Python Programming

Module 1: Python Fundamentals

Lesson 3: Control Flow Statements

Neba Nfonsang
University of Denver

Lesson 3: Control Flow Statements

- Introduction to conditional flow statements
- Conditional statements
  - If,
  - If-else,
  - If-elif-else
- Loops
  - for loops and while loops

A flight descending over the silicon valley. The pilot controls how the plane lands.

Introduction to Control Flow Statements

- The **control flow** of a program is the order in which the program's code is executed.
- Python statements are executed sequentially from top to bottom.
- Control flow statements change a program's control flow by conditionally and/or repeatedly executing certain blocks of code.

Control Flow Statements

- Control flow statements in Python basically consist of:
  - Conditional statements (if, if-else, if-elif-else statements)
  - Loops (for and while loops)
- Control flow statements are compound statements.
- So, control flow statements are used for repetitive execution of code and/or conditional execution of code.
- Note that other statements such as **break** and **continue** also accompany control statements

Conditional Statements

- Conditional statements are also called branching statements.
- Conditional statements allow programs to make decision or select a course of action based on whether some test conditions are True or False.

Conditional Statements

- Conditional statements are statements that allow the execution a block(s) of code only when a condition is True, otherwise, the block of code is skipped.
- Conditional statements basically consist of a header line and an indented block(s) of code.

Basic structure of a conditional statement

keyword condition:
block of code

- **Keyword**: we could have a keyword such as **if** or **elif**
- **Condition:** the condition is a Boolean expression
- **code block(s):** Examples are print statements, etc.

if statements

- An if statement is the simplest form of a conditional statement.
- The header line of the if statement starts with the if keyword, followed by the condition, and ends with a colon(:).
- The header line is then followed by an indented body of code which is executed when the condition or Boolean expression evaluates to True.
- The general syntax for an if statement is:

```
if condition:
        block of code
```

How if Statements Work

if statement

```
# block of code is executed if
# condition is True
if True:
    print("yes, condition is true, hip hip hooray!")
print("this statement will always be printed")
```

```
yes, condition is true, hip hip hooray!
this statement will always be printed
```

```
# the block of code is not executed
# when condition evaluates to false
# the block of code is skipped and next
# line of code is executed.
if False:
    print("hmm, condition is false, no way to print this!")
print("this statement is always printed")
```

```
this statement is always printed
```

if statement in Action

```
1  luggage_weight = input("How many pounnds does your luggage weigh? ")
2  luggage_weight = float(luggage_weight)
3
4  if luggage_weight > 50:
5      print("Your luggage weight is more than 50 pounds\n"
6            "You will be charged $30 for the extra weight")
7  print("Thank you for your business")
```

```
How many pounnds does your luggage weigh? 60
Your luggage weight is more than 50 pounds
You will be charged $30 for the extra weight
Thank you for your business
```

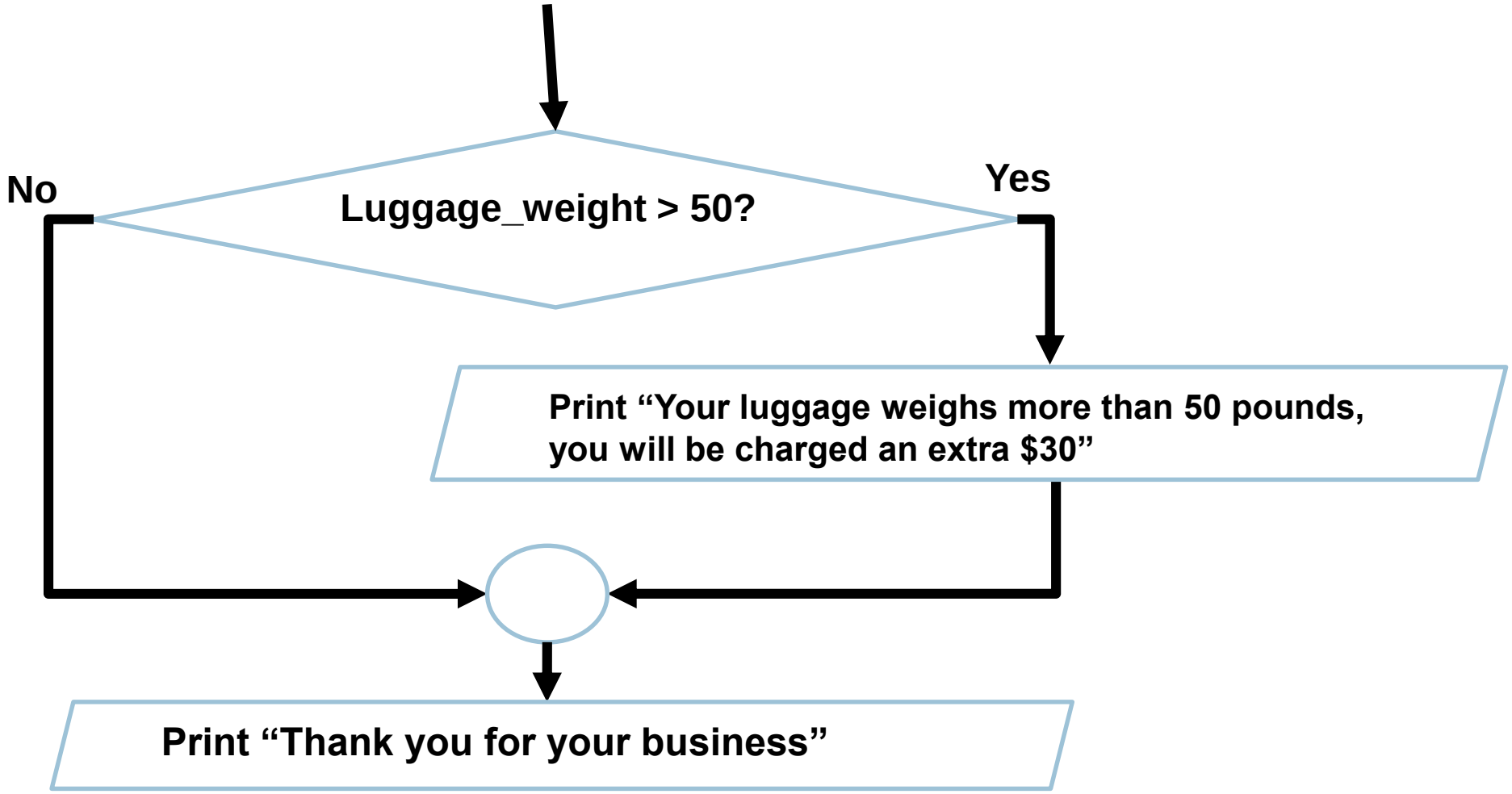
A program that collects weight of bag and prints a message if weight is greater than 50 pounds

