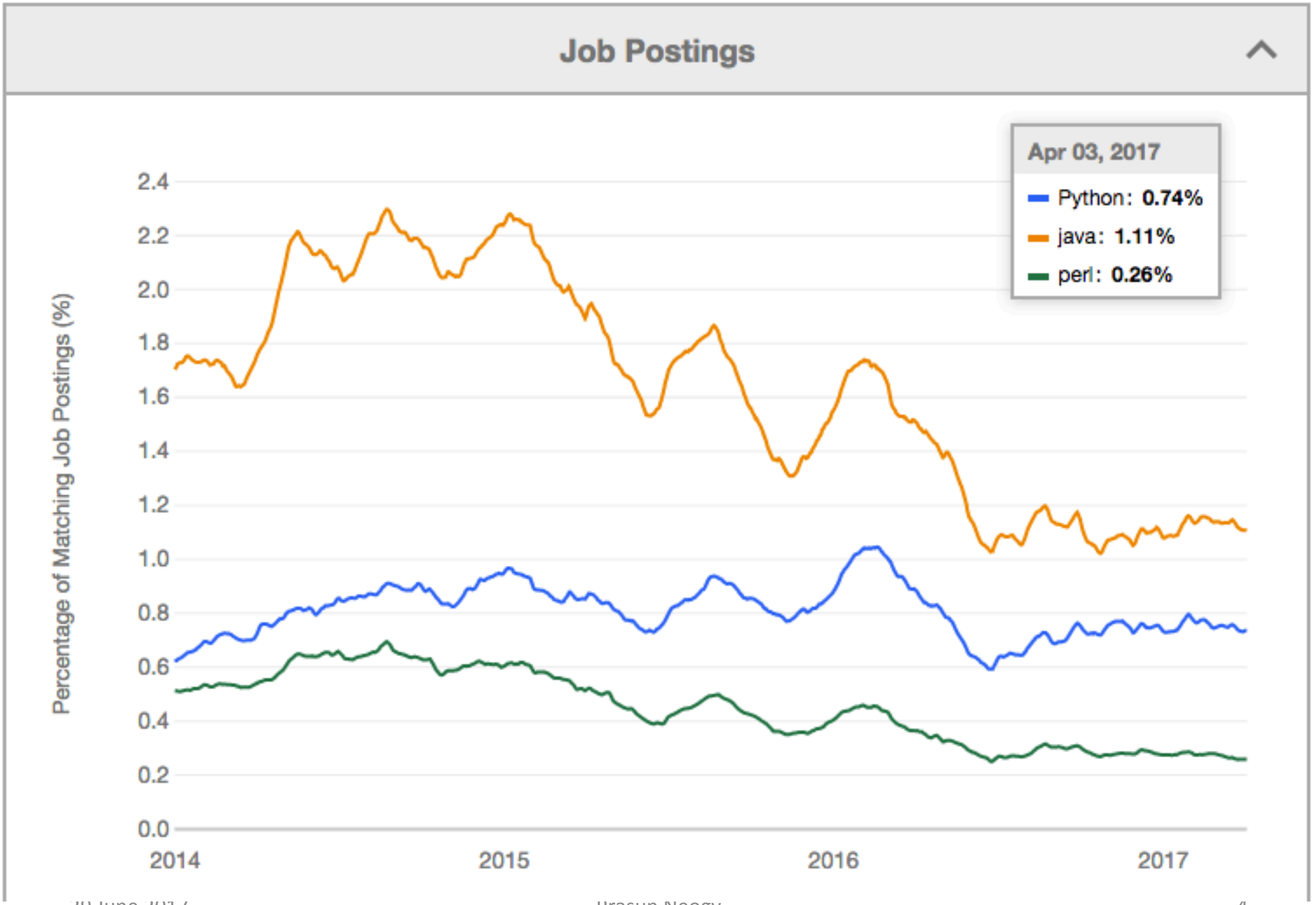
Python – S3

Contents

"Give a man a fish and you feed him for a day.
Teach a man to fish and you feed him for a lifetime."

