3. Control Statements and Program Development

Objectives ¶

- Decide whether to execute actions with the statements `if` , `if` … `else` and if…elif…else.
- Execute statements repeatedly with `while` and `for` .
- Shorten assignment expressions with augmented assignments.
- Use the `for` statement and the built-in `range` function to repeat actions for a sequence of values.
- Perform sentinel-controlled iteration with `while` .

Objectives (cont.)

- Learn problem-solving skills: understanding problem requirements, dividing problems into smaller pieces, developing algorithms to solve problems and implementing those algorithms in code.
- Develop algorithms through the process of top-down, stepwise refinement.
- Create compound conditions with the Boolean operators `and` , `or` and `not` .

Objectives (cont.)

- Stop looping with `break` .
- Force the next iteration of a loop with `continue` .
- Use some functional-style programming features to write scripts that are more concise, clearer, easier to debug and easier to parallelize.

Outline

Outline

Outline

This content is based on Chapter 3 of the book **Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and the Cloud** (https://amzn.to/2VvdnxE).

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

3.5 `if` Statement

- Pseudocode: Suppose that a passing grade on an examination is 60. The pseudocode

```
If student’s grade is greater than or equal to 60
    Display 'Passed'
```

- If the condition is true, 'Passed' is displayed. Then, the next pseudocode statement in order is “performed.”
- If the condition is false, nothing is displayed, and the next pseudocode statement is “performed.”
- Indentation emphasizes that 'Passed' is displayed only if the condition is true.

Corresponding `if` Statement

In [1]:

```
grade = 85
```

In [2]:

```
if grade >= 60:
    print('Passed')
```

```
Passed
```

Suite Indentation

- Indenting a suite is required.

In [3]:

```
if grade >= 60:
print('Passed')  # statement is not indented properly
```

```
  File "<ipython-input-3-eb5359d7857b>", line 2
    print('Passed')  # statement is not indented properly
        ^
IndentationError: expected an indented block
```

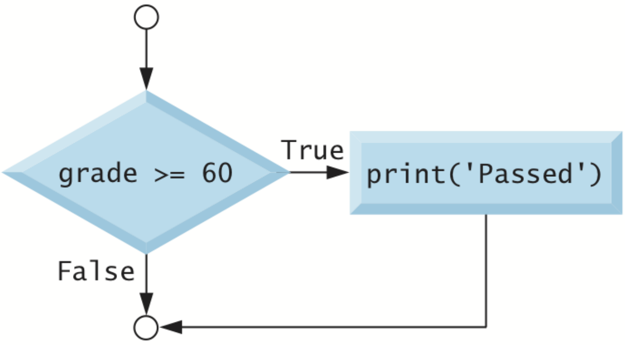
- Statements in a suite must have the same indentation.

In [5]:

```
if grade >= 60:
    print('Passed')
  print('Good job!)
```

```
  File "<tokenize>", line 3
    print('Good job!)
    ^
IndentationError: unindent does not match any outer indentation leve
l
```

`if` Statement Flowchart



- The decision (diamond) symbol contains a condition that can be either `True` or `False`.
- Two flowlines emerging from it:
  - One indicates the direction to follow when the condition in the symbol is `True`.
  - The other indicates the direction to follow when the condition is `False`.

Every Expression Can Be Treated as `True` or `False`

In [6]:

```
if 1:
    print('Nonzero values are true, so this will print')
```

```
Nonzero values are true, so this will print
```

In [7]:

```
if 0:
    print('Zero is false, so this will not print')
```

An Additional Note on Confusing `==` and `=`

- Using `==` instead of `=` in an assignment statement can lead to subtle problems.
- Writing `grade == 85` when we intend to define a variable with `grade = 85` would cause a `NameError`.
- Logic error: If `grade` had been defined **before** the preceding statement, then `grade == 85` would evaluate to `True` or `False`, depending on `grade`'s value, and not perform the intended assignment.

