CH-3: Control Statements and Program Development

September 2024

Algorithms

- You can solve any computing problem by executing a series of actions in a specific order.
- An algorithm is a procedure for solving a problem in terms of:
  1. the actions to execute, and
  2. the order in which these actions execute.
- Correctly specifying the order in which the actions execute is essential.
- Program control specifies the order in which statements (actions) execute in a program.
- We'll investigate program control using Python's control statements.

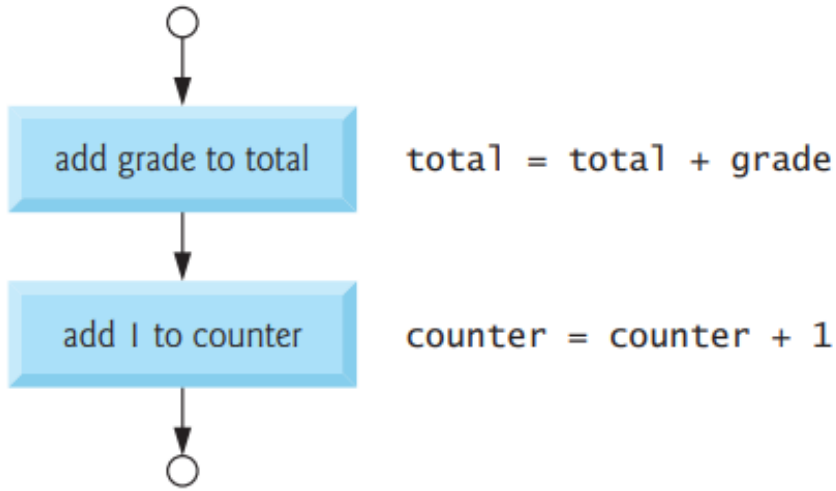
Pseudocode

- Pseudocode is an informal English-like language for "thinking out" algorithms.
- You write text that describes what your program should do.
- You then convert the pseudocode to Python by replacing pseudocode statements with their Python equivalents.
- Addition-Program Pseudocode:
  Prompt the user to enter the first integer
  Input the first integer
  ...
  Prompt the user to enter the second integer
  Input the second integer
  ...
  Add first integer and second integer, store their sum
  Display the numbers and their sum.
- Write Python statements that perform the tasks described by this section's pseudocode. Enter the integers 10 and 5.

Control Statements

- Usually, statements in a program execute in the order in which they're written. This is called **sequential execution**.
- Various Python statements enable you to specify that the next statement to execute may be **other than** the next one **in sequence**. This is called **transfer of** control and is achieved with Python **control statements**.
- Forms of Control:
  - **sequential:** The default mode of execution where statements are executed one after the other, in the order they appear.
  - **selection:** Involves decision-making in the code, where the execution flow is determined based on conditions.
  - **repetition:** Involves repeating a block of code multiple times, based on a condition or a set number of iterations.

- Flowcharts:
  - A flowchart is a *graphical representation* of an **algorithm** or a part of one.
  - You draw flowcharts using rectangles, diamonds, rounded rectangles, and small circles that you connect by arrows called flowlines.



- We use:
  - rounded rectangle: Indicates the start or end of the process
  - rectangle or action symbol: to indicate any action, such as a calculation or an input/output operation
  - arrows or flowlines: show the order in which the actions execute
  - decision or diamond symbol: Represents a decision point where the flow can branch based on a yes/no question or true/false condition
  - small circles or connector symbol: In a flowchart for only a part of an algorithm, we omit the rounded rectangles, instead using small circles called connector symbols.

if Statement

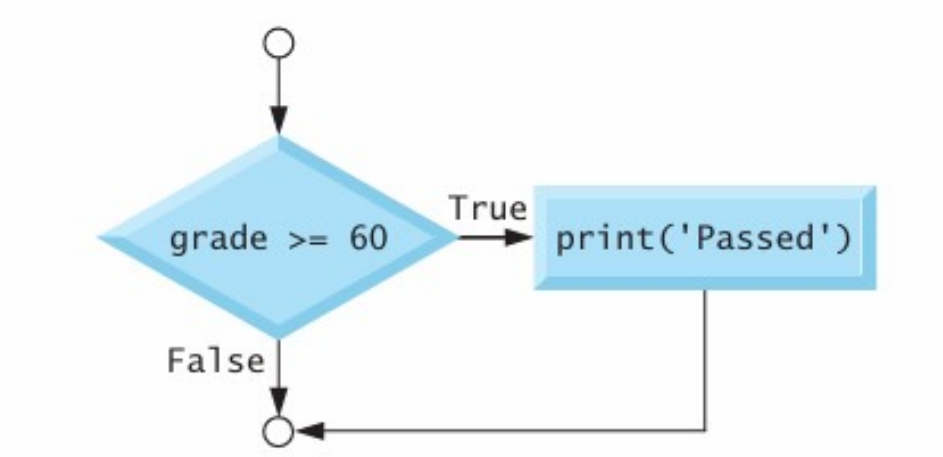
- Suppose that a passing grade on an examination is 60. The pseudocode:

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

determines whether the condition "student's grade is greater than or equal to 60" is true or false. If the condition is true, 'Passed' is displayed.

- When you use the code in python, indenting a suite is required; otherwise, an `IndentationError` syntax error occurs.
- An `IndentationError` also occurs if you have more than one statement in a suite and those statements do not have the same indentation.

- if Statement Flowchart:



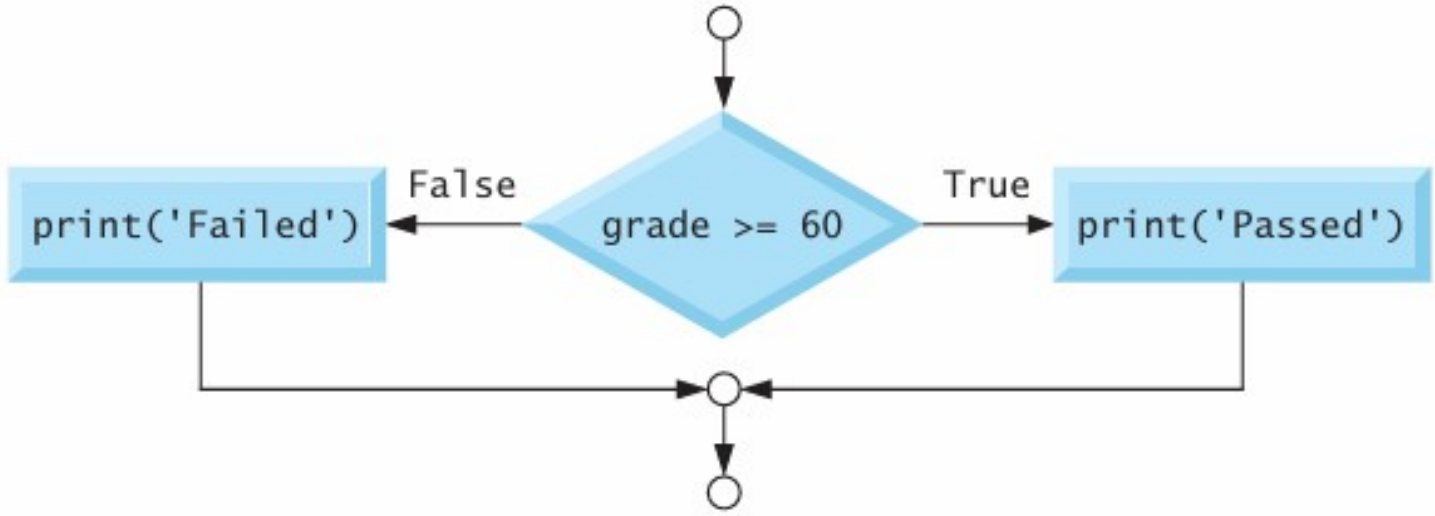

```
if grade>=60:
    print("Passed")
```

if. . . else Statements

- The if. . . else statement performs different suites, based on whether a condition is True or False.
- The pseudocode below displays 'Passed' if the student's grade is greater than or equal to 60; otherwise, it displays 'Failed':

If student's grade is greater than or equal to 60
    Display 'Passed'
else
    Display 'Failed

- if...else Statement Flowchart:




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

- Multiple Statements in a Suite✓

if. . . elif. . . else Statement

- You can test for many cases using the if. . . elif. . . else statement.