– Chinese proverb


Job Postings
Apr 03, 2017
Python: 0.74%
java: 1.11%
perl: 0.26%
Percentage of Matching Job Postings (%)
2.4
2.2
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
2014
2015
2016
2017

Variables

Variables

- When you're using variables in Python, you need to adhere to a few rules and guidelines.
- Breaking some of these rules will cause errors; other guidelines just help you write code that's easier to read and understand.

- Be sure to keep the following variable rules in mind:

1. Variable names can contain only letters, numbers, and underscores. They can start with a letter or an underscore, but not with a number. For instance, you can call a variable *message_1* but not *1_message*.
2. Spaces are not allowed in variable names, but underscores can be used to separate words in variable names. For example, *greeting_message* works, but *greeting message* will cause errors.
3. Avoid using Python keywords and function names as variable names; that is, do not use words that Python has reserved for a particular programmatic purpose, such as the word print.

Variables

4. Variable names should be short but descriptive. For example, *name* is better than *n*, *student_name* is better than *s_n*, and *name_length* is better than *length_of_persons_name*.
5. Be careful when using the lowercase letter *l* and the uppercase letter *O* because they could be confused with the numbers *1* and *0*.

- It can take some practice to learn how to create good variable names, especially as your programs become more interesting and complicated.
- As you write more programs and start to read through other people's code, you'll get better at coming up with meaningful names.

- *The Python variables you're using at this point should be lowercase.*
- *You won't get errors if you use uppercase letters, but it's a good idea to avoid using them for now.*

Numbers

Integers

- You can add (+), subtract (-), multiply (*), and divide (/) integers in Python.
- Python supports the order of operations too, so you can use multiple operations in one expression.
- You can also use parentheses [actually my tip is : you should use parentheses] to modify the order of operations so Python can evaluate your expression in the order you specify.
- For example :

```
>>> 2 + 3*4
14
>>> (2 + 3) * 4
20
```

Numbers

Floats/Decimal

- Python calls any number with a decimal point a *float*.
- Consider this : For example, the following should yield zero, but it does not. The result is close to zero, but there are not enough bits to be precise here:

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.551115123125783e-17
```

- However, with decimals, the result can be dead-on:

```
>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
 Decimal('0.0')
```

Python's Core Data Types

- The following table gives you an idea of Python's built-in objects.

Object type	Example literals/creation
Numbers	1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()
Strings	'spam', "Bob's", b'a\x01c', u'sp\xc4m'
Lists	[1, [2, 'three'], 4.5], list(range(10))
Dictionaries	{'food': 'spam', 'taste': 'yum'}, dict(hours=10)
Tuples	(1, 'spam', 4, 'U'), tuple('spam'), namedtuple
Files	open('eggs.txt'), open(r'C:\ham.bin', 'wb')
Sets	set('abc'), {'a', 'b', 'c'}
Other core types	Booleans, types, None

Python's Built-in Numeric tools

- Python provides a set of tools for processing number objects:

- Expression operators :: + , - , * , / , >> , ** , & , etc.
- Built-in mathematical functions :: pow, abs, round, int, hex, bin, etc.
- Utility modules :: random, math, etc.

Conditional Execution

Conditional Execution

Boolean expressions

- A boolean expression is an expression that is either true or false.
- The following examples use the operator ==, which compares two operands and produces True if they are equal and False otherwise:

```
>>>5==5
True
>>>5==6
False
```

- True and False are special values that belong to the type bool; they are not strings.
- Other comparison operators are : !=, >, <, >=, <=

- Note : A common error is to use a single equal sign (=) instead of a double equal sign (==).
- Remember that = is an assignment operator and == is a comparison operator.

Conditional Execution

Logical operators

- There are three logical operators: and, or, and not.
- The semantics (meaning) of these operators is similar to their meaning in English.
- For example,

x > 0 and x < 10 is true only if x is greater than 0 *and* less than 10.

- Finally, the not operator negates a boolean expression, so

not (x > y) is true if x > y is false, that is, if x is less than or equal to y.

Conditional Execution

Conditional execution

- In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly.
- Conditional statements give us this ability.
- The simplest form is the if statement:

