Cours d'introduction à Python

Hugues Van Assel

November 15, 2022

Overview

Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages (for example, Pascal or C); parentheses (()) can be used for grouping. For example:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # division always returns a floating point number
1.6
```

Numbers

The integer numbers (e.g. 2, 4, 20) have type int, the ones with a fractional part (e.g. 5.0, 1.6) have type float.
Division (/) always returns a float. To do floor division and get an integer result (discarding any fractional result) you can use the // operator; to calculate the remainder you can use %:

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3  # floor division discards the fractional part
5
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # floored quotient * divisor + remainder
17
```

Numbers

Use the ** operator to calculate powers

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

The equal sign (=) is used to assign a value to a variable.

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

Numbers

0perators with mixed type operands convert the integer operand to floating point.

```
>>> 4 * 3.75 - 1
14.0
```

Python also has built-in support for complex numbers, and uses the j or J suffix to indicate the imaginary part (e.g. 3+5j).

Exercice

Écrire un programme qui, à partir de la saisie d'un rayon et d'une hauteur, calcule le volume d'un cône droit.

```
# Import ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
from math import pi

# Programme principal ========================================================
rayon = float(input("Rayon du cône (m) :"))
hauteur = float(input("Hauteur du cône (m) :"))

volume = 
print("Volume du cône =", volume, "m3")
```

Strings

Python's strings can be enclosed in single quotes ('...') or double quotes ("...") with the same result 2. \ can be used to escape quotes:

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

The built-in function len() returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Strings

Strings can be concatenated (glued together) with the + operator, and repeated with *

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Strings can be indexed (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

Strings

Indices may also be negative numbers, to start counting from the right:

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

Note that since -0 is the same as 0, negative indices start from -1. In addition to indexing, slicing is also supported. While indexing is used to obtain individual characters, slicing allows you to obtain substring:

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

Strings

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>> word[:2]   # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]   # characters from position 4 (included) to the end
'on'
>>> word[-2:]  # characters from the second-last (included) to the end
'on'
```

Note how the start is always included, and the end always excluded. This makes sure that s[:i] + s[i:] is always equal to s:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

The extended slicing notation string[start:stop:step] uses three arguments start, stop, and step

Exercice

Exercise

Write a command to reverse a string.

List

Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type:

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

List

Like strings (and all other built-in sequence types), lists can be indexed and sliced:

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

List

Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content:

```
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a shallow copy of the list:

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

List

Lists also support operations like concatenation:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

You can also add new items at the end of the list, by using the append():

```
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

List

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

Exercise

Exercise

Write a program to print a specified list after removing the 0th, 4th and 5th elements.

List

It is possible to nest lists (create lists containing other lists), for example:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

Towards Programming

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
```

- The first line contains a multiple assignment: the variables a and b simultaneously get the new values 0 and 1
- The while loop executes as long as the condition (here: a ¡ 10) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false.
- The body of the loop is indented: indentation is Python's way of grouping statements.

Exercise

Exercise

Écrire un programme qui approxime e, pour n assez grand, en utilisant la formule :

$$e = \sum_{i=0}^{n} \frac{1}{i!}$$

Exercise

Ecrire un programme qui prend un entier positif en entrée et annonce combien de fois de suite cet entier est divisible par 2.

Exercise

On dispose d'une feuille de papier d'épaisseur 0,1 mm. Combien de fois doit-on la plier au minimum pour que l'épaisseur dépasse la hauteur de la tour Eiffel 324 m.

Control flow

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more elif parts, and the else part is optional. The keyword 'elif' is short for 'else if', and is useful to avoid excessive indentation.

Exercise

Exercise

L'utilisateur donne un entier positif n et le programme affiche PAIR s'il est divisible par 2, IMPAIR sinon.

Exercise

Ecrire programme qui reçoit une liste d'entiers et qui renvoie le minimum, le maximum et la moyenne de cette liste.

Control flow

The for statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Control flow

If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy. It generates arithmetic progressions:

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

Exercise

Exercise

Produisez puis affichez la liste des valeurs 4x+z où x et z varient chacun entre 0 inclus et 5 exclu.

Exercise

Ecrivez une fonction qui prend en entrée une liste de nombres et renvoie la sous-liste de ses éléments compris entre -1 et 1.

Control flow

To iterate over the indices of a sequence, you can combine range() and len() as follows:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

or use enumerate.