if statement in Action

```
1  luggage_weight = input("How many pounnds does your luggage weigh? ")
2  luggage_weight = float(luggage_weight)
3
4  if luggage_weight > 50:
5      print("Your luggage weight is more than 50 pounds\n"
6            "You will be charged $30 for the extra weight")
7  print("Thank you for your business")
```

```
How many pounnds does your luggage weigh? 40
Thank you for your business
```

Control Flow Diagram for an if Statement



if-else Statement

- The if-else has an additional else clause in addition to the if statement that we have already examined.
- The **else** clause ends with a colon(:) followed by an indented block of code on the next line, which is executed only when the condition of the if statement evaluates to false

- The general syntax for an if-else statement is:

```
if condition:
        block of code
else:
        block of code
```

Using if-else statement for Error Checking

```
1  # using if-else statement to check for
2  # invalid input or errors
3  area = input("Enter area your circle: ")
4  area = eval(area)
5  pi = 3.14
6  if area > 0:
7      radius = (area/pi)**0.5
8      radius = round(radius, 2)
9      print("The radius of your circle is ", radius)
10 else:
11     print("Error: your area must be a positive number")
```

A program that collects the area of a circle and calculates the radius of the circle

```
Enter area your circle: -100
Error: your area must be positive numbers
```

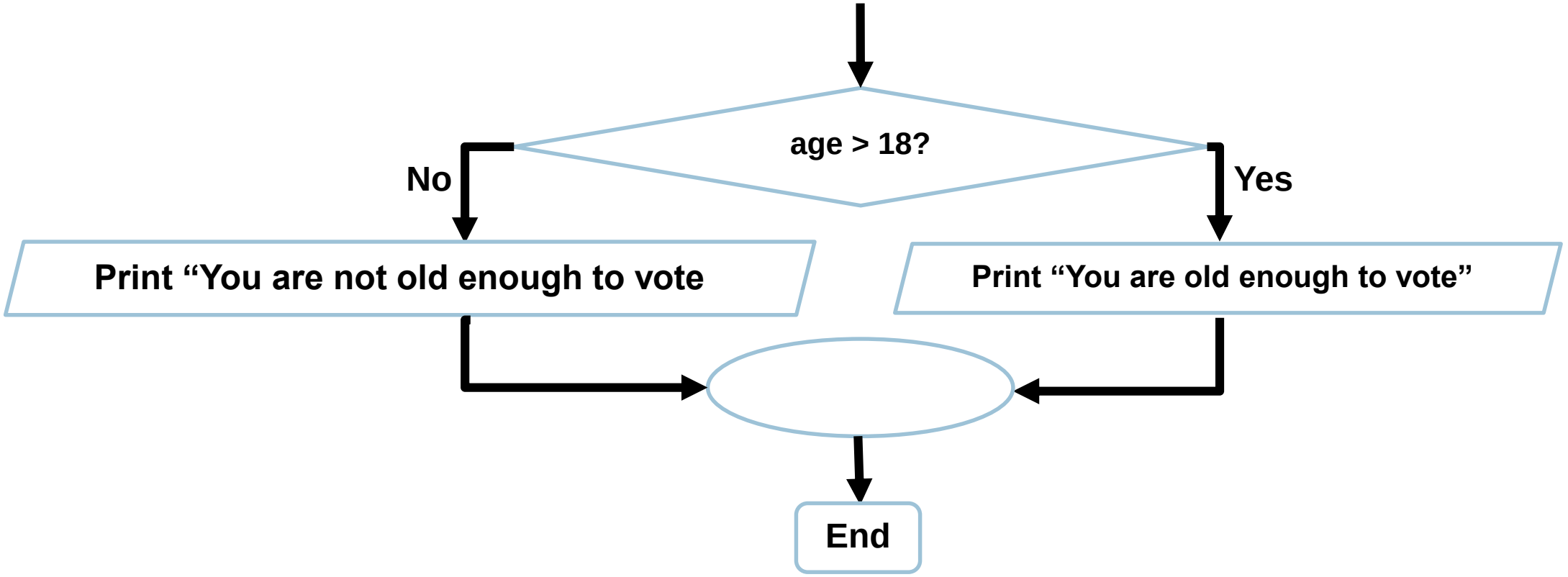
if-else Statement in Action

- An if-else statement is used when there is an alternative option given that the if condition evaluates to false. That implies mutually exclusive options are possible.

```
1  age = input("Please, enter your age: ")
2  age = int(age)
3
4  if age >= 18:
5      print("You are old enough vote!")
6  else:
7      print("You are not old enough to vote!")
```

```
Please, enter your age: 15
You are not old enough to vote!
```

Control Flow Diagram for if-else Statement



if-elif-else Statements

- If-elif-else statements are used when there are three or more mutually exclusive options.
- The condition of the if statement is tested and the code body of the if-statement is executed given that the if condition evaluated to True.
- When the if condition evaluates to False, the elif condition is tested and executed when True.
- If the elif condition evaluates to false, then the body of code nested under the else statement is executed.

if-elif-else Statements

- Note that, if there are more than 3 conditions, you just need to keep adding more elif parts of the code as needed.
- Sometimes, you need to make sure that all the options considered are not only mutually exclusive but collectively exhaustive.
- In situations where user input is involved, you may want to include an option that handles wrong input.

```
if condition:
        block of code
elif condition:
        block of code
…
…
else:
        block of code
```

if-elif-else statement

- Let’s create a program that tells a potential employee what salary they would earn for a particular position based on their highest level of education. It is assumed that the employee has earned an undergraduate, master’s or PhD degree.