```
if x > 0 :
    print ('x is positive' )
```

- The boolean expression after the if statement is called the condition.
- We end the if statement with a colon character (:) and the line(s) after the if statement are indented.
- If the logical condition is true, then the indented statement gets executed. If the logical condition is false, the indented statement is skipped.

Conditional Execution

- if statements have the same structure as function definitions or for loops.
- The statement consists of a header line that ends with the colon character (:) followed by an indented block.
- Statements like this are called compound statements because they stretch across more than one line.
- There is no limit on the number of statements that can appear in the body, but there has to be at least one.
- If you enter an if statement in the Python interpreter, the prompt will change from three chevrons to three dots to indicate you are in the middle of a block of statements as shown below:

```
>>> x=3
>>> if  x<10 :
...     print ('Small')
...
Small
>>>
```

Conditional Execution

Alternative execution

- A second form of the if statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed.
- The syntax looks like this:

```
if x % 2 == 0 :
      print ('x is even' )
else :
      print ('x is odd' )
```

- If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays a message to that effect.
- If the condition is false, the second set of statements is executed.
- Since the condition must be true or false, exactly one of the alternatives will be executed.
- The alternatives are called branches, because they are branches in the flow of execution.

Conditional Execution

Chained conditionals

- Sometimes there are more than two possibilities and we need more than two branches.
- One way to express a computation like that is a chained conditional:

```
if x < y :
      print ('x is less than y' )
elif x > y :
      print ('x is greater than y' )
else :
      print ('x and y are equal' )
```

- elif is an abbreviation of “else if.”
- Again, exactly one branch will be executed.
- There is no limit on the number of elif statements.
- If there is an else clause, it has to be at the end, but there doesn’t have to be one.

Conditional Execution

Nested conditionals

- One conditional can also be nested within another.
- We could have written the trichotomy example like this:

```
if x == y :
    print ('x and y are equal' )
else :
    if x < y :
        print ('x is less than y' )
    else :
        print ('x is greater than y' )
```

- The outer conditional contains two branches.
- The first branch contains a simple statement.
- The second branch contains another if statement, which has two branches of its own.
- Those two branches are both simple statements, although they could have been conditional statements as well.

Conditional Execution

Catching exceptions using try and except

- Earlier we saw a code segment where we used the raw_input and int functions to read and parse an integer number entered by the user.

>>> speed = input(Enter value)

>>> int(speed)

- Now if instead of entering any numeric integer, if we had entered , say a string, the Python Interpreter will give an error message, like:

ValueError: invalid literal for int()

- However if this code is placed in a Python script and this error occurs, your script immediately stops in its tracks with a traceback.
- It does not execute the following statement.

- In order to handle such situations, there is a conditional execution structure built into Python to handle these types of expected and unexpected errors called “try / except”.

Conditional Execution

Catching exceptions using try and except

- The idea of try and except is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs.
- These extra statements (the except block) are ignored if there is no error.
- You can think of the try and except feature in Python as an “insurance policy” on a sequence of statements.
- Let’s see an example :

```
inp = input('Enter Fahrenheit Temperature:')
try:
    fahr = float(inp)
    cel=(fahr-32.0)*5.0/9.0
    print (cel )
except:
    print ('Please enter a number' )
```

Conditional Execution

- Python starts by executing the sequence of statements in the try block.
- If all goes well, it skips the except block and proceeds.
- If an exception occurs in the try block, Python jumps out of the try block and executes the sequence of statements in the except block.

- Handling an exception with a try statement is called catching an exception.
- In this example, the except clause prints an error message.
- In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

Conditional Execution

Indentation

- One of the most distinctive features of Python is its use of indentation to mark blocks of code. Consider the if-statement

```
if pwd == 'apple':
        print('Logging on ...')
else:
        print('Incorrect password.')
print('All done!')
```

- Consider another example :

```
# airfare.py
age = int(input('How old are you? '))
if age <= 2:
        print(' free')
elif 2 < age < 13:
        print(' child fare’)
else:
        print('adult fare')
```

Conditional Execution

- Let's see a complete example :