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

3.6 `if…else` and `if…elif…else` Statements

- Performs different suites, based on whether a condition is `True` or `False`.
- Pseudocode:

```
If student’s grade is greater than or equal to 60
    Display 'Passed'
Else
    Display 'Failed'
```

- Correspondong Python code with variable `grade` initalized to `85`

In [1]:

```
grade = 85
```

In [2]:

```
if grade >= 60:
    print('Passed')
else:
    print('Failed')
```

```
Passed
```

- Assign `57` to `grade`, then shows the `if`…`else` statement again to demonstrate that only the `else` suite executes

In [3]:

```
grade = 57
```

In [4]:

```
if grade >= 60:
    print('Passed')
else:
    print('Failed')
```

```
Failed
```

In IPython:

- The up and down arrow keys navigate backwards and forwards through the current interactive session’s snippets.
- Pressing *Enter* re-executes the snippet that’s displayed.
- In JupyterLab, you can select a cell in its left margin, press *C* to copy it and *V* to paste it below the currently selected cell.

In [5]:

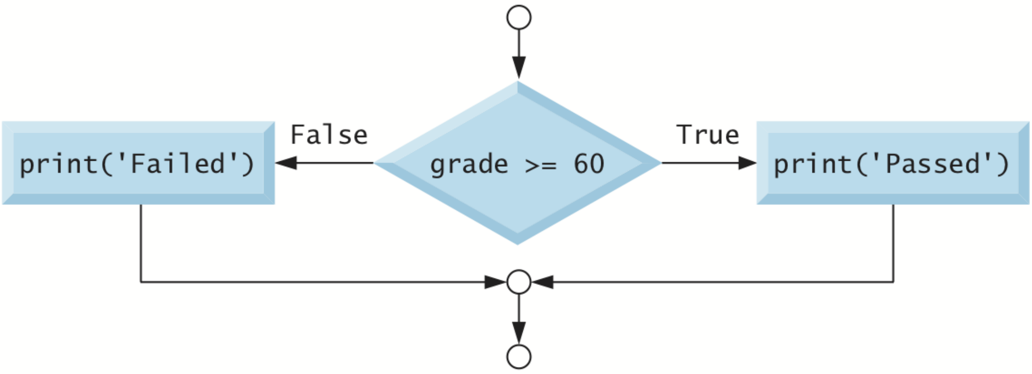
```
grade = 99
```

In [6]:

```
if grade >= 60:
    print('Passed')
else:
    print('Failed')
```

```
Passed
```

`if`…`else` Statement Flowchart



Conditional Expressions

- Sometimes the suites in an `if`…`else` statement assign different values to a variable, based on a condition

In [7]:

```
grade = 87
```

In [8]:

```
if grade >= 60:
    result = 'Passed'
else:
    result = 'Failed'
```

In [9]:

```
result
```

Out[9]:

```
'Passed'
```

- Can write statements like this using a concise conditional expression.
- The parentheses are not required, but they make it clear that the statement assigns the conditional expression's value to `result`.

In [10]:

```
result = ('Passed' if grade >= 60 else 'Failed')
```

In [11]:

```
result
```

Out[11]:

```
'Passed'
```

- In interactive mode, you also can evaluate the conditional expression directly.

In [12]:

```
'Passed' if grade >= 60 else 'Failed'
```

Out[12]:

```
'Passed'
```

Multiple Statements in a Suite

In [13]:

```
grade = 49
```

In [14]:

```
if grade >= 60:
    print('Passed')
else:
    print('Failed')
    print('You must take this course again')
```

```
Failed
You must take this course again
```

- If you do not indent the second `print` , then it’s not in the `else` ’s suite.
- In that case the statement always executes, creating strange incorrect output.

In [15]:

```
grade = 100
```

In [16]:

```
if grade >= 60:
    print('Passed')
else:
    print('Failed')
print('You must take this course again')
```

```
Passed
You must take this course again
```

`if`…`elif`…`else` Statement

- Can test for many cases.
- Only the action for the first `True` condition executes.