- Given that the base salary is $50000, an employee with an undergraduate, master’s or PhD degree should earn 10%, 20% or 30% of the base salary respectively, in addition to the base salary.

if-elif-else Statements in Action

```
1  edu_level = input("Enter your highest education level\n"
2                    "enter u for undergraduate, m for master's\n"
3                    "or p for phd: ")
4  if edu_level == "u":
5      salary = 50000*0.1 + 50000
6  elif edu_level == "m":
7      salary = 50000*0.2 + 50000
8  else:
9      salary = 50000*0.3 + 50000
10 print("Your salary for this position will be",'%.2f' % salary)
```

A program that collects level of education and prints salary based on educational level

```
Enter your highest education level
enter u for undergraduate, m for master's
or p for phd: m
Your salary for this position will be 60000.00
```

Exercise

- Write a program that collects revenue and cost and displays profit, loss, or break even!

Exercise

- Write a program that collects two numbers, compares and prints a message about whether the numbers are equal or not.

Loops

- Python’s two main loop constructs are **for loops** and **while loops.**
- Loops are generally used for repetitive or iterative task.
- for loops are used when the number of iterations is known.
- while loops are generally used when the number of iterations is not known.

for loop

- A for loop is a control statement that allows a block of code to be executed repeatedly for a specific number of times.
- The number of repeated executions in a for loop is determined by the length of the iterable in the loop's header line.
- **The number of iterations is known ahead of time.**
- A for loop is usually used if a task needs to be done repeatedly on some elements of a sequence or iterable where number of iterations are known.

for loop

- There are situations where the number of iterations are not known.
- If you choose a target number and ask a user to guess what that number is, you don’t know how long it will take the user to guess the target number.
- The user may even quit without guessing the correct number. In that case, the number of iterations is unknown. A while loop will be better for such situations.
- So, a for loop differs from a while loop based on the fact that its iterations are always definite.

Components of a for loop

The following are the components of for loop:

- The **for** keyword
- The loop **variable**
- The **in** keyword
- The **iterable**
- **Colon(:)**
- **A block of code**

```
for variable in iterable:
        block of code (do
        something)
```

How a for Loop Works

- When a for loop is run, the first item of the iterable is accessed and assigned to the loop variable. The code block is then executed. The second item in the iterable is accessed and assigned to the loop variable and code block is executed... (this process is repeated until the last item is accessed).

```
for variable in iterable:
        block of code (do something)
```

- When the last item in the iterable is accessed and the block of code is executed, the loop stops.

