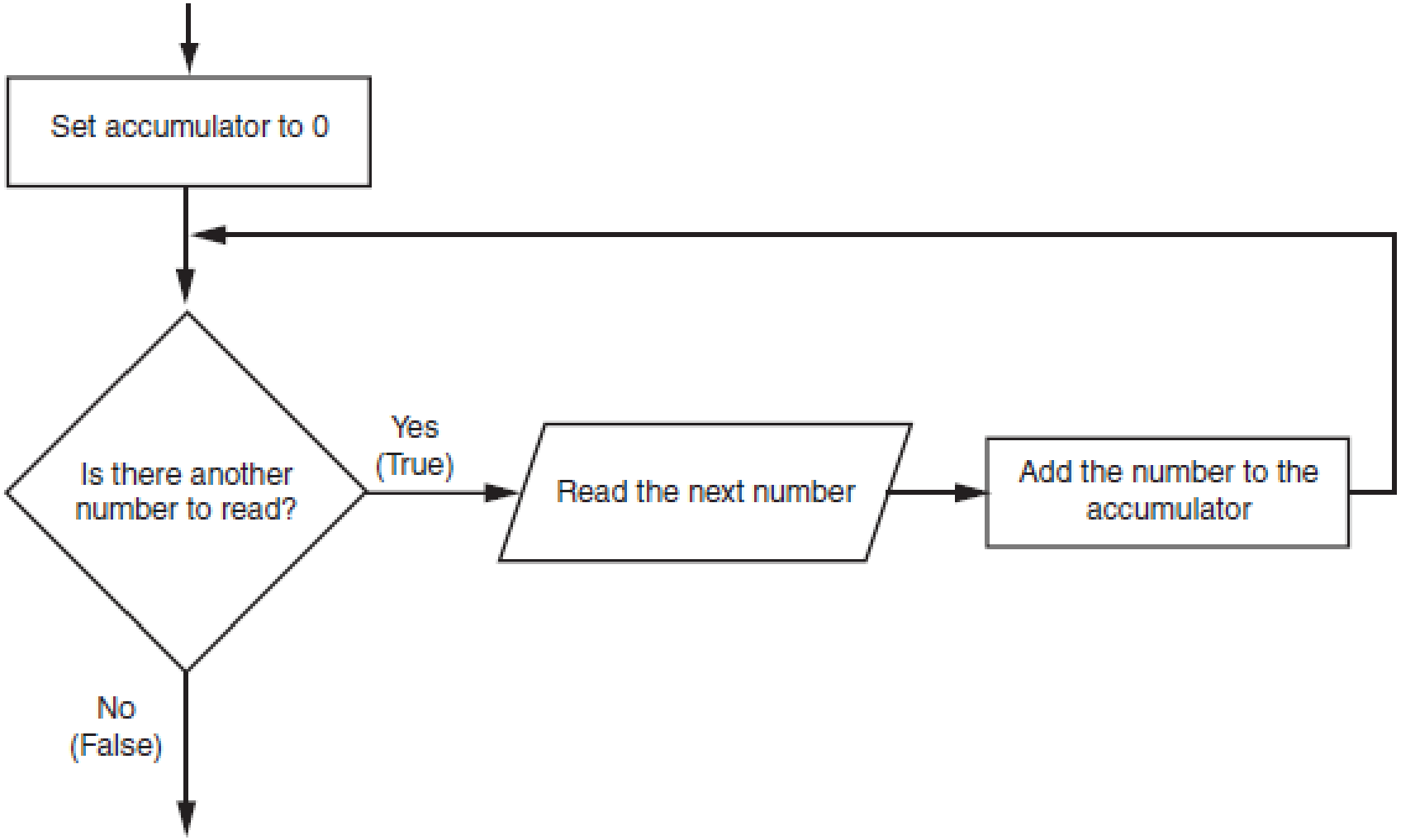
Learn the accumulation pattern

- Programs often need to calculate a total within a loop (like a series of numbers)
  - Typically include two elements:
    - An accumulator (total) variable that is set to a starting value
    - A loop that reads each number in series (could come from user input, file, network…)
  - At the end of the loop, accumulator will reference the total
- The following code adds up the numbers from 1 to 10:

```
total = 0
for number in range(1, 11):
    total = total + number
print(total)
```

Learn the accumulation pattern

Figure 5-7 Logic for calculating a running total



Augmented assignment operators are nice

- In many assignment statements, the variable on the left of the = operator also appears on the right:

```
total = total + number
```

- Augmented assignment uses shorthand operators:

```
total += number
```

Augmented Assignment Version	Equivalent To
`x += y`	`x = x + y`
`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x %= y`	`x = x % y`

Do this now

Write a program to:

- Ask the user how many ages to enter (e.g., we know there are n people in the room), then ask for that many ages and print the total and average at the end.

Write a second program to:

- Repeatedly ask for an age (unknown/indefinite number of ages), stopping when the user enters -1, then print the total and average of the ages.

