# Lecture-2

## Control Statements in Python

# 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 previous one in sequence. This is called transfer of control and is achieved with Python control statements.

# Keywords

The words if, elif, else, while, for, True and False are keywords that Python reserves to implement its features, such as control statements. Using a keyword as a variable name is a syntax error. The following table showing the lists Python's keywords.

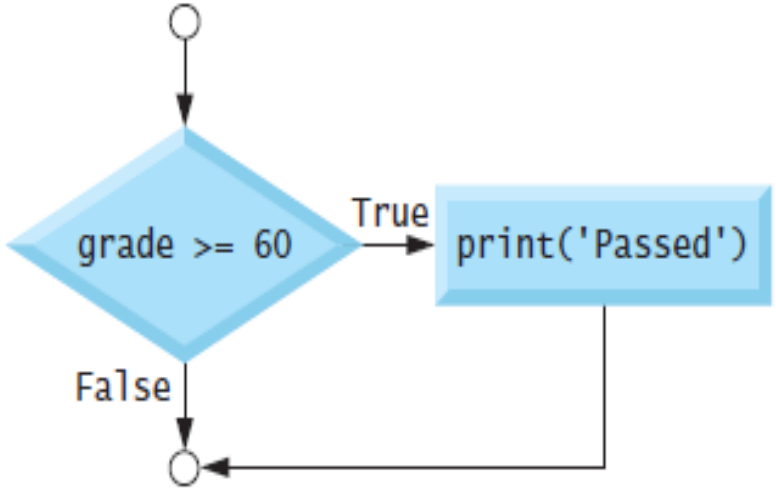
| | | | | | | |
|---|---|---|---|---|---|---|
| **and** | **as** | **assert** | **async** | **await** | **break** | **class** |
| **continue** | def | del | elif | else | except | **False** |
| **finally** | for | from | global | if | import | **in** |
| **is** | lambda | None | nonlocal | not | or | **pass** |
| **raise** | **return** | **True** | **try** | **while** | **with** | **yield** |

# if Statement

Suppose that a passing grade on an examination is 60. Then the pseudocode will be as following:

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

```
In [1]: grade = 85
In [2]: if grade >= 60:
   ...: print('Passed')
   ...: passed
```



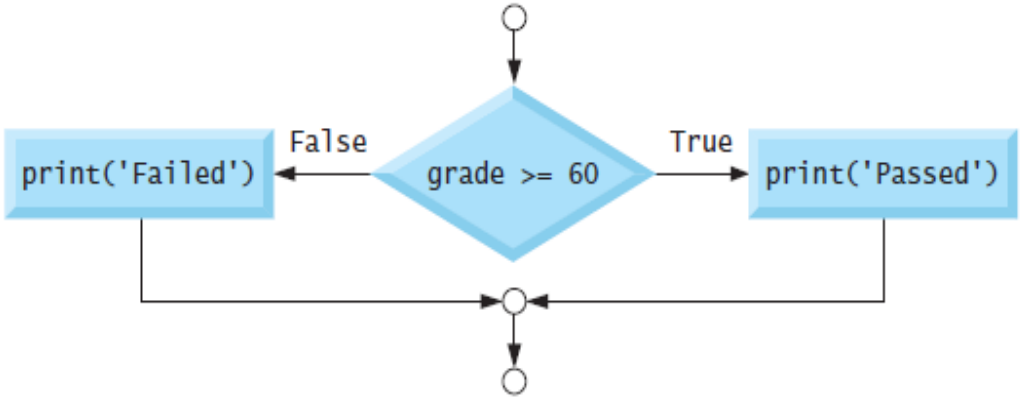

The flowchart for the if statement

# 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'*

```
In [1]: grade = 85
In [2]: if grade >= 60:
.....:     print('Passed')
.....: else:
.....:     print('Failed')
.....: Passed
```