In [17]:

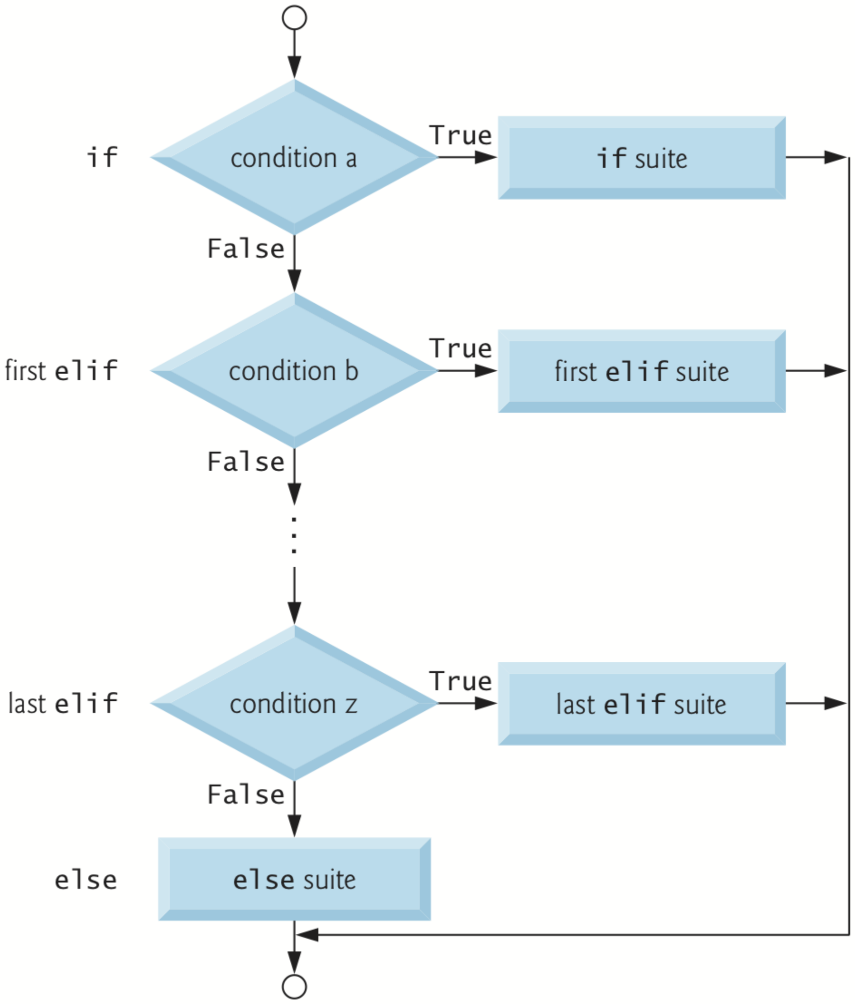
```
grade = 77
```

In [18]:

```
if grade >= 90:
    print('A')
elif grade >= 80:
    print('B')
elif grade >= 70:
    print('C')
elif grade >= 60:
    print('D')
else:
    print('F')
```

```
C
```

`if`…`elif`…`else` Statement Flowchart



`else` Is Optional

- Handle values that do not satisfy any of the conditions.
- Without an `else` , if no conditions are `True` , the program does not execute any of the statement's suites.

Logic Errors

- For a nonfatal logic error, code executes, but produces incorrect results.
- For a fatal logic error in a script, an exception occurs, Python displays a traceback, then the script terminates.
- A fatal error in interactive mode terminates the current snippet, then IPython waits for your next input.

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

3.7 `while` Statement

- Repeats one or more actions while a condition remains `True`.