Nested loops are a common structure

- Loops can be contained inside other loops
- This is common for things like coordinates (x, y)
- j is a common inner loop counter, then k inside that, so: i, j, k
- The inner loop goes through all its iterations for each iteration of the outer loop

```
for each person in the class
    for each of the person's assignments
        mark assignment

while customer in checkout
    for each item
        scan item
```

Nested loops are a common structure

What is the output of the following code?

```
for i in range(3):
    for j in range(2):
        print(i, "-", j)
```

```
0 - 0
0 - 1
1 - 0
1 - 1
2 - 0
2 - 1
```

Nested loops

- How would you write a program to print whose turn and what round it is for a game?
  Say we want 3 rounds, and we have 3 names…

```
Round 1 - Miles
Round 1 - Ella
Round 1 - Chet
Round 2 - Miles
Round 2 - Ella
Round 2 - Chet
Round 3 - Miles
Round 3 - Ella
Round 3 - Chet
```

- What's the output of the following code?

```
for round_number in range(1, 4):
    for name in ["Miles", "Ella", "Chet"]:
        print("Round", round_number, "-", name)
```

Nested loops

- How would we print a line or something after each round (not after each person?)

- Note the location: *outside (after)* the inner loop, but *inside* the outer loop

```
for round_number in range(1, 4):
    for name in ["Miles", "Ella", "Chet"]:
        print("Round", round_number, "-", name)
    print("--------------------")
```

```
Round 1 - Miles
Round 1 - Ella
Round 1 - Chet
---------------
Round 2 - Miles
Round 2 - Ella
Round 2 - Chet
---------------
Round 3 - Miles
Round 3 - Ella
Round 3 - Chet
---------------
```

Learn to avoid common loop problems

- Loop never starts
  - while loop condition is always False or for loop sequence is empty

```
while level < 1 and level > 6:  # Such as?
```

- Off-by-one error
  - Just like with decisions, always double-check your boundary conditions

```
for number in range(1, 10):  # This is 1-9 not 1-10
```

- Loop never ends
  - infinite loop, infinite loop, infinite loop, infinite loop, infinite loop, infinite loop, infinite loop, infinite loop, infinite loop, infinite loop, infinite loop, …

You (usually) need to avoid infinite loops

- Loops must contain a way to terminate
  - Something inside a while loop must eventually make the condition False
- If a loop does not have a way of stopping, it repeats forever – until program is interrupted/cancelled by the user
- Usually occurs because the programmer:
  - Forgot to include stopping code inside the loop
  - Got the condition wrong

```
level = int(input("Level: "))
while level < 1 or level > 6:
    print("Invalid level")
print("Level", level)
```

Test your loops systematically

- Just like with decision structures, we need to write good tests for repetition structures
- With decisions we should test:
  - Each path: if, elif, else…
  - Each boundary condition
- Guess what?...
- With repetitions we should test:
  - Each path: true (loop), false (no loop)
  - Each boundary condition
- What data would you test our game level program with?

Now that we know the 'tools', how can we plan to use them?

JAMES COOK UNIVERSITY AUSTRALIA

Look for repetition during problem decomposition

- What words in a problem description indicate repetition?
  - Look for words like **repeat**, **until**, **keep going**, **continue**, **n times**…

- Get a user's income, then calculate and display their tax repeatedly until they enter a negative income.

Use decomposition to look for repetition in this:

Ask a student to enter the number of subjects they take. Then repeatedly ask for the name of each subject. If one of their subjects is "CP1401", congratulate them for their fine choice.

Ask a student to enter the number of subjects they take. Then ask for the name of each subject. If one of their subjects is "CP1401", congratulate them for their fine choice.

Choose the right kind of loop

- When choosing a loop, we can ask, “How many times will it run?”
- if we know the answer - use a **definite** – for – loop
  - E.g., ask the user for a number repeatedly until they have entered five numbers.

- if we don’t (or can’t know) - use an **indefinite** – while – loop
  - E.g., get numbers from the user until they enter zero or less

Don't force the wrong choice of loop to work

- It is *possible* to use a while loop to do definite iteration, but it would not the best choice of loop… that's what for loops are for.
  - E.g., You could use a while loop to iterate through numbers 1 to 10 by having a counter starting at 1 that you +1 each time through the loop… but for loops with range already do this (more easily).
- Using a for loop and maintaining your own counter (ignoring the target variable) is also poor, since for loops already do this.

- These would be considered *anti-patterns*

Don't force the wrong choice of loop to work

Pseudocode for loops should sound natural

```
while condition
    statement

repeat 5 times
    statement

for number 1-5
    statement (usually using number)

for each item in sequence
    statement (usually using item)
```

Do this now – write pseudocode

- Ask the user for a number repeatedly until they have entered five numbers, then calculate the total and average of the five numbers.

```
total = 0
repeat 5 times
      get number
      total = total + number
average = total / 5
display total and average
```

Notice we clearly specified a loop but didn't need to use "for" or "range".
We don't use +=
Because these things are language-specific.

Which kind of loop (for=definite, while=indefinite)?

1. Printing the even numbers from 16 to 100
2. Getting names until a blank one is entered
3. Reading every line in a text file
4. Counting how many vowels there are in a sentence
5. Practising writing loops until you understand them well

Now do these next steps

- Find an everyday process that uses repetition and rewrite it as an algorithm in pseudocode
- Practise writing algorithms and programs that use repetition structures
  - Remember to think about which type of loop to choose first