Loop Variables and Iterables

- The **loop variable** is also called the **iteration or control** variable.
- The loop variable references the values in the iterable one-by-one, after each iteration until all the values in the iterable are referenced. Each time a value is referenced, the previously referenced value is overwritten.
- An iterable is any object that you can get an iterator from. An iterable has the __item__() or __getitem__() method. Examples are all sequences, dictionaries and files objects.

for loop in Action – indentation matters!

```
1  # a for loop that prints numbers in a list
2  my_list = [1, 2, 3, 4]
3  for num in my_list:
4      print(num)
```

```
1
2
3
4
```

You would notice that, when the print statement is indented to the same level as the code block, the print statement is executed for every iteration. When the print statement is not indented, it is executed only after the last iteration and only the last results will be printed.

```
1  # a for loop that squares the numbers
2  # in my_list
3  my_list = [1, 2, 3, 4]
4  for num in my_list:
5      squared_num = num**2
6      print(squared_num)
```

```
1
4
9
16
```

```
1  # a for loop that squares the numbers
2  # in my_list
3  my_list = [1, 2, 3, 4]
4  for num in my_list:
5      squared_num = num**2
6  print(squared_num)
```

```
16
```

for loop with the **range()** Function

```
1  # a for loops that collects a name
2  # from a user three times
3  for item in range(3):
4      number = input("Enter a name: ")
```

```
Enter a name: Nicole
Enter a name: Gabriel
Enter a name: Juan
```

```
1  range()
```

```
Init signature: range(self, /, *args, **kwargs)
Docstring:
range(stop) -> range object
range(start, stop[, step]) -> range object
```

- The range (start, stop[, step]) specifies a range of numbers from start up to but not including stop number, in increments of step number.
- If only one number is passed into the function, it would be considered a stop number, start number will default to 0 while step defaults to 1

for loop with User Input/Append Method

```
1  # a for loop that squares the numbers
2  # in my_list and stores them in a list
3  my_list = [1, 2, 3, 4]
4  squared_list = []  # initialize an empty list
5  for num in my_list:
6      squared_num = num**2
7      squared_list.append(squared_num)
8  print(squared_list)
```

```
[1, 4, 9, 16]
```

A program that loops throughs numbers in a list, square the numbers and store them in another list

for loop with if Statement

```
1  # output numbers from 10 up to not
2  # including 20 using the range function
3  range(10, 20)
```

```
range(10, 20)
```

```
1  # create a list from the range() function
2  number_list = list(range(10, 20))
3  print(number_list)
```

```
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```
1  # using for loop with a conditional statement
2  # print odd numbers
3  number_list = list(range(10, 20))
4  for number in number_list:
5      if number%2 == 1:
6          print(number)
```

```
11
13
15
17
19
```

A program that prints only odd numbers in a list

Count with for loops

```
1  # let's count how many numbers
2  # are divisible by 3 in the number_list
3  # initialize counter to start from zero
4  number_list = list(range(10, 20))
5  counter = 0
6  for number in number_list:
7      if number%3 == 0:
8          counter = counter + 1
9  print(counter)
```

```
3
```

A program that counts how many numbers in a list are divisible by 3

Count with for loops

```
1  # let's add more information
2  number_list = list(range(10, 20))
3  counter = 0
4  for number in number_list:
5      if number%3 == 0:
6          counter = counter + 1
7          print(number, "is divisible by three")
8      else:
9          print(number, "is not divisible by three")
10 print("How many numbers are divisible by three? ", counter)
```

```
10 is not divisible by three
11 is not divisible by three
12 is divisible by three
13 is not divisible by three
14 is not divisible by three
15 is divisible by three
16 is not divisible by three
17 is not divisible by three
18 is divisible by three
19 is not divisible by three
How many numbers are divisible by three?  3
```

Multiplication with for loops

```
1  # let's multiply all the numbers
2  # in number list
3  prod = 1
4  for number in number_list:
5      prod = prod*number
6      print(prod)
7  # print function indented to
8  # show results for each iteration
```

```
10
110
1320
17160
240240
3603600
57657600
980179200
17643225600
335221286400
```

A program that multiplies all numbers in a list

Nested for loops

```
1  # create a multiplication table
2  # try to understand what is going on
3  # we will use nested for loops
4  for num1 in range(1,4):
5      for num2 in range(1,5):
6          result = num1*num2
7          print(num1, "x", num2, "=", result, "\t", end="")
8      print()
```

```
1 x 1 = 1       1 x 2 = 2       1 x 3 = 3       1 x 4 = 4
2 x 1 = 2       2 x 2 = 4       2 x 3 = 6       2 x 4 = 8
3 x 1 = 3       3 x 2 = 6       3 x 3 = 9       3 x 4 = 12
```

for loop: find minimum number

```
1  # a for loop that finds the
2  # the minimum number in a list
3  # do no use sort or min method
4  my_list = [3, 2, 5, 1, 4]
5
6  # initialize the lowest number
7  lowest = my_list[0]
8
9  for number in my_list:
10     if number < lowest:
11         lowest = number
12 print(lowest)
```

```
1
```