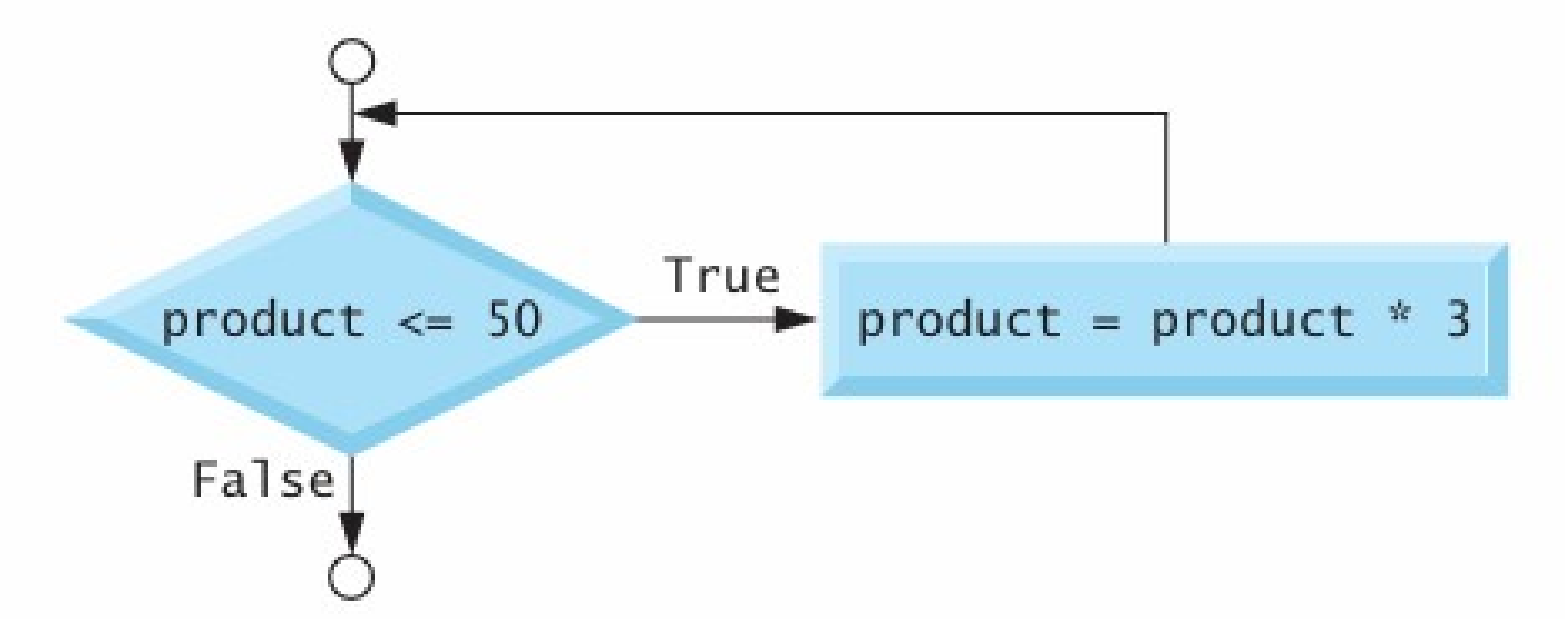
```
if grade>=90:
    print("A")
elif grade>=80:
    print("B")
elif grade>=70:
    print("C")
else:
    print("F")
```

- The else in the if. . . elif. . . else statement is optional. Including it enables you to handle values that do not satisfy any of the conditions.
- **Fatal logic error** $\rightarrow$ An error in reasoning that causes the program to produce completely wrong results (but still runs).
- **Nonfatal logic error** $\rightarrow$ A smaller mistake in reasoning that gives partly incorrect or unexpected results but does not crash the program.

while Statement

- The while statement allows you to repeat one or more actions while a condition remains True. Such a statement often is called a loop.
- The following pseudocode specifies what happens when you go shopping:

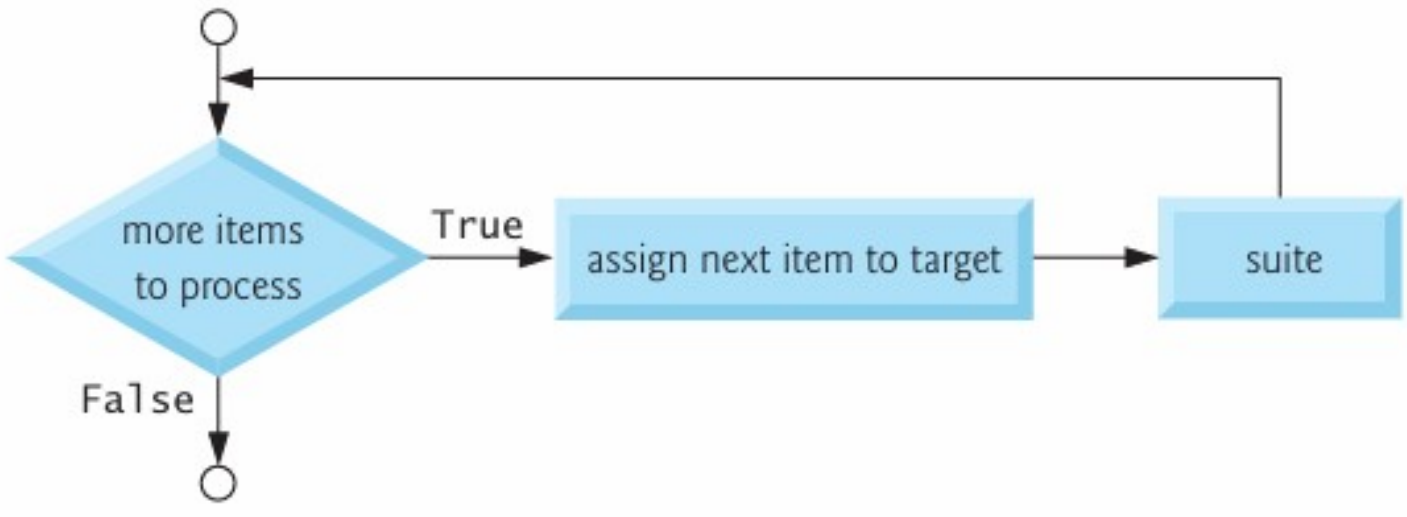
While there are more items on my shopping list
    Buy next item and cross it off my list.




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

for Statement

- Like the while statement, the for statement allows you to *repeat* an action or several actions.
- The `for` statement performs its action(s) for each item in a sequence of items. For example, a string is a sequence of individual characters.
- for Statement Flowchart:



```
for <variable> in <sequence of values>:
    <code>
     ...
```

- **Each time through the loop**, `<variable>` takes a value
- First time, `<variable>` is the first value in sequence
- Next time, `<variable>` gets the second value
- etc. until `<variable>` runs out of values.

- Function print's `end` Keyword Argument:
  - The built-in function print displays its argument(s), then moves the cursor to the next line.
  - You can change this behavior with the argument end, as in print(character, end=' ') .
  - Python calls end a keyword argument, but end is not a Python keyword.
  - The end keyword argument is **optional**.
- Function print's `sep` Keyword Argument :
  - You can use the keyword argument sep (short for separator) to specify the string that appears **between** the items that print displays.
  - When you do not specify this argument, print uses a space character by default.

- Iterables, Lists and Iterators:
  - **Iterables:** Objects that can return elements one by one (e.g., list, tuple, str, set, dict, range). Implement `__iter__()`.
  - **Lists:** Ordered, mutable iterable, defined with `[ ]`, allows duplicates. Example: `[1, 2, 3]`.
  - **Iterators:** Created from an iterable using `iter()`, provide elements via `next()`. Implement `__iter__()` and `__next__()`.
  - Relation: A list is an iterable; an iterator is derived from an iterable.
- Built-In range Function:
  - The range() function in Python is a built-in function used to generate a sequence of numbers.
  - It's commonly used in loops to iterate over a block of code a certain number of times.
  - The range() function can be called with one, two, or three arguments:
    range(stop), range(start, stop), range(start, stop, step)
    - **start** and **step** arguments are **optional**.
  - Off-By-One Errors occurs when you assume that range's argument value is included in the generated sequence.

Self Check

Q.1 (Fill-In) Function _____ generates a sequence of integers.

Q.2 (IPython Session) Use the range function and a for statement to calculate the total of the integers from 0 through 1,000,000.

Augmented Assignments

- Augmented assignments abbreviate assignment expressions in which the same variable name appears on the left and right of the assignment's $=$.

Augmented assignment	Sample expression	Explanation	Assigns
Assume: `c = 3, d = 5, e = 4, f = 2, g = 9, h = 12`			
`+=`	`c += 7`	`c = c + 7`	10 to c
`-=`	`d -= 4`	`d = d - 4`	1 to d
`*=`	`e *= 5`	`e = e * 5`	20 to e
`**=`	`f **= 3`	`f = f ** 3`	8 to f
`/=`	`g /= 2`	`g = g / 2`	4.5 to g
`//=`	`g //= 2`	`g = g // 2`	4 to g
`%=`	`h %= 9`	`h = h % 9`	3 to h

Self Check

Q.1 (Fill-In) If x is 7, the value of x after evaluating $x* = 5$ is _____.