Control flow

The break statement breaks out of the innermost enclosing for or while loop. Loop statements may have an else clause; it is executed when the loop terminates through exhaustion of the iterable (with for) or when the condition becomes false (with while), but not when the loop is terminated by a break statement.

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

Control flow

The continue statement continues with the next iteration of the loop:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found an odd number", num)
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

Function

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword def introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

Function

```
>>> def fib2(n):  # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

- The return statement returns with a value from a function. return without an expression argument returns None. Falling off the end of a function also returns None.
- The statement result.append(a) calls a method of the list object result. A method is a function that 'belongs' to an object and is named obj.methodname, where obj is some object (this may be an expression), and methodname is the name of a method that is defined by the object's type.

Exercise

Given a number n, write an efficient function to print all prime factors of n. For example, if the input number is 12, then output should be “2 2 3”. And if the input number is 315, then output should be “3 3 5 7”.

Exercise

```
def primeFactors(n):

    # Print the number of two\'s that divide n
    while n % 2 == 0:
        print 2,
        n = n / 2

    # n must be odd at this point
    # so a skip of 2 ( i = i + 2) can be used
    for i in range(3,int(math.sqrt(n))+1,2):

        # while i divides n , print i ad divide n
        while n % i== 0:
            print i,
            n = n / i

    # Condition if n is a prime
    # number greater than 2
    if n > 2:
        print n
```

List methods

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting a position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

Methods like insert, remove or sort only modify the list have no return value printed – This is a design principle for all mutable data structures. Not all data can be sorted or compared. For instance, [None, 'hello', 10] doesn't sort because integers can't be compared to strings.

Exercise

Given a list, return a list of sum of digits of each element.

```
The original list is : [12, 67, 98, 34]
List Integer Summation : [3, 13, 17, 7]
```

Exercise

Given a list, return a list of sum of digits of each element.

```
The original list is : [12, 67, 98, 34]
List Integer Summation : [3, 13, 17, 7]
```

```
res = []
for ele in test_list:
    sum = 0
    for digit in str(ele):
        sum += int(digit)
    res.append(sum)
```

Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved ("last-in, first-out"). To add an item to the top of the stack, use append(). To retrieve an item from the top of the stack, use pop() without an explicit index. For example:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

Exercise

Given a string str, we need to print reverse of individual words.

Examples:

```
Input : Hello World
Output : olleH dlroW

Input :  Geeks for Geeks
Output : skeeG rof skeeG
```

Exercise

```
# reverses individual words of a string
def reverserWords(string):
    st = list()

    # Traverse given string and push all characters
    # to stack until we see a space.
    for i in range(len(string)):
        if string[i] != " ":
            st.append(string[i])

        # When we see a space, we print
        # contents of stack.
        else:
            while len(st) > 0:
                print(st[-1], end= "")
                st.pop()
            print(end = " ")

    # Since there may not be space after
    # last word.
    while len(st) > 0:
        print(st[-1], end = "")
        st.pop()
```

Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved ("first-in, first-out"); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use collections.deque which was designed to have fast appends and pops from both ends. For example:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

Queues

Given a queue. The task is to reverse the queue using another another empty queue.

Examples:

```
Input: queue[] = {1, 2, 3, 4, 5}
Output: 5 4 3 2 1

Input: queue[] = {10, 20, 30, 40}
Output: 40 30 20 10
```

Queues

```
from collections import deque

# Function to return the reversed queue
def reverse(q):

    # Size of ueue
    s = len(q)

    # Second queue
    ans = deque()

    for i in range(s):

        # Get the last element to the
        # front of queue
        for j in range(s - 1):
            x = q.popleft()
            q.appendleft(x)

        # Get the last element and
        # add it to the new queue
        ans.appendleft(q.popleft())
    return ans
```

Tuple

A tuple consists of a number of values separated by commas, for instance:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.

Exercise Tuple

Given a Tuple List, perform sort on basis of total digits in tuple.

Examples:

Input *: test_list = [(3, 4, 6, 723), (1, 2), (134, 234, 34)]*
Output *: [(1, 2), (3, 4, 6, 723), (134, 234, 34)]*
Explanation *: 2 < 6 < 8, sorted by increasing total digits.*

Input *: test_list = [(1, 2), (134, 234, 34)]*
Output *: [(1, 2), (134, 234, 34)]*
Explanation *: 2 < 8, sorted by increasing total digits.*

Exercise Tuple