```
# Python program to check if the input year is a leap year or not

year = int(input("Enter a year: "))
if (year % 4) == 0:
    if (year % 100) == 0:
        if (year % 400) == 0:
            print("{0} is a leap year".format(year))
        else:
            print("{0} is not a leap year".format(year))
    else:
        print("{0} is a leap year".format(year))
else:
    print("{0} is not a leap year".format(year))
```

Conditional Execution

- Let's see another complete example :

```
# Python program to find the largest number among the three input numbers
# take three numbers from user
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))
num3 = float(input("Enter third number: "))

if (num1 > num2) and (num1 > num3):
    largest = num1
elif (num2 > num1) and (num2 > num3):
    largest = num2
else:
    largest = num3
print("The largest number is",largest)
```

Iterations

Iterations

The While Statement

- One form of iteration in Python is the while statement.
- Here is a simple program that counts down from five and then says "Blastoff!".

```
n=5
while  n > 0 :
      print (n )
      n = n - 1
print 'Blastoff!'
```

- You can almost read the while statement as if it were English.
- It means, "While n is greater than 0, display the value of n and then reduce the value of n by 1. When you get to 0, exit the while statement and display the word Blastoff!"

Iterations

The While Statement

- More formally, here is the flow of execution for a while statement:

1. Evaluate the condition, yielding True or False.
2. If the condition is false, exit the while statement and continue execution at the next statement.
3. If the condition is true, execute the body and then go back to step 1.

- This type of flow is called a loop because the third step loops back around to the top.
- Each time we execute the body of the loop, we call it an iteration.
- For the above loop, we would say, “It had five iterations” which means that the body of the loop was executed five times.

break Statement

The While Statement

- The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates.
- We call the variable that changes each time the loop executes and controls when the loop finishes the iteration variable.
- If there is no iteration variable, the loop will repeat forever, resulting in an infinite loop.
- You can however write an infinite loop, and then come out of the loop using a ‘break’ statement, like:

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print (line)
print ('Done!’)
```

continue Statement

The Continue Statement

- Sometimes you are in an iteration of a loop and want to finish the current iteration and immediately jump to the next iteration.
- In that case you can use the continue statement to skip to the next iteration without finishing the body of the loop for the current iteration.
- An example :

```
while True:
    line = input('> ')
    if line[0] == '#' :
        continue
    if line == 'done':
        break
    print (line)
print ('Done!')
```

Definite Loops

Definite loops using for

- Sometimes we want to loop through a set of things such as a list of words, the lines in a file or a list of numbers.
- When we have a list of things to loop through, we can construct a *definite* loop using a for statement.
- We call the while statement an *indefinite* loop because it simply loops until some condition becomes False whereas the for loop is looping through a known set of items so it runs through as many iterations as there are items in the set.
- The syntax of a for loop is similar to the while loop in that there is a for statement and a loop body:

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
    print ('Happy New Year:', friend )
print ('Done!')
```

Definite loops

Definite loops using for

- In Python terms, the variable friends is a list of three strings and the for loop goes through the list and executes the body once for each of the three strings in the list resulting in this output:

Happy New Year: Joseph

Happy New Year: Glenn

Happy New Year: Sally

Done!

- In particular, friend is the iteration variable for the for loop.
- The variable friend changes for each iteration of the loop and controls when the for loop completes.
- The iteration variable steps successively through the three strings stored in the friends variable.

Loop patterns

Loop patterns

- Often we use a for or while loop to go through a list of items or the contents of a file and we are looking for something such as the largest or smallest value of the data we scan through.
- These loops are generally constructed by:
  - Initializing one or more variables before the loop starts.
  - Performing some computation on each item in the loop body, possibly changing the variables in the body of the loop.
  - Looking at the resulting variables when the loop completes.

- We will use a list of numbers to demonstrate the concepts and construction of these loop patterns.

Iterations

Counting and summing loops

- For example, to count the number of items in a list, we would write the following for loop:

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    count = count + 1
print ('Count: ', count )
```

- We set the variable count to zero before the loop starts, then we write a for loop to run through the list of numbers. Our iteration variable is named itervar and while we do not use itervar in the loop, it does control the loop and cause the loop body to be executed once for each of the values in the list.
- In the body of the loop, we add one to the current value of count for each of the values in the list. While the loop is executing, the value of count is the number of values we have seen “so far”.
- Once the loop completes, the value of count is the total number of items.