Q.2 (IPython Session) Create a variable x with the value 12. Use an exponentiation augmented assignment statement to square x's value. Show x's new value.

Program Development: Sequence-Controlled Repetition

- The most challenging part of solving a problem on a computer is *developing an algorithm* for the solution.
- We'll see **problem solving** and **program development** by creating scripts that solve two *class-averaging problems*.

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 10 students took a quiz. Their grades (integers in range 1-100) 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:
  1. Keep a running total of the grades.
  2. Calculate the average—the total of the grades divided by the number of grades.
  3. Display the result.

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.

Coding the Algorithm in Python

Introduction to f-strings

- f-string (short for formatted string) allow embedding expressions directly in strings using {}.
- Add **f** before the opening quote:

```
x = 10
print(f"The value of x is: {x}")
# Output: The value of x is: 10

# Expressions can be used inside {}
print(f"x + 5 = {x + 5}")
print(f"Length of 'hello': {len('hello')}")
```

- Inside {}, you can put variables, calculations, or function calls.

f-string Formatting Options

- Alignment:
  - < : Left-align
  - > : Right-align (default)
  - ^ : Center-align
- Custom padding character by placing it before the alignment symbol.
- Number formatting allows precision control along with alignment.

```
name = "Jay"
print(f"{name:<10}")  # Left-align
print(f"{name:>10}")  # Right-align
print(f"{name:^10}")  # Center-align

print(f"{name:*^10}")  # Custom padding with '*'

num = 3.14159
print(f"{num:>10.2f}")
# Right-align, 2 decimal places
```

Self Check

Q.1 (Fill-In) A(n) _____ describes what a program is supposed to do, but not how the program should do it.

Q.2 (Fill-In) Many of the scripts you'll write can be decomposed into three phases: _____, _____ and _____.

Q.3 (IPython Session) Display an f-string in which you insert the values of the variables number1 (7) and number2 (5) and their product. The displayed string should be

```
7 times 5 is 35.
```

Program Development: Sentinel-Controlled Repetition

- Consider the following requirements statement:
  Develop a class-averaging program that processes an arbitrary number of grades each time the program executes.
- One way to solve this problem is to use a special value called a sentinel value (also called a signal value, a dummy value or a flag value) to indicate "end of data entry."
- The user enters grades one at a time until all the grades have been entered.
- The user then enters the sentinel value to indicate that there are no more grades.
- Sentinel-controlled repetition is often called indefinite repetition because the number of repetitions is *not* known before the loop begins executing.

Program Development: Sentinel-Controlled Repetition

Developing the Pseudocode Algorithm with Top-Down, Stepwise Refinement

- We begin with a pseudocode representation of the top: Determine the class average for the quiz
- The top is a single statement that conveys the program's overall function.
- The top specifies *what* should be done, but not *how* to implement it.
- So we begin the refinement process.
- We decompose the top into a sequence of smaller tasks—a process sometimes called divide and conquer.

- This results in the following first refinement:
  Initialize variables
  Input, sum and count the quiz grades
  Calculate and display the class average
- Each refinement represents the complete algorithm -only the level of detail varies.
- The algorithm does not yet provide enough detail for us to write the Python program.
- Second Refinement: To proceed to the second refinement, we commit to specific variables. The program needs to maintain
  - a grade variable in which each successive user input will be stored,
  - a running total of the grades,
  - a count of how many grades have been processed and
  - a variable that contains the calculated average.

- The following is the class-average problem's complete second refinement:

  Initialize total to zero
  Initialize grade counter to zero

  Input the first grade (possibly the sentinel)
  While the user has not entered the sentinel
      Add this grade into the running total
      Add one to the grade counter
      Input the next grade (possibly the sentinel)

  If the counter is not equal to zero
      Set the average to the total divided by the counter
      Display the average
  Else
      Display "No grades were entered".

```
# Initialize variables
total = 0
grade_counter = 0

# Input the first grade (sentinel value is -1)
grade = int(input("Enter grade (-1 to end): "))

# While the user has not entered the sentinel
while grade != -1:
    total += grade
    grade_counter += 1
    grade = int(input("Enter grade (-1 to end): "))

# After input loop
if grade_counter != 0:
    average = total / grade_counter
    print(f"Class average is {average:.2f}")
else:
    print("No grades were entered.")
```

Program Development: Nested Control Statements

- Consider the following requirements statement:
  A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, several of the students who completed this course took the licensing examination. The college wants to know how well its students did on the exam. You have been asked to write a program to summarize the results. You have been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam and a 2 if the student failed.
- Your program should analyze the results of the exam as follows:

1. Input each test result (i.e., a 1 or a 2). Display the message "Enter result" each time the program requests another test result.
2. Count the number of test results of each type.
3. Display a summary of the test results indicating the number of students who passed and the number of students who failed.
4. If more than eight students passed the exam, display "Bonus to instructor."