Standard Deviation with for Loop

```
# calculate the standard deviation
# of scores in a list
my_scores = [95, 90, 90, 87, 100, 92]
dev_square_list = []
total_score = 0
total_dev_square = 0

# compute mean
for score in my_scores:
    total_score += score
mean = total_score/len(my_scores)

# compute square deviations and
# store in a list
for score in my_scores:
    deviation = score - mean
    deviation_square = deviation**2
    dev_square_list.append(deviation_square)

```

```
# compute standard deviation
# use df = len(dev_square_list)-1
# if sample standard deviation
df = len(dev_square_list)
for dev_square in dev_square_list:
    total_dev_square += dev_square
variance = total_dev_square/df
standard_deviation = variance**0.5
standard_deviation
```

```
4.189935029992179
```

```
# check the answer using np.std()
import numpy as np
np.std(my_scores)
```

```
4.189935029992179
```

while loop

- A while loop repeatedly executes a block of code when a certain condition is true, otherwise, the loop breaks and control jumps out of the loop to execute the statement(s) that follow(s) the while block.
- A while loop starts with a header line that ends with a colon, followed by an indented block of code.

General form of the while loop

```
while condition:
        block of code
```

- **A while loop consists of:**
- A while keyword
- A condition (Boolean expression)
- Colon
- Indented block of code

while loop

```
1  # display from 1 - 5
2  # the last value of x is printed
3  x = 0
4  while x < 5:
5      x = x + 1
6      print(x)
```

```
1
2
3
4
5
```

A program that prints numbers from 1 - 5

```
1  # display from 1 - 5
2  # the last value of x is not printed
3  x = 1
4  while x < 6:
5      print(x)
6      x = x + 1
```

```
1
2
3
4
5
```

A **while loop** with an Optional **else** Clause

```
1  # an else statement is optional
2  # so, a while-else statement
3  # could be used
4  x = 0
5  while x < 5:
6      x += 1
7      print(x)
8  else:
9      print("Done")
```

```
1
2
3
4
5
Done
```

```
1  # an equivalent of the
2  # while-else statement
3  x = 0
4  while x < 5:
5      x += 1
6      print(x)
7  print("Done")
```

```
1
2
3
4
5
Done
```

When a while-else statement is optionally used, the code block before the else clause is executed completely before the code block after the else clause.

While loop

- A while loop that runs continually and needs to be manually stopped is called an **infinite loop.**
- An example of an infinite loop

```
1  # this while loop is infinite
2  while True:
3      print("hello")
```

- Don’t run a while loop that does not break unless you are trying to heat up your computer on a cold winter day☺
- A **break** statement is usually used to terminate an infinite loop

Using the break statement

- A break statement is used to break out of the loop.
- This statement is used to terminate an infinite loop or just any loop when a certain condition is met.
- **while True:** could be used as the header for an infinite while loop.

```
1  # using break statements to terminate
2  # an infinite while loop
3  x = 0
4  while True:
5      x = x + 1
6      print(x)
7      if x == 5:
8          break
```

```
1
2
3
4
5
```

Using the **continue** statement

- The continue statement moves or returns control back to the top of the loop (to the header line) when some condition is met.

```
1  # using continue
2  x = 0
3  while True:
4      x = x + 1
5      if x%2 == 0:
6          continue
7      print(x)
8      if x >= 9:
9          break
```

```
1
3
5
7
9
```

while loop and **break** Statement

```
1  # application of while loop with break
2  # a while loop that collects names
3  name_list = []
4  while True:
5      name = input("Enter a name\n"
6                   "otherwise, enter q to quit: ")
7      if name == "q":
8          break
9      name_list.append(name)
10 print(name_list)
11
```

```
Enter a name
otherwise, enter q to quit: John
Enter a name
otherwise, enter q to quit: Jackson
Enter a name
otherwise, enter q to quit: Mary
Enter a name
otherwise, enter q to quit: q
['John', 'Jackson', 'Mary']
```

Counting with a while loop

```
1  # count from 0 start up to 10
2  # stop number is excluded
3  # increment by 2
4  start = 0
5  step = 2
6  stop = 10
7  while start < stop:
8      print(start)
9      start += step
```

```
0
2
4
6
8
```

```
1  # count backward
2  # when x is zero, the loop stops
3  # if x will never be zero, the
4  # loop will become infinite
5  x = 10
6  while x:
7      x = x - 2
8      print(x)
```

```
8
6
4
2
0
```

Note that **while 0**: is the same as **while False:** because 0 is equivalent to a Boolean False.

A while loop that Sorts Numbers in a List

```
1  # use while loops to sort a list
2  # sort in ascending order
3  my_list = [3, 2, 5, 1, 4]
4  sorted_list = []
5  while my_list:
6      lowest = my_list[0]
7      for number in my_list:
8          if number < lowest:
9              lowest = number
10     sorted_list.append(lowest)
11     index = my_list.index(lowest)
12     my_list.pop(index)
13 print(sorted_list)
```

```
[1, 2, 3, 4, 5]
```

A program that sorts numbers in a list without using the minimum or sort method.