In [1]:

```
product = 3
```

In [2]:

```
while product <= 50:
    product = product * 3
```

In [3]:

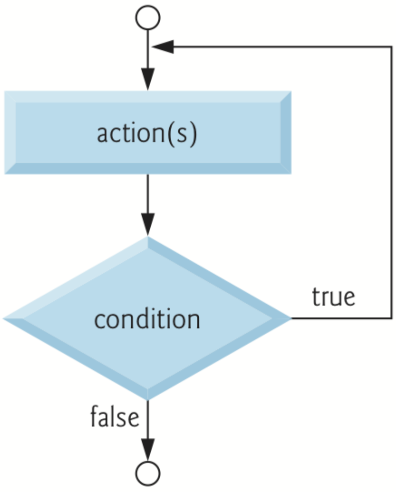
```
product
```

Out[3]:

```
81
```

- To prevent an infinite loop, something in the `while` suite must change `product`'s value, so the condition eventually becomes `False`.

`while` Statement Flowchart



DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

3.8 `for` Statement

- Repeat an action or several actions for each item in a sequence of items.
- A string is a sequence of individual characters.

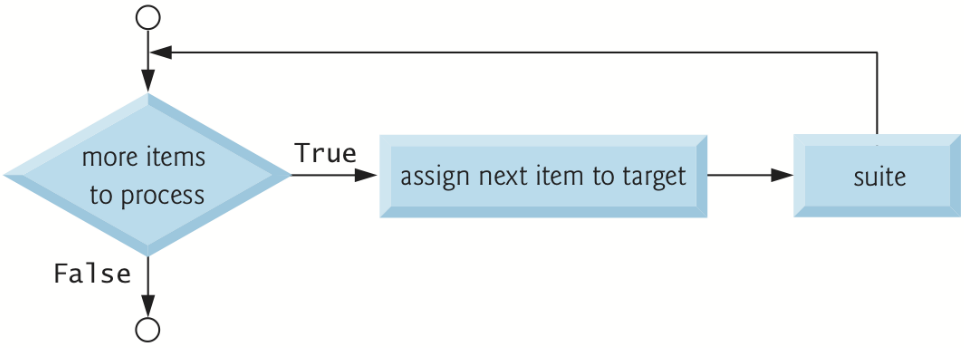
In [1]:

```
for character in 'Programming':
    print(character, end='  ')
```

```
P  r  o  g  r  a  m  m  i  n  g
```

- Upon entering the `for` loop, Python assigns the 'P' in 'Programming' to the **target variable** between keywords `for` and `in`.
- After executing the suite, Python assigns to character the next item in the sequence (that is, the `'r'` in `'Programming'`), then executes the suite again.
- Continues while there are more items in the sequence.
- Using the target variable in the suite is common but not required.

`for` Statement Flowchart



Function `print`'s `end` Keyword Argument

- `print` displays its argument(s), then moves the cursor to the next line.
- Can change this behavior with the argument `end`:

```
print(character, end='  ')
```

- `end` is a **keyword argument**, but it's not a Python keyword.
- The *Style Guide for Python Code* recommends placing no spaces around a keyword argument's =.
- Keyword arguments are sometimes called named arguments.

Function `print` ’s `sep` Keyword Argument

- Keyword argument `sep` (short for separator) specifies the string that appears between the items that print displays.
- A space character by default.
- To remove the spaces, use an empty string with no characters between its quotes.

In [2]:

```
print(10, 20, 30, sep=', ')
```

```
10, 20, 30
```

3.8.1 Iterables, Lists and Iterators

- The sequence to the right of the `for` statement’s in keyword must be an iterable.
  - An object from which the `for` statement can take one item at a time.
- One of the most common iterables is a list, which is a comma-separated collection of items enclosed in square brackets (`[` and `]`).

In [3]:

```
total = 0
```

In [4]:

```
for number in [2, -3, 0, 17, 9]:
    total = total + number
```

In [5]:

```
total
```

Out[5]:

```
25
```

- Each sequence has an iterator.
- The for statement uses the iterator “behind the scenes” to get each consecutive item until there are no more to process.

3.8.2 Built-In `range` Function and Generators

- Creates an iterable object that represents a sequence of consecutive integer values starting from 0 and continuing up to, but not including, the argument value.