Iterations

Counting and summing loops

- Another similar loop that computes the total of a set of numbers is as follows:

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar
print ('Total: ', total)
```

- Note the difference between this code and the previous one. Here we are using the iteration variable.

Iterations

Maximum and minimum loops

- To find the largest value in a list or sequence, consider the following loop:

```
largest = None
print ('Before:', largest)
for itervar in [3, 41, 12, 9, 74, 15]:
    if largest is None or itervar > largest :
        largest = itervar
    print ('Loop:', itervar, largest)
print ('Largest:', largest)
```

- When the program executes, the output is as follows:

```
Before: None
Loop: 3 3
Loop: 41 41
Loop: 12 41
Loop: 9 41
Loop: 74 74
Loop: 15 74
Largest: 74
```

Iterations

Maximum and minimum loops

- To compute the smallest number, the code is very similar with one small change:

```
smallest = None
print ('Before:', smallest)
for itervar in [3, 41, 12, 9, 74, 15]:
    if smallest is None or itervar < smallest:
        smallest = itervar
    print ('Loop:', itervar, smallest )
print ('Smallest:', smallest )
```

Iterations -- Example

- Let's see another complete Python program :

```
#!/usr/bin/env  python3
N = 10
sum = 0
count = 0
while count < N:
    number = float(input("Enter a number : "))
    sum = sum + number
    count = count + 1
average = float(sum)/N
print("N = %d , Sum = %f" % (N, sum))
print("Average = %f" % average)
```

Iterations - Examples

Let us consider an example :

```
# Python program to find the factorial of a number provided by the user.
# take input from the user
num = int(input("Enter a number: "))
factorial = 1
# check if the number is negative, positive or zero
if num < 0:
    print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
    print("The factorial of 0 is 1")
else:
    for i in range(1,num + 1):
        factorial = factorial*i
    print("The factorial of",num,"is",factorial)
```

Iterations – Examples

```
## Calculating Compound Interest
principal = 1000.0   # starting principal
rate = .08           # interest rate

print ("Initial Deposit Amount : ", principal)
print ("Rate of Interest is : ", rate )
print ("Year %21s" % "Amount on deposit")

for year in range( 1, 11 ):
  amount = principal * ( 1.0 + rate ) ** year
  print ("%4d%21.2f" % ( year, amount ))
```

Note : Formatting the print statement “year %21s” % “Amount on deposit” → implies that the last string will be aligned from right taking max 21 spaces!!!

We will discuss formatting later as well.

Iterations – Examples

- The output of the previous program is :

```
Initial Deposit Amount :  1000.0
Rate of Interest is :  0.08
Year     Amount on deposit
   1              1080.00
   2              1166.40
   3              1259.71
   4              1360.49
   5              1469.33
   6              1586.87
   7              1713.82
   8              1850.93
   9              1999.00
  10              2158.92
```

Iterations – Examples

```
import random

# set the initial values from 1 to 5
the_number = random.randrange(5) + 1
guess = int(input("Take a guess: "))
tries = 1

# guessing loop
while (guess != the_number):
        if (guess > the_number):
                print (”Try Lower...")
        else:
                print (”Try Higher...")

        guess = int(input("Take a guess: "))
        tries += 1

print ("You guessed it!  The number was", the_number)
print ("And it only took you", tries, "tries!\n")
```

Iterations – Examples

```
total = 0.0
count = 0
while count < 3:
        number=float(input("Enter a number: "))
        count = count + 1
        total = total + number

average = total / 3
print ("The average is " + str(average))
```

- Find out the output of the following :

```
print (7 > 10)
print (4 < 16)
print (4 == 4)
print (4 <= 4)
print (4 >= 4)
print (4 != 4)
```

Iterations – Examples

Sentinel controlled while loop :