```
def count_digs(tup):

    # gets total digits in tuples
    return sum([len(str(ele)) for ele in tup ])

# initializing list
test_list = [(3, 4, 6, 723), (1, 2), (12345,), (134, 234, 34)]

# printing original list
print("The original list is : " + str(test_list))

# performing sort
test_list.sort(key = count_digs)

# printing result
print("Sorted tuples : " + str(test_list))
```

Sets

Python also includes a data type for sets. A set is an unordered collection with no duplicate elements.

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Exercise

Exercise

Write a function that returns the difference between two lists (list of elements that are not in the union of the two input lists).

Solution difference of two lists

```
# Python code t get difference of two lists
# Using set()
def Diff(li1, li2):
    return list(set(li1) - set(li2)) + list(set(li2) - set(li1))

# Not using set()
def Diff(li1, li2):
    li_dif = [i for i in li1 + li2 if i not in li1 or i not in li2]
    return li_dif
```

Dictionaries

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

Dictionaries

The dict() constructor builds dictionaries directly from sequences of key-value pairs:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Exercise dictionaries

Given a dictionary with values list, extract key whose value has most unique values.

Input *: test_dict = {"Gfg" : [5, 7, 9, 4, 0], "is" : [6, 7, 4, 3, 3], "Best" : [9, 9, 6, 5, 5]}*
Output *: "Gfg"*
Explanation *: "Gfg" having max unique elements i.e 5.*

Input *: test_dict = {"Gfg" : [5, 7, 7, 7, 7], "is" : [6, 7, 7, 7], "Best" : [9, 9, 6, 5, 5]}*
Output *: "Best"*
Explanation *: 3 (max) unique elements, 9, 6, 5 of "Best".*

Solution Exercise dictionaries

```
# initializing dictionary
test_dict = {"Gfg" : [5, 7, 5, 4, 5],
             "is" : [6, 7, 4, 3, 3],
             "Best" : [9, 9, 6, 5, 5]}

# printing original dictionary
print("The original dictionary is : " + str(test_dict))

max_val = 0
max_key = None
for sub in test_dict:

    # test for length using len()
    # converted to set for duplicates removal
    if len(set(test_dict[sub])) > max_val:
        max_val = len(set(test_dict[sub]))
        max_key = sub

# printing result
print("Key with maximum unique values : " + str(max_key))
```

Mutable / Immutable

Objects of built-in type that are mutable are:

- Lists
- Sets
- Dictionaries

Objects of built-in type that are immutable are:

- Numbers
- Strings
- Tuples

Looping techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate() function:

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

Looping techniques

To loop over two or more sequences at the same time, the entries can be paired with the zip() function:

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

To loop over a sequence in sorted order, use the sorted() function which returns a new sorted list while leaving the source unaltered:

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for i in sorted(basket):
...     print(i)
...
apple
apple
banana
orange
orange
pear
```

Looping techniques

Using set() on a sequence eliminates duplicate elements. The use of sorted() in combination with set() over a sequence is an idiomatic way to loop over unique elements of the sequence in sorted order:

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

Exercise

Given a string and a number N, we need to mirror the characters from the N-th position up to the length of the string in alphabetical order. In mirror operation, we change 'a' to 'z', 'b' to 'y', and so on.
Examples:

```
Input : N = 3
        paradox
Output : paizwlc
We mirror characters from position 3 to end.

Input : N = 6
        pneumonia
Output : pneumlmrz
```

Solution exercise

```
def mirrorChars(input,k):

    # create dictionary
    original = 'abcdefghijklmnopqrstuvwxyz'
    reverse = 'zyxwvutsrqponmlkjihgfedcba'
    dictChars = dict(zip(original,reverse))

    # separate out string after length k to change
    # characters in mirror
    prefix = input[0:k-1]
    suffix = input[k-1:]
    mirror = ''

    # change into mirror
    for i in range(0,len(suffix)):
        mirror = mirror + dictChars[suffix[i]]

    # concat prefix and mirrored part
    print (prefix+mirror)
```

Exercise

There are some natural number whose all permutation is greater than or equal to that number eg. 123, whose all the permutation (123, 231, 321) are greater than or equal to 123.
Given a natural number **n**, the task is to count all such number from 1 to n.

Examples:

Input : *n = 15.*
Output : *14*
Explanation:
1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12,
13, 14, 15 are the numbers whose all
permutation is greater than the number
itself. So, output 14.