In [6]:

```
for counter in range(10):
    print(counter, end=' ')
```

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

Off-By-One Errors

A logic error known as an off-by-one error occurs when you assume that `range`'s argument value is included in the generated sequence.

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

3.10 Program Development: Sequence-Controlled Iteration

- Most challenging part of solving a problem on a computer is developing an algorithm for the solution.
- Once a correct algorithm has been specified, creating a working Python program from the algorithm is typically straightforward.

3.10.1 Requirements Statement

- A **requirements statement** describes what a program is supposed to do, but not how the program should do it.
- Consider the following simple requirements statement:

> A class of ten students took a quiz. Their grades (integers in the range 0 – 100) are 98, 76, 71, 87, 83, 90, 57, 79, 82, 94. Determine the class average on the quiz.

- Once you know the problem's requirements, you can begin creating an algorithm to solve it. Then, you can implement that solution as a program.
- The algorithm for solving this problem must:
  - Keep a running total of the grades.
  - Calculate the average—the total of the grades divided by the number of grades.
  - Display the result.

3.10.2 Pseudocode for the Algorithm

```
Set total to zero
Set grade counter to zero
Set grades to a list of the ten grades

For each grade in the grades list:
    Add the grade to the total
    Add one to the grade counter

Set the class average to the total divided by the number of grades
Display the class average
```

- Note the mentions of **total** and **grade counter**.
- We'll use these in the script to calculate the average.
- Variables for totaling and counting normally are initialized to zero.

3.10.3 Coding the Algorithm in Python

```python
# fig03_01.py
"""Class average program with sequence-controlled iteration."""

# initialization phase
total = 0  # sum of grades
grade_counter = 0
grades = [98, 76, 71, 87, 83, 90, 57, 79, 82, 94]  # list of 10 grades

# processing phase
for grade in grades:
    total += grade  # add current grade to the running total
    grade_counter += 1  # indicate that one more grade was processed

# termination phase
average = total / grade_counter
print(f'Class average is {average}')
```

In [1]:

```
run fig03_01.py
```

```
Class average is 81.7
```

Execution Phases

- Initialization phase creates the variables needed to process the grades and set these variables to appropriate initial values.
- Processing phase processes the grades, calculating the running total and counting the number of grades processed so far.
- Termination phase calculates and displays the class average.
- Many scripts can be decomposed into these three phases.

3.10.4 Introduction to Formatted Strings

- An **f-string** (short for formatted string) allows inserting values into a string.
- The letter f before the string's opening quote indicates it's an f-string.
- You specify where to insert values by using placeholders delimited by curly braces ({ and }).
- `{average}` converts the variable average's value to a string representation, then replaces `{average}` with that *&replacement text**.
- Replacement-text expressions may contain values, variables or other expressions.

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

3.13 Built-In Function `range`: A Deeper Look

- Function `range`'s two-argument version produces a sequence of consecutive integers from its first argument's value up to, but not including, the second argument's value

In [1]:

```
for number in range(5, 10):
    print(number, end=' ')
```

```
5 6 7 8 9
```

- Function `range`'s three-argument version produces a sequence of integers from its first argument's value up to, but not including, the second argument's value, incrementing by the third argument's value (the step)

In [2]:

```
for number in range(0, 10, 2):
    print(number, end=' ')
```

```
0 2 4 6 8
```

- If the third argument is negative, the sequence progresses from the first argument's value down to, but not including the second argument's value, decrementing by the third argument's value

In [3]:

```
for number in range(10, 0, -2):
    print(number, end=' ')
```

```
10 8 6 4 2
```

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

3.15 `break` and `continue` Statements

- Executing a `break` statement in a `while` or `for` immediately exits that statement.