After reading the requirements statement carefully, we make the following observations about the problem:

1. The program must process 10 test results. We'll use a for statement and the range function to control repetition.
2. Each test result is a number—either a 1 or a 2. Each time the program reads a test result, the program must determine if the number is a 1 or a 2. We test for a 1 in our algorithm. If the number is not a 1, we assume that it's a 2. (An exercise at the end of the chapter considers the consequences of this assumption.)
3. We'll use two counters—one to count the number of students who passed the exam and one to count the number of students who failed.
4. After the script processes all the results, it must decide if more than eight students passed the exam so that it can bonus the instructor.

First Refinement:

- *Initialize variables*
- *Input the ten exam grades and count passes and failures*
- *Summarize the exam results and decide whether instructor should receive a bonus*

Second Refinement:...

Complete Pseudocode Algorithm:

Initialize passes to zero
Initialize failures to zero

For each of the ten students
    Input the next exam result

    If student passed
        Add one to passes
    Else
        Add one to failures
Display the number of passes
Display the number of failures

If more than eight students passed
    Display "Bonus to instructor"

```
# Initialize counters
passes = 0
failures = 0

# Process results of 10 students
for i in range(10):
    result = input(f"Enter result for student {i+1} \
    (pass/fail): ").strip().lower()
    if result == "pass":
        passes += 1
    else:
        failures += 1

# Display results
print(f"Number of passes: {passes}")
print(f"Number of failures: {failures}")

# Check for bonus
if passes > 8:
    print("Bonus to instructor")
```

Self Check

Q.1 (IPython Session) Use a for statement to input two integers. Use a nested if-else statement to display whether each value is even or odd. Enter 10 and 7 to test your code.

Function `range()` in Python

- **One argument:** Generates numbers from 0 up to (but not including) `n`.

```
for i in range(5):
    print(i)    # 0 1 2 3 4
```

- **Two arguments:** Generates numbers from `start` to `stop-1`.

```
for i in range(2, 6):
    print(i)    # 2 3 4 5
```

- **Three arguments:** Adds a `step` value.

```
for i in range(1, 10, 2):
    print(i)    # 1 3 5 7 9
```

- **Negative step:** Counts downward.

```
for i in range(10, 0, -2):
    print(i)    # 10 8 6 4 2
```

Self Check

Q.1 (True/False) Function call range(1, 10) generates the sequence 1 through 10.

Q.2 (IPython Session) What happens if you try to print the items in range(10, 0, 2)?

Q.3 (IPython Session) Use a for statement, range and print to display on one line the sequence of values 99 88 77 66 55 44 33 22 11 0, each separated by one space.

Q.4 (IPython Session) Use for and range to sum the even integers from 2 through 100, then display the sum.

Using Type Decimal for Monetary Amounts

- The `decimal` module provides accurate decimal floating-point arithmetic, useful for money or financial calculations, where binary floating-point (`float`) can give rounding errors.
- Provides exact precision (no floating-point rounding errors).
- You can set precision and rounding modes manually.
- Works with **Decimal objects** instead of `float`.
- Useful for currency, interest, or scientific data requiring accuracy.

```
from decimal import Decimal

# Normal float arithmetic
print(0.1 + 0.2)            # Output: 0.30000000000000004

# Decimal arithmetic
a = Decimal('0.1')
b = Decimal('0.2')
print(a + b)                # Output: 0.3
```

- You can safely use `Decimal` with **integers**. Python automatically converts the `int` to a `Decimal` for accurate arithmetic.

```
from decimal import Decimal

a = Decimal('10.5')
b = 2          # int
print(a + b)    # Works fine ---> Decimal('12.5')
```