Input : *n = 100.*
Output : *54*

Exercise

```
def countNumber(n):
    result = 0

    # Pushing 1 to 9 because all number
    # from 1 to 9 have this property.
    s = []
    for i in range(1, 10):

        if (i <= n):
            s.append(i)
            result += 1

        # take a number from stack and add
        # a digit smaller than or equal to last digit
        # of it.
        while len(s) != 0:
            tp = s[-1]
            s.pop()
            for j in range(tp % 10, 10):
                x = tp * 10 + j
                if (x <= n):
                    s.append(x)
                    result += 1

    return result
```

Files

open() returns a file object, and is most commonly used with two arguments: open(filename, mode).

```
>>> f = open('workfile', 'w')
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. Mode can be 'r' when the file will only be read, 'w' for only writing (an existing file with the same name will be erased), and 'a' opens the file for appending; any data written to the file is automatically added to the end. 'r+' opens the file for both reading and writing. The mode argument is optional; 'r' will be assumed if it's omitted.

Files

It is good practice to use the with keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point.

```
>>> with open('workfile') as f:
...     read_data = f.read()

>>> # We can check that the file has been automatically closed.
>>> f.closed
True
```

If you're not using the with keyword, then you should call f.close() to close the file and immediately free up any system resources used by it.

Files

After a file object is closed, either by a with statement or by calling f.close(), attempts to use the file object will automatically fail.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

Files

To read a file's contents, call f.read(size), which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode). size is an optional numeric argument. When size is omitted or negative, the entire contents of the file will be read and returned. Otherwise, at most size characters (in text mode) or size bytes (in binary mode) are read and returned. If the end of the file has been reached, f.read() will return an empty string ('').

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

Files

f.readline() reads a single line from the file; a newline character (\ n) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if f.readline() returns an empty string, the end of the file has been reached, while a blank line is represented by '\ n', a string containing only a single newline.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

Files

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

Exercise

Ex 1

Write a python program to find the longest words.

Ex 2

Write a Python program to count the frequency of words in a file.

Ex 3

Write a Python program to count the number of nonempty lines in the file.

Solutions

Write a python program to find the longest words.

```
def longest_word(filename):
    with open(filename, 'r') as infile:
                words = infile.read().split()
    max_len = len(max(words, key=len))
    return [word for word in words if len(word) == max_len]
```

Write a Python program to count the frequency of words in a file.

```
from collections import Counter
def word_count(fname):
        with open(fname) as f:
                return Counter(f.read().split())
```

Write a Python program to count the number of nonempty lines in the file.

```
file = open("sample.txt", "r")
line_count = 0
for line in file:
    if line != "\n":
        line_count += 1
file.close()
```

Files

f.write(string) writes the contents of string to the file, returning the number of characters written.

```
>>> f.write('This is a test\n')
15
```

Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) – before writing them:

```
>>> value = ('the answer', 42)
>>> s = str(value)  # convert the tuple to string
>>> f.write(s)
18
```

Files

To change the file object's position, use f.seek(offset, whence). The position is computed from adding offset to a reference point; the reference point is selected by the whence argument. A whence value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. whence can be omitted and defaults to 0, using the beginning of the file as the reference point.

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

Exercise

Ex

Write a Python program to reverse the lines and then the words of poem.txt.

Exercise

```
with open('reversed_words.txt', 'w') as writer1:
    with open('reversed_line.txt', 'w') as writer2:
        with open('poem.txt', 'r') as reader:
            # Note: readlines doesn't trim the line endings
            line = reader.readlines()

            # Alternatively you could use
            # writer.writelines(reversed(line))
            for rever in reversed(line):
                writer1.write(rever[::-1])
                writer2.write(rever)
```

Files

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the read() method only returns strings, which will have to be passed to a function like int(), which takes a string like '123' and returns its numeric value 123. When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.
Python allows you to use the popular data interchange format called JSON (JavaScript Object Notation). The standard module called json can take Python data hierarchies, and convert them to string representations; this process is called serializing. Reconstructing the data from the string representation is called deserializing.

Files

If you have an object x, you can view its JSON string representation with a simple line of code:

```
>>> import json
>>> x = [1, 'simple', 'list']
>>> json.dumps(x)
'[1, "simple", "list"]'
```

Files

Another variant of the dumps() function, called dump(), simply serializes the object to a text file. So if f is a text file object opened for writing, we can do this:

```
json.dump(x, f)
```

To decode the object again, if f is a text file object which has been opened for reading:

```
x = json.load(f)
```

This simple serialization technique can handle lists and dictionaries, but serializing arbitrary class instances in JSON requires a bit of extra effort. The reference for the json module contains an explanation of this.

Functions

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

This function can be called in several ways:

- giving only the mandatory argument: ask ok('Do you really want to quit?')
- giving one of the optional arguments: ask ok('OK to overwrite the file?', 2)
- or even giving all arguments: ask ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')

Functions

The default values are evaluated at the point of function definition in the defining scope, so that

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

Functions

Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

Functions

Functions can also be called using keyword arguments of the form kwarg=value. For instance, the following function:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

accepts one required argument (voltage) and three optional arguments (state, action, and type). This function can be called in any of the following ways:

```
parrot(1000)                                          # 1 positional argument
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')         # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

but all the following calls would be invalid:

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
parrot(110, voltage=220)     # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

Functions

When a final formal parameter of the form **name is present, it receives a dictionary (see Mapping Types — dict) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form *name (described in the next subsection) which receives a tuple containing the positional arguments beyond the formal parameter list. (*name must occur before **name.) For example, if we define a function like this:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

Functions

It could be called like this:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

and of course it would print:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Functions

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in range() function expects separate start and stop arguments. If they are not available separately, write the function call with the *-operator to unpack the arguments out of a list or tuple:

```
>>> list(range(3, 6))            # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))            # call with arguments unpacked from a list
[3, 4, 5]
```

Functions

In the same fashion, dictionaries can deliver keyword arguments with the **-operator:

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin
```

Functions

Small anonymous functions can be created with the lambda keyword. This function returns the sum of its two arguments: lambda a, b: a+b. Like nested function definitions, lambda functions can reference variables from the containing scope:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

Functions

The above example uses a lambda expression to return a function.
Another use is to pass a small function as an argument:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

Exercise

Ex 1

Write a Python program to sort a list of dictionaries using Lambda.

Ex 2

Write a Python program to find the list with maximum and minimum length using lambda.

Ex 3

Write a Python program to remove all elements from a given list present in another list using lambda.

Solutions

Write a Python program to sort a list of dictionaries using Lambda:

```
models = [{'make':'Nokia', 'model':216, 'color':'Black'}, {'make':'Mi Max', 'model':'2', 'color':'G
print("Original list of dictionaries :")
print(models)
sorted_models = sorted(models, key = lambda x: x['color'])
```

Write a Python program to find the list with maximum and minimum length using lambda:

```
def max_length_list(input_list):
    max_length = max(len(x) for x in input_list )
    max_list = max(input_list, key = lambda i: len(i))
    return(max_length, max_list)

def min_length_list(input_list):
    min_length = min(len(x) for x in input_list )
    min_list = min(input_list, key = lambda i: len(i))
    return(min_length, min_list)
```

Write a Python program to remove all elements from a given list present in another list using lambda.

```
def index_on_inner_list(list1, list2):
    result = list(filter(lambda x: x not in list2, list1))
    return result
```

Practice

Ex 1

Write a function that generates one of 3 numbers according to given probabilities

Ex 2

Given two polynomials represented by two arrays, write a function that multiplies given two polynomials.

Ex 3

Given n dice each with m faces, numbered from 1 to m, find the number of ways to get sum X. (X is the summation of values on each face when all the dice are thrown.)

Solution EX 1

```
def random(x, y, z, px, py, pz):

    # Generate a number from 1 to 100
    r = random.randint(1, 100)

    #  r is smaller than px with probability px/100
    if (r <= px):
        return x

    # r is greater than px and smaller than
    # or equal to px+py with probability py/100
    if (r <= (px+py)):
        return y

    # r is greater than px+py and smaller than
    # or equal to 100 with probability pz/100
    else:
        return z
```

Solution EX 2

```
def multiply(A, B, m, n):

    prod = [0] * (m + n - 1);