In [1]:

```
for number in range(100):
    if number == 10:
        break
    print(number, end=' ')
```

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

- Executing a `continue` statement in a `while` or `for` loop skips the remainder of the loop's suite.
    - In a `while`, the condition is then tested to determine whether the loop should continue executing.
    - In a `for`, the loop processes the next item in the sequence (if any)

In [2]:

```
for number in range(10):
    if number == 5:
        continue
    print(number, end=' ')
```

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

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

3.16 Boolean Operators `and`, `or` and `not`

Boolean Operator `and`

- Ensure that two conditions are both `True` with the **Boolean `and` operator**.

In [1]:

```
gender = 'Female'
```

In [2]:

```
age = 70
```

In [3]:

```
if gender == 'Female' and age >= 65:
    print('Senior female')
```

```
Senior female
```

- *Truth table* for the `and` operator:

expression1	expression2	expression1 `and` expression2
`False`	`False`	`False`
`False`	`True`	`False`
`True`	`False`	`False`
`True`	`True`	`True`

Boolean Operator `or`

- Ensure that one *or* both of two conditions are `True` with the **Boolean `or` operator**.

In [4]:

```
semester_average = 83
```

In [5]:

```
final_exam = 95
```

In [6]:

```
if semester_average >= 90 or final_exam >= 90:
    print('Student gets an A')
```

```
Student gets an A
```

- *Truth table* for the `or` operator:

expression1	expression2	expression1 or expression2
`False`	`False`	`False`
`False`	`True`	`True`
`True`	`False`	`True`
`True`	`True`	`True`

Improving Performance with Short-Circuit Evaluation

- Python stops evaluating an `and` expression as soon as it knows whether the entire condition is `False`.
- Python stops evaluating an `or` expression as soon as it knows whether the entire condition is `True`.
- In expressions that use `and`, make the condition that's more likely to be `False` the leftmost condition.
- In `or` operator expressions, make the condition that's more likely to be `True` the leftmost condition.

Boolean Operator `not`

- "Reverse" the meaning of a condition.
- **Unary operator**—it has only *one* operand.

In [7]:

```
grade = 87
```

In [8]:

```
if not grade == -1:
    print('The next grade is', grade)
```

```
The next grade is 87
```

In [9]:

```
if grade != -1:
    print('The next grade is', grade)
```

```
The next grade is 87
```

- Truth table for the `not` operator.

expression	not expression
`False`	`True`
`True`	`False`

- Precedence and grouping of the operators introduced so far—shown in decreasing order of precedence.

Operators	Grouping
`()`	left to right
`**`	right to left
`*` `/` `//` `%`	left to right
`+` `-`	left to right
`<` `<=` `>` `>=` `==` `!=`	left to right
`not`	left to right
`and`	left to right
`or`	left to right

This content is based on Chapter 3 of the book **Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and the Cloud** (https://amzn.to/2VvdnxE).

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

3.17 Intro to Data Science: Measures of Central Tendency—Mean, Median and Mode

- **Measures of central tendency**:
    - **mean**—the *average value* in a set of values.
    - **median**—the *middle value* when all the values are arranged in sorted order.
    - **mode**—the *most frequently occurring value*.
- Each represents a “central” value in a set of values.
    - A value which is in some sense typical of the others.

In [1]:

```
grades = [85, 93, 45, 89, 85]
```

In [2]:

```
sum(grades) / len(grades)
```

Out[2]:

```
79.4
```

- `sum` and `len` are both examples of functional-style programming reductions
- The Python Standard Library’s **`statistics`** module provides functions for calculating the **reductions** mean, median and mode.

In [3]:

```
import statistics
```

In [4]:

```
statistics.mean(grades)
```

Out[4]:

```
79.4
```

In [5]:

```
statistics.median(grades)
```

Out[5]:

```
85
```

In [6]:

```
statistics.mode(grades)
```

Out[6]:

```
85
```

- Sorting `grades` helps you see the median and mode.

In [7]:

```
sorted(grades)
```

Out[7]:

```
[45, 85, 85, 89, 93]
```

DISCLAIMER: The authors and publisher of this book have used their best efforts in preparing the book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in these books. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.