if…else Statement Flowchart

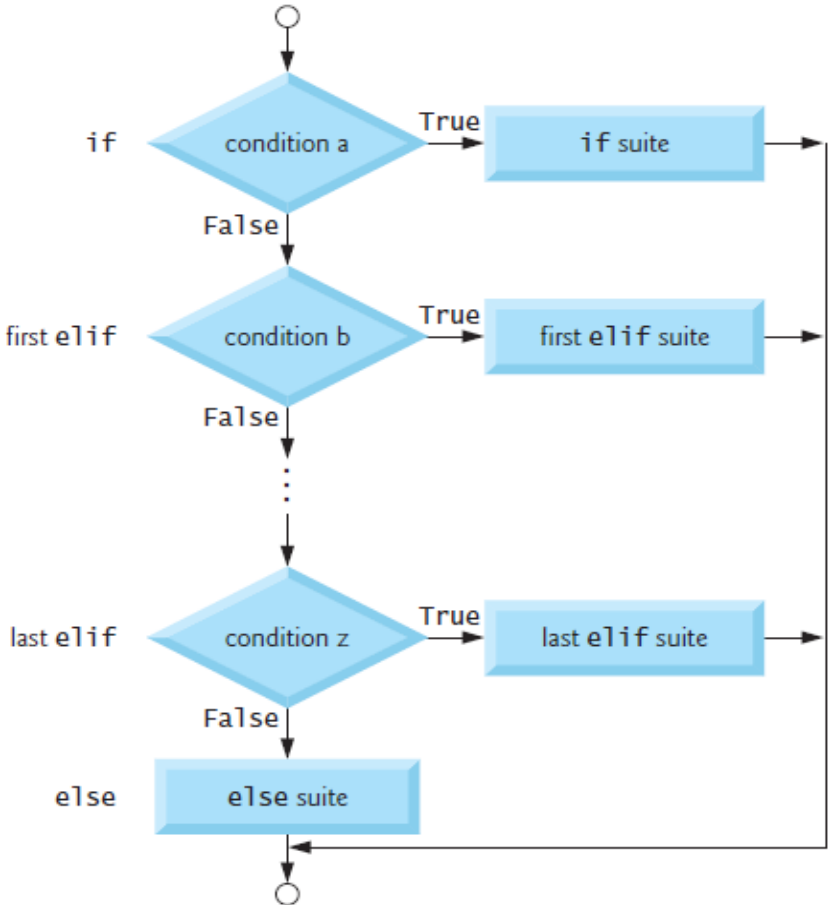
# if…elif…else Statement

You can test for many cases using the **if…elif…else statement**. The following pseudocode displays "A" for grades greater than or equal to 90, "B" for grades in the range 80–89, "C" for grades 70–79, "D" for grades 60–69 and "F" for all other grades:

```
If student's grade is greater than or equal to 90
    Display "A"
Else If student's grade is greater than or equal to 80
    Display "B"
Else If student's grade is greater than or equal to 70
    Display "C"
Else If student's grade is greater than or equal to 60
    Display "D"
Else
    Display "F"
```

# if…elif…else Statement

```
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')
```




if…elif…else Statement Flowchart

# 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:

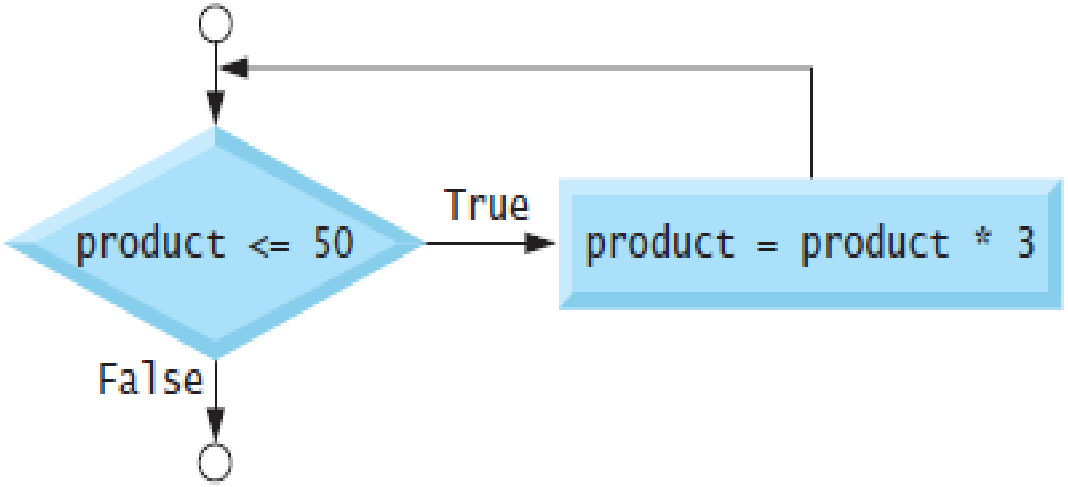
*While there are more items on my shopping list, Buy next item and cross it off my list*

If the condition "there are more items on my shopping list" is *true*, you perform the action "Buy next item and cross it off my list." You *repeat* this action while the condition remains *true*. You stop repeating this action when the condition becomes *false*—that is, when you've crossed all items off your shopping list.