```
# initialization phase
totalScore = 0 # sum of scores
numberScores = 0 # number of scores entered
# processing phase
score = input( "Enter score, (Enter -9 to end): " ) # get one score
score = int( score ) # convert string to an integer

while score != -9: # -9 is used as a sentinel ( a lookout or sentry value )
        totalScore = totalScore + score
        numberScores = numberScores + 1
        score = input( "Enter score, (Enter -9 to end): " )
        score = int( score )
# termination phase
if numberScores != 0: # division by zero would be a run-time error
        average = float( totalScore ) / numberScores
        print ("Class average is", average)
else:
        print ("No scores were entered")
```

Iterations – Examples

Examples with 'break and 'continue' :

```
for c in range (0,6):
        if c == 3:
                break
        print (c)
```

- The output is :

0
1
2

```
for c in range (0,6):
        if c == 3:
                continue
        print (c)
```

- The output is :

0
1
2
4
5

Iterations – Examples

```
# Given a DNA sequence, find the percentages of nucleotides in the sequence
# that are A's, C's, T's, and G's.
dnaSequence = "atcgatgagagctagcgata”
aCount = 0
cCount = 0
tCount = 0
gCount = 0
for c in dnaSequence:
        if c == 'a':
                aCount = aCount + 1
        elif c == 'c':
                cCount = cCount + 1
        elif c == 't':
                tCount = tCount + 1
        elif c == 'g':
                gCount = gCount + 1
sequenceLength = len(dnaSequence)
print ("Percentage of A's in sequence:", (float(aCount) / sequenceLength) * 100 )
print ("Percentage of C's in sequence:", (float(cCount) / sequenceLength) * 100 )
print ("Percentage of T's in sequence:", (float(tCount) / sequenceLength) * 100 )
print ("Percentage of G's in sequence:", (float(gCount) / sequenceLength) * 100)
```

Formatting in Python

- Basic formatting:

Old -- '%s %s' % ('one', 'two')

New -- '{} {}'.format('one', 'two')

Output -- *one two*

Old -- '%d %d' % (1, 2)

New -- '{} {}'.format(1, 2)

Output -- *1 2*

You can also do this in new-style formatting :

New -- '{1} {0}'.format('one', 'two')

Output -- *two one*

Formatting in Python

- Padding and aligning strings :

Align right:

Old -- '%10s' % ('test',)

New -- '{:>10}'.format('test')

Output

Align left:

Old -- '%-10s' % ('test',)

New -- '{:10}'.format('test')

Output

Formatting in Python

- Padding and aligning strings :

You can choose the padding character:

New -- '{:_<10}'.format('test')

Output

```
test______
```

And also center align values:

New -- '{:^10}'.format('test')

Output

```
   test   
```

Formatting in Python

- Numbers :

Integers:

Old -- '%d' % (42,)

New -- '{:d}'.format(42)

Output -- *42*

Floats:

Old -- '%f' % (3.141592653589793,)

New -- '{:f}'.format(3.141592653589793)

Output -- *3.141593*

Formatting in Python

- Padding Numbers :

Similar to strings numbers can also be constrained to a specific width.

Old -- '%4d' % (42,)

New -- '{:4d}'.format(42)

Output

```
  42
```

Again similar to truncating strings the precision for floating point numbers limits the number of positions after the decimal point.

For floating points the padding value represents the length of the complete output. In the example below we want our output to have at least 6 characters with 2 after the decimal point.

Old -- '%06.2f' % (3.141592653589793,)

New -- '{:06.2f}'.format(3.141592653589793)

Output

```
003.14
```

Formatting in Python

- Datetime :

from datetime import datetime

- Now we can format as :

New -- '{:%Y-%m-%d %H:%M}'.format(datetime(2001, 2, 3, 4, 5))

Output

```
2001-02-03 04:05
```

Reference

- Python for Informatics – C. Severance
- Think Python – A. B. Downey - [O'Reilly]
- Python Crash Course – Eric Matthes [No starch Press]
- A Byte of Python – Swaroop C H
- Introducing Python – Bill Lubanovic [O'Reilly]

- Learning Python – 5th Ed – Mark Lutz - [O'Reilly]

End of Presentation

Python – S3