    # Multiply two polynomials term by term

    # Take ever term of first polynomial
    for i in range(m):

        # Multiply the current term of first
        # polynomial with every term of
        # second polynomial.
        for j in range(n):
            prod[i + j] += A[i] * B[j];

    return prod;
```

Solution EX 3

```
def findWays(m,n,x):
    # Create a table to store results of subproblems. One extra
    # row and column are used for simpilicity (Number of dice
    # is directly used as row index and sum is directly used
    # as column index). The entries in 0th row and 0th column
    # are never used.
    table=[[0]*(x+1) for i in range(n+1)] #Initialize all entries as 0

    for j in range(1,min(m+1,x+1)): #Table entries for only one dice
        table[1][j]=1

    # Fill rest of the entries in table using recursive relation
    # i: number of dice, j: sum
    for i in range(2,n+1):
        for j in range(1,x+1):
            for k in range(1,min(m+1,j)):
                table[i][j]+=table[i-1][j-k]

    #print(dt)
    # Uncomment above line to see content of table

    return table[-1][-1]
```

Numpy

```python
import numpy as np # c'est la convention que tout le monde adopte

a = np.array([1, 2, 3, 4]) # on affecte un tableau numpy "à la main"
# on verra qu'il y a des techniques automatiques très utiles

b = 2.5 * a  # multiplication globale membre à membre
print(b) # b est devenu un tableau de flottants

c = np.array([5, 6, 7, 8]) # des entiers

d = b+c
print(d) # d a été "transtypé" en flottants

# un exemple un peu plus "brutal" qui montre que ça ne rame pas trop
posi = np.random.rand(10000000,2) # 1e7 lignes, 2 colonnes de positions x,y
x, y = posi[:,0], posi[:,1] # les sections de tableaux fonctionnent (slices)
#%precision 3 # commenté pour lancer tout le script
print(posi) # on voit la notation en double []
print(x)    # c'est un vecteur

x0, y0 = 0.5, 0.5

dist =  np.sqrt((x-x0)**2 + (y-y0)**2) # est un (gros) vecteur
print(dist.argmin()) # indice du min de ce tableau
print(dist.min()) # le min

# on peut ainsi coder de manière "vectorisée" : élégant et efficace
# d'une manière très proche des langages historiques comme Scilab, Matlab, etc
```

matplotlib

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-10, 10, 200)
y = np.sin(np.pi*x) / (np.pi*x)

plt.plot(x, y)
plt.show()
```

In a more pythonic way:

```
def monPlot(x, y, title='', color='red', linestyle='dashed', linewidth=2):
    """Un tracé y=f(x) un peu customisé."""
    fig = plt.figure() # une figure, vide pour l'instant
    axes = fig.add_subplot(111) # il y aura un graphe dedans
    axes.plot(x, y, color=color, linestyle=linestyle, linewidth=linewidth)
    axes.set_title(title)
    axes.grid() # on ajoute la grille derrière
    plt.show()

def f(a,b,c,d):
    """ Définir et plotter un trinôme"""
    x = np.linspace(-10,10,20)
    y = a*(x**3) + b*(x**2) + c*x + d
    title = '$f(x) = (%s)x^3 + (%s)x^2 + (%s)x + (%s)$' % (a,b,c,d) # Rq dessous
    monPlot(x,y, title=title)
```

Practice

Ex 4

Given an array of size n, generate and print all possible combinations of r elements in array. For example, if input array is $\{1, 2, 3, 4\}$ and r is 2, then output should be $\{1, 2\}$, $\{1, 3\}$, $\{1, 4\}$, $\{2, 3\}$, $\{2, 4\}$ and $\{3, 4\}$.

Ex 5

Write a function that takes two parameters n and k and returns the value of Binomial Coefficient C(n, k).

Ex 6

Given a number n, find the smallest number that has same set of digits as n and is greater than n. If n is the greatest possible number with its set of digits, then print "not possible".

Solution EX 4

```
def combinationUtil(arr, data, start,
                    end, index, r):

    # Current combination is ready
    # to be printed, print it
    if (index == r):
        for j in range(r):
            print(data[j], end = " ");
        print();
        return;

    # replace index with all
    # possible elements. The
    # condition "end-i+1 >=
    # r-index" makes sure that
    # including one element at
    # index will make a combination
    # with remaining elements at
    # remaining positions
    i = start;
    while(i <= end and end - i + 1 >= r - index):
        data[index] = arr[i];
        combinationUtil(arr, data, i + 1,
                        end, index + 1, r);
        i += 1;
```

Solution EX 5

```
def binomialCoefficient(n, k):
    # since C(n, k) = C(n, n - k)
    if(k > n - k):