# While Statement

Let’s use a while statement to find the first power of 3 larger than 50:

```
In [1]: product = 3
In [2]: while product <= 50:
……..:     product = product * 3
……..:
In [3]: product
Out[3]: 81
```

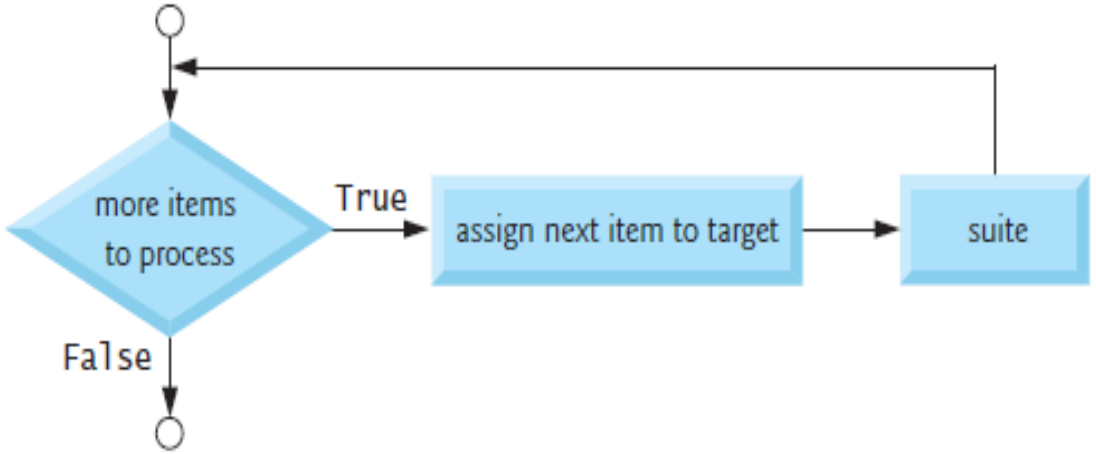



While Statement Flowchart

# For Statements

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. Let's display 'Programming' with its characters separated by two spaces:

```
In [1]: for character in 'Programming':
.......: print(character, end='  ')
.......: P  r  o  g  r  a  m  m  i  n  g
```




for Statement Flowchart

# Iterables , Lists and Iterators

The sequence to the right of the for statement's in keyword must be an **iterable**. An iterable is an object from which the for statement can take one item at a time until no more items remain. Python has other iterable sequence types besides strings. One of the most common is a **list**, which is a comma-separated collection of items enclosed in square brackets ([ and ]). The following code totals five integers in a list:

```
In [3]: total = 0
In [4]: for number in [2, -3, 0, 17, 9]:
.....: total = total + number
.....:
In [5]: total
Out[5]: 25
```

# Built-In range Function

Let's use a for statement and the built-in **range function** to iterate precisely 10 times, displaying the values from 0 through 9:

```
In [6]: for counter in range(10):
……..: print(counter, end=' ')
……..: 0 1 2 3 4 5 6 7 8 9
```

Use the range function and a for statement to calculate the total of the integers from 0 through 10.

```
In [1]: total = 0
In [2]: for number in range(10):
...:     total = total + number
...:
In [3]: total
Out[3]: 45
```

# Built-In Function range: A Deeper Look

Function range also has two- and three-argument versions. As you've seen, range's one argument version produces a sequence of consecutive integers from 0 up to, but not including, the argument's value. 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, as in:

```
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, which is known as the **step**:

```
In [2]: for number in range(0, 10, 2):
...:     print(number, end=' ')
...:     0 2 4 6 8
```

# Built-In Function range: A Deeper Look

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, as in:

```
In [3]: for number in range(10, 0, -2):
...:     print(number, end=' ')
...:     10 8 6 4 2
```

Use for and range to sum the even integers from 2 through 100, then display the sum.

```
In [1]: total = 0
In [2]: for number in range(2, 101, 2):
...:     total += number
...:
In [3]: total
Out[3]: 2550
```

# Augmented Assignments

**Augmented assignments** abbreviate assignment expressions in which the same variable name appears on the left and right of the assignment's =, as total does in:

```
for number in [1, 2, 3, 4, 5]:
            total = total + number
```

reimplements this using an **addition augmented assignment (+=) statement:**

```
In [1]: total = 0
In [2]: for number in [1, 2, 3, 4, 5]:
...:    total += number     # add number to total and store in number
...:
In [3]: total
Out[3]: 15
```

The += expression in snippet  first adds number's value to the current total, then stores the new value in total.

# 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. In the following code, range produces the integer sequence 0–15, but the loop terminates when number is 10:

```
In [1]: for number in range(15):
...:     if number == 10:
...:     break
...:     print(number, end=' ')
...:     0 1 2 3 4 5 6 7 8 9
```

# Break and continue Statements

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
```

# Python
# Program Development
# Using
# Control Statement

# Sequence-Controlled Repetition

- The most challenging part of solving a problem is developing an algorithm for the solution. Once a correct algorithm has been specified, creating a working Python program is typically straightforward.
- This section and the next going to present 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 the range 0 – 100) are 98, 76, 71, 87, 83, 90, 57, 79, 82, 94. Determine the class average on the quiz.*

# Sequence-Controlled Repetition

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.

**Pseudocode for the Algorithm**

**Step 1:** Set total to zero
**Step 2:** Set grade counter to zero
**Step 3:** Set grades to a list of the ten grades

**Step 4:** For each grade in the grades list:
Add the grade to the total
Add one to the grade counter
**Step 5:** Set the class average to the total divided by the number of grades
**Step 6:** Display the class average

# Coding the Algorithm in Python

```
"""Class average program with sequence-controlled repetition."""“

# initialization phase
1 total = 0 # sum of grades
2 grade_counter = 0
3 grades = [98, 76, 71, 87, 83, 90, 57, 79, 82, 94] # list of 10 grades
4
5 # processing phase
6 for grade in grades:
7             total += grade # add current grade to the running total
8             grade_counter += 1 # indicate that one more grade was
processed
9
10 # termination phase
11 average = total / grade_counter
12 print(f 'Class average is {average}')
```

```
Class average is 81.7
```

# Formatted Strings

Following example uses the simple **f-string** (short for **formatted string**) to format this script's result by inserting the value of average into a string like: f 'Class average is {average}' The letter f before the string's opening quote indicates it's an f-string. You specify where to insert values by using replacement-text by curly braces ({ and }). Replacement-text expressions may contain values, variables or other expressions, such as calculations or function calls.

Example:

```
print(f 'Class average is {average}')
```

# Sentinel-Controlled Repetition

Let's generalize the class-average problem. 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.

# Implementing Sentinel-Controlled Iteration

The following script implements the algorithm and shows a sample execution in which the user enters three grades and the sentinel value.

```
1   # fig03_02.py
2   """Class average program with sentinel-controlled iteration."""
3
4   # initialization phase
5   total = 0  # sum of grades
6   grade_counter = 0  # number of grades entered
7
8   # processing phase
9   grade = int(input('Enter grade, -1 to end: '))  # get one grade
10
11  while grade != -1:
12      total += grade
13      grade_counter += 1
14      grade = int(input('Enter grade, -1 to end: '))
15
16  # termination phase
17  if grade_counter != 0:
18      average = total / grade_counter
19      print(f'Class average is {average:.2f}')
20  else:
21      print('No grades were entered')
```

```
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 72
Enter grade, -1 to end: -1
Class average is 85.67
```

# Nested Control Statements

Let’s work through another complete problem. Once again, we plan the algorithm using pseudocode and we develop a corresponding Python script.

Consider the following requirements statement:

- A college offers a course that prepares students for the IELTS exam.
- Last year, several of the students who completed this course took the IELTS exam.
- The college wants to know how well their students did on the exam.
- You have been asked to write a program to summarize the results.
- You have been given a list of 10 students.
- Next to each name is written a **1** if the student passed the exam and a **2** if the student failed.

# Nested Control Statements

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.
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 the instructor can get bonus.

# Implementing the Algorithm

```
1   # fig03_03.py
2   """Using nested control statements to analyze examination results."""
3
4   # initialize variables
5   passes = 0  # number of passes
6   failures = 0  # number of failures
7
8   # process 10 students
9   for student in range(10):
10      # get one exam result
11      result = int(input('Enter result (1=pass, 2=fail): '))
12
13      if result == 1:
14          passes = passes + 1
15      else:
16          failures = failures + 1
17
18  # termination phase
19  print('Passed:', passes)
20  print('Failed:', failures)
21
22  if passes > 8:
23      print('Bonus to instructor')
```

Output

```
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 2
Enter result (1=pass, 2=fail): 2
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 2
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 2
Passed: 6
Failed: 4
```

# Getting Your Questions Answered

Online forums enable you to interact with other Python programmers and get your Python questions answered. Popular Python and general programming forums include:

- python-forum.io
- StackOverflow.com
- https://www.dreamincode.net/forums/forum/29-python/
- Colab Notebook for lab

# Thank You