Using a Found or Flag variable

- Found variables provide a way to tracking values in a loop.
- The flag variable is initialized with a Boolean False, when some condition is true, the variable is turned on to True.
- You want to collect scores of students from a teacher and store the scores in a list, then find the class average.
- You also just want to know if any student had a hundred. You can use a found variable to do this.
- A found variable is useful when knowing that an event happened is necessary in the future, so you need to check the occurrence at the end of the loop.

Flag or Found Variable Example

```
found = False
score_list = []
total = 0

while True:
    score = input("Enter scores one-by-one\n"
                  "hit enter to quit: ")
    if score == "": break
    score = eval(score)
    if score == 100:
        found = True
    total += score
    score_list.append(score)

class_average = total/len(score_list)

print("********************************")
print("Class Average: ", class_average)
print("Did any student have a score of 100?")
if found:
    print("Yes, a student was found with a score of 100")
else:
    print("No, no student had a score of 100")
print("Score List: ", score_list)
```

Output

```
Enter scores one-by-one
hit enter to quit: 90
Enter scores one-by-one
hit enter to quit: 80
Enter scores one-by-one
hit enter to quit: 100
Enter scores one-by-one
hit enter to quit:
********************************
Class Average:  90.0
Did any student have a score of 100?
Yes, a student was found with a score of 100
Score List:  [90, 80, 100]
```

Flag or Found Variable Example

```
1  found = False
2  score_list = []
3  total = 0
4
5  while True:
6      score = input("Enter scores one-by-one\n"
7                    "hit enter to quit: ")
8      if score == "": break
9      score = eval(score)
10     if score == 100:
11         found = True
12     total += score
13     score_list.append(score)
14
15 class_average = total/len(score_list)
16
17 print("*******************************")
18 print("Class Average: ", class_average)
19 print("Did any student have a score of 100?")
20 if found:
21     print("Yes, a student was found with a score of 100")
22 else:
23     print("No, no student had a score of 100")
24 print("Score List: ", score_list)
```

- Output

```
Enter scores one-by-one
hit enter to quit: 80
Enter scores one-by-one
hit enter to quit: 70
Enter scores one-by-one
hit enter to quit: 85
Enter scores one-by-one
hit enter to quit:
*******************************
Class Average:  78.33333333333333
Did any student have a score of 100?
No, no student had a score of 100
Score List:  [80, 70, 85]
```

Program Development and Tools

- Let’s take a look at:
- Program planning
- Flowcharts
- Pseudocode
- Good Practices

Pseudo Code

- A pseudocode is a human language version of actual computer code that helps programmers develop or describe algorithms.
- Pseudocode are descriptions or outlines of the steps involved in solving a problem
- Well written pseudocodes could be easily converted into Python codes
- Pseudocodes are useful for planning what the program will do. Pseudocodes are not Python codes and therefore cannot be executed by the interpreter

An Example of a Pseudocode

- *# write a program to add numbers in a list*
- *# use the name **my_list** to reference the list*
- *# initialize the total to zero*
- *# for number in my_list:*
  - *# total equals total plus number*
- *# print “total”*

```
1  # write a program to add numbers in a list
2  my_list = [20, 30, 40, 50, 60]
3  total = 0
4  for number in my_list:
5      total += number
6  print("Total: ", total)
```

```
Total:  200
```

Program Planning

- Many programmers plan their programs using a sequence of steps called **Software Development Life Cycle,** consisting of the following step-by-step process.

- **Analyze:** During this stage, the programmer defines the problem by clearly understanding and stating what the program will do. The input and output of the program are understood.

Program Planning

- **Design:** The solution to the problem is planned at this stage by finding the sequence of steps or the algorithm for the program. How will the program accomplish its task?
- **Implementation**: This is the coding phase where the program is written and the algorithm coded.
- **Integration:** For larger programs, there is a need for different parts of the program to be brought together into functioning whole.
- **Test and Maintenance:** A program could have a lifespan of 5 to 15 years. During this time, errors are detected and removed with maintenance done.

Program Planning

- **Complete the documentation:** organize the documentation for the program so that other developers and users of the program can understand the program.
- Internal documentation includes comments
- Other types of documentations include manuals especially for commercial programs.
- Pseudocode, flowcharts, etc could also be part of the documentation.
- Documentation should be happening through out the development life cycle.