- With float - Not allowed (`TypeError`)
  You cannot directly mix `float` and `Decimal`, because `float` is binary-based and `Decimal` is base-10 exact — mixing them could cause precision loss.

```
from decimal import Decimal

a = Decimal('1.1')
b = 2.2        # float
print(a + b)    # TypeError
```

break and continue Statements

- The break and continue statements alter a loop's flow of control.
- Executing a break statement in a **while** or **for** immediately *exits* that statement.
- Executing a continue statement in a **while** or **for** loop *skips* the remainder of the loop's suite.
- Finding the First Prime Number in a Range (using `break`):

```
for n in range(10, 30):
    for i in range(2, n):
        if n % i == 0:
            break # Not prime -> break inner loop
    else:
        print("First prime number in range:", n)
        break # Found first prime -> stop outer loop
-------OUTPUT---------------
First prime number in range: 11
```

- Skipping Odd Numbers (using `continue`):

```
for i in range(1, 10):
    if i % 2 != 0:
        continue    # Skip odd numbers
    print(i, "is even")
------OUTPUT-----------
2 is even
4 is even
6 is even
8 is even
```

- Password Input Example (using `break`):

```
correct_password = "python123"
for attempt in range(3):
    pwd = input("Enter password: ")
    if pwd == correct_password:
        print("Access granted ")
        break
    else:
        print("Wrong password, try again.")
else:
    print("Account locked ")
```

Boolean Operators *and*, *or* and *not*

- The conditional operators $>, <, >=, <=, ==$and $!=$ can be used to form simple conditions such as grade$>=$60.
- To form more complex conditions that combine simple conditions,use the *and*, *or* and *not* Boolean operators.
- To ensure that two conditions are **both** True before executing a control statement's suite, use the Boolean **and** operator to combine the conditions.

expression1	*expression2*	*expression1* and *expression2*
False	False	False
False	True	False
True	False	False
True	True	True

- Use the Boolean *or* operator to test whether one or both of two conditions are True.

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*. Similarly, Python stops evaluating an *or* expression as soon as it knows whether the entire condition is *True*. This is called *short-circuit evaluation*.

- The Boolean not operator "reverses" the meaning of a condition—*True* becomes *False* and *False* becomes *True*.
- This is a unary operator—it has only one operand.

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

- The precedence and grouping of the operators:

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

Self Check

Q.1 (IPython Session) Assume that `i = 1, j = 2, k = 3` and `m = 2`. What does each of the following conditions display?

(a) `(i >= 1) and (j < 4)`
(b) `(m <= 99) and (k < m)`
(c) `(j >= i) or (k == m)`
(d) `(k + m < j) or (3 - j >= k)`
(e) `not (k > m)`.

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

- We analyze data with several additional descriptive statistics, including:
  - 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.
- These are measures of central tendency-each is a way of producing a single value that represents a "central" value in a set of values.
- The Python Standard Library's `statistics` module provides functions for calculating the mean, median and mode-these too are reductions.
- To use these capabilities, first import the statistics module:

```
import statistics
```

- Then, you can access the module's functions with **statistics.** followed by the name of functions to call.
- Each function's argument must be an *iterable*.
- The mode function causes a `StatisticsError` for lists like: [1,2,2, 3,4,3] in which there are two or more "most frequent" values. Such a set of values is said to be **bimodal**.

Exercises

Q.1 (Validating User Input) Modify the following script to validate its inputs. For any input, if the value entered is other than 1 or 2, keep looping until the user enters a correct value. Use one counter to keep track of the number of passes, then calculate the number of failures after all the user's inputs have been received.

```
#file_1
"""Using nested control statements to analyze
examination results."""

# initialize variables
passes = 0 # number of passes
failures = 0 # number of failures

# process 10 students
for student in range(10):
    # get one exam result
    result = int(input("Enter result (1=pass, 2=fail):"))

    if result == 1:
        passes = passes + 1
    else:
        failures = failures + 1
```

Exercises

Q.2 (What's Wrong with This Code?) What is wrong with the following code?

```
a = b = 7
print('a =', a, '\nb =', b)
```

Q.3 (What Does This Code Do?) What does the following program print?

```
for row in range(10):
    for column in range(10):
        print('<' if row % 2 == 1 else '>', end='')
    print()
```

Exercises

Q.4 (Fill in the Missing Code) In the code below

```
for ***:
    for ***:
        print('@')
    print()
```

replace the so that when you execute the code, it displays two rows, each containing seven symbols, as in:

```
@@@@@@@
@@@@@@@
```

Exercises

Q.5 Create a script that plays the part of the independent computer, giving its user a simple medical diagnosis. The script should prompt the user with 'What is your problem?' When the user answers and presses Enter, the script should simply ignore the user's input, then prompt the user again with `'Have you had this problem before (yes or no)?'` If the user enters `'yes'`, print `'Well, you have it again.'` If the user answers `'no'`, print `'Well, you have it now.'`

Q.6 (Separating the Digits in an Integer) Write a script that separate a five-digit integer into its individual digits and display them. Implement your script to use a loop that in each iteration "picks off" one digit (left to right) using the // and % operators, then displays that digit.