Flowchart – Symbols

Flowline: connects symbols

Processing: used for arithmetic and data manipulation operations

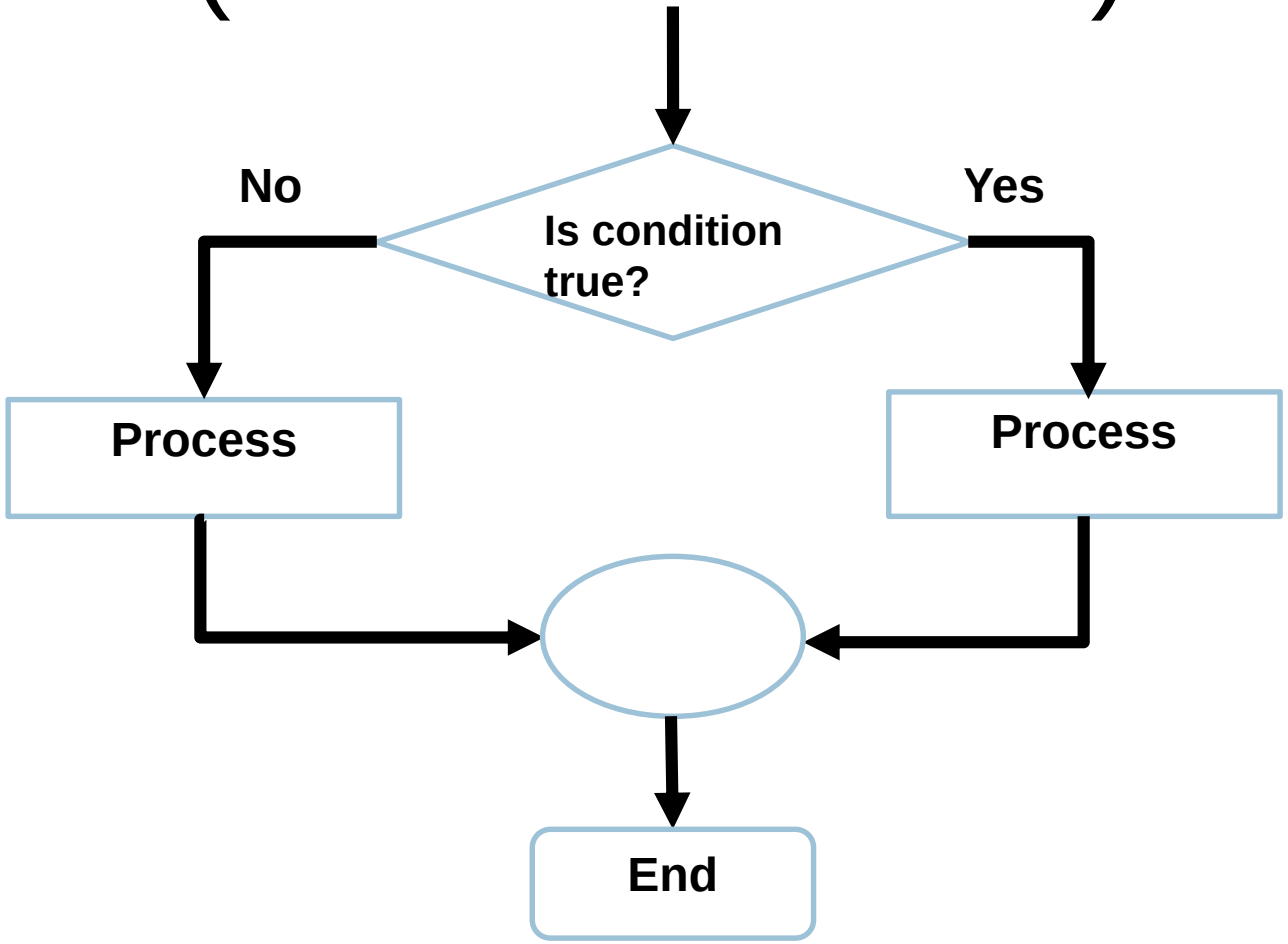
Connector : joins different flowlines

Terminal: represents start or end of task

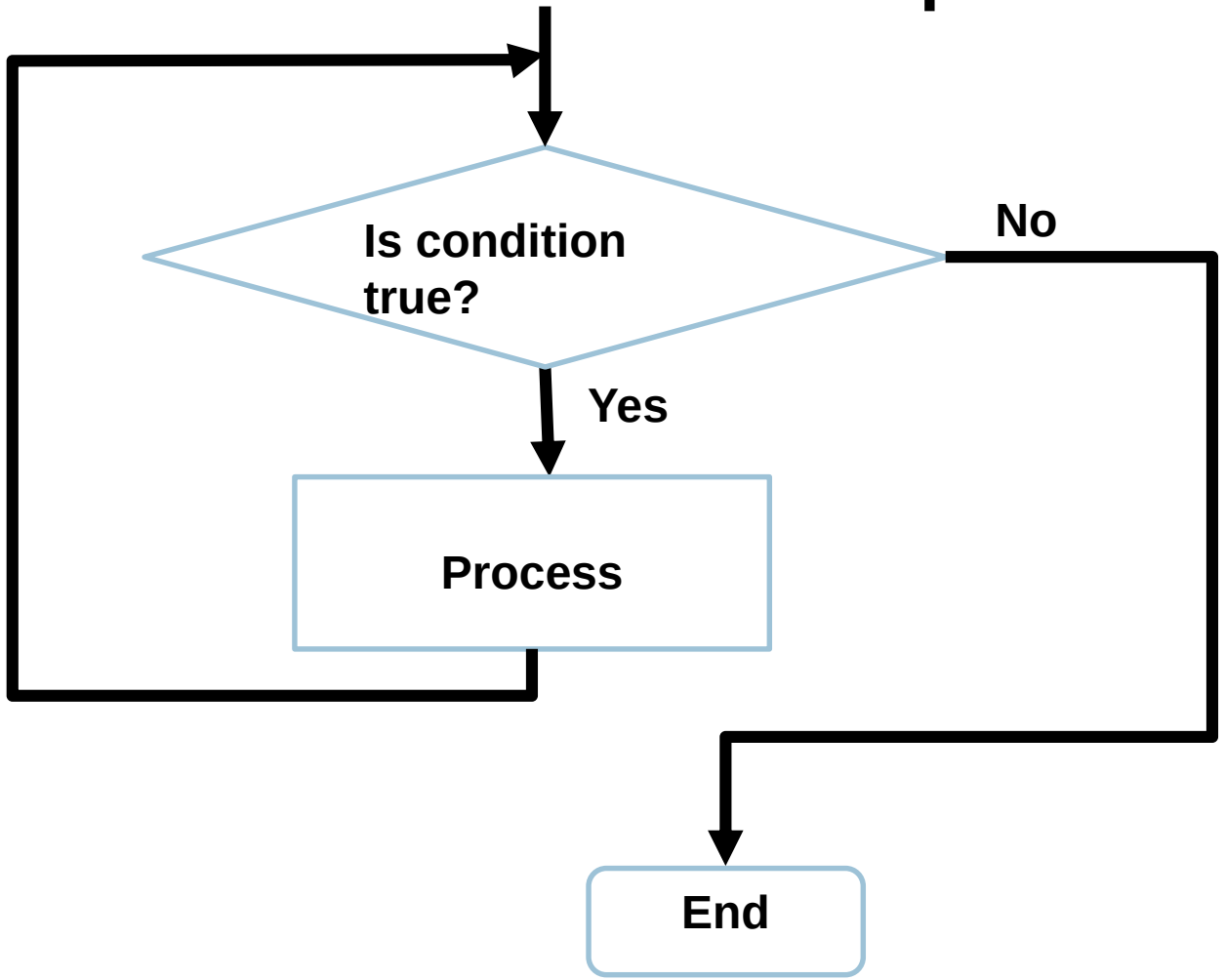
Input/Output: Used to input or output data

Decision: used for any logic or comparison operation. One entry and two exits.

General flowchart for a conditional statement (if-else statement)



General Flowchart of a Loop



Good Programming Practice

- **Comments:** use comments through out the program. Comments help others to understand the program and to debug the program.
- Comments are also helpful when you revisit your code to modify it or update it.
- Every program should start with a comments that briefly describes what the program does.
- Use blank lines to enhance readability.

Good Programming Practice

- **Variable names:** Choose variable names that are meaningful. It is better to understand a program by just reading the program than to read excessive comments and documentations.
- **Spaces**: Avoid having spaces between binary operators such as **==**.
- **Avoid reversing the order of comparison operators:** For example, use != and >= instead of =! and =>.

Good Programming Practice

- **Avoid using float variables for counting of loops:** That is, your counter should reference integer values. This is because floats are approximates and may result to inaccurate test for termination of loop.
- **Insert a blank line before and after each control structure** to increase readability.
- **Avoid changing the value of the control or loop variable** in a for loop.

Escape Sequence

- \n: Move cursor to a newline
- \t: Horizontal TAB
- \': single quote
- \": Double quote
- \\: Backslash (\)

Escape Sequence

```
1 # newline
2 print("Hello John:\nHow are you doing today?")
```

```
Hello John:
How are you doing today?
```

```
1 # newline
2 print("Hello John: \n"
3       "How are you doing today?")
```

```
Hello John:
How are you doing today?
```

```
1 # horizontal tab
2 print("fruits\t amount\n mango\t 20\n apple\t 30")
```

```
fruits   amount
 mango   20
 apple   30
```

Escape Sequence

```
1 # using both double and single quotes
2 print("It's time to learn Python")
```

```
It's time to learn Python
```

```
1 # using single quotes with escape sequence
2 print('It\'s time to learn Python')
```

```
It's time to learn Python
```

```
1 # using double quotes with escape sequence
2 print("It\"s time to learn Python")
```

```
It"s time to learn Python
```

```
1 # using single quotes with no escape sequence
2 # this will generate an error message
3
4 print('It's time to learn Python')
```

```
  File "<ipython-input-17-d28b71cfd372>", line 2
    print('It's time to learn Python')
                ^
SyntaxError: invalid syntax
```

```
1 # using double quotes with no escape sequence
2 # this will generate an error
3
4 print("It"s time to learn Python")
```

```
  File "<ipython-input-24-794ba4fcdbb4>", line 4
    print("It"s time to learn Python")
                ^
SyntaxError: invalid syntax
```

Yes, you made it to the end of Lesson 3

- Your Python muscles are getting stronger!
- Congratulations!
- Let’s head on to Functions and Modules. I’ll see you in lesson 4

End of Lesson