Exercises

Q.7 (Nested Loops) Write a script that displays the following triangle patterns separately, one below the other. Separate each pattern from the next by one blank line. Use for loops to generate the patterns. Display all asterisks (*) with a single statement of the form

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

which causes the asterisks to display side by side. [Hint: For the last two patterns, begin each line with zero or more space characters.]

```
(a)          (b)          (c)          (d)
*            **********   **********             *
**           *********     *********            **
***          ********       ********           ***
****         *******         *******          ****
*****        ******           ******         *****
******       *****             *****        ******
*******      ****               ****       *******
********     ***                 ***      ********
*********    **                   **     *********
**********   *                     *    **********
```

Exercises

Q.8 (Miles Per Gallon) Drivers are concerned with the mileage obtained by their automobiles. One driver has kept track of several tankfuls of gasoline by recording miles driven and gallons used for each tankful. Develop a sentinel-controlled-repetition script that prompts the user to input the miles driven and gallons used for each tankful. The script should calculate and display the miles per gallon obtained for each tankful. After processing all input information, the script should calculate and display the combined miles per gallon obtained for all tankfuls (that is, total miles driven divided by total gallons used).

```
Enter the gallons used (-1 to end): 12.8
Enter the miles driven: 287
The miles/gallon for this tank was 22.421875
Enter the gallons used (-1 to end): 10.3
Enter the miles driven: 200
The miles/gallon for this tank was 19.417475
Enter the gallons used (-1 to end): 5
Enter the miles driven: 120
The miles/gallon for this tank was 24.000000
Enter the gallons used (-1 to end): -1
The overall average miles/gallon was 21.601423
```

Exercises

Q.9 (Palindromes) A palindrome is a number, word or text phrase that reads the same backwards or forwards. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554 and 11611. Write a script that reads in a five-digit integer and determines whether it's a palindrome. [Hint: Use the // and % operators to separate the number into its digits.]

Exercises

Q.10 Write a script that inputs a nonnegative integer and computes and displays its factorial. Try your script on the integers 10, 20, 30 and even larger values. Did you find any integer input for which Python could not produce an integer factorial value?

Q.11 (Challenge: Approximating the Mathematical Constant) Write a script that computes the value of π from the following infinite series. Print a table that shows the value of π approximated by one term of this series, by two terms, by three terms, and so on. How many terms of this series do you have to use before you first get 3.14? 3.141? 3.1415? 3.14159?

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \ldots$$

Exercises

Q.12 (Nested Control Statements) Use a loop to find the two largest values of 10 numbers entered.

Q.13 **(Calculate Change Using Fewest Number of Coins)** Write a script that inputs a purchase price of a dollar or less for an item. Assume the purchaser pays with a dollar bill. Determine the amount of change the cashier should give back to the purchaser. Display the change using the fewest number of pennies, nickels, dimes and quarters. For example, if the purchaser is due 73 cents in change, the script would output:

```
Your change is:
2 quarters
2 dimes
3 pennies
```

Exercises

Q.14 **(Optional else Clause of a Loop)** The `while` and `for` statements each have an optional `else` clause. In a `while` statement, the `else` clause executes when the condition becomes `False`. In a for statement, the `else` clause executes when there are no more items to process. If you break out of a `while` or `for` that has an `else`, the `else` part does not execute. Execute the following code to see that the `else` clause executes only if the `break` statement does not:

```
for i in range(2):
    value = int(input('Enter an integer (-1 to break): '))
    print('You entered:', value)

    if value == -1:
        break
else:
    print('The loop terminated without executing the break')
```

Exercise

Q.15 **(Brute-Force Computing: Pythagorean Triples)** A right triangle can have sides that are all integers. The set of three integer values for the sides of a right triangle is called a Pythagorean triple. These three sides must satisfy the relationship that the sum of the squares of two of the sides is equal to the square of the hypotenuse. Find all Pythagorean triples for `side1`, `side2` and `hypotenuse` (such as 3, 4 and 5) all no larger than 20. Use a triple-nested for-loop that tries all possibilities. This is an example of "brute-force" computing.

Q.16 (Binary-to-Decimal Conversion) Input an integer containing 0s and 1s (i.e., a "binary" integer) and display its decimal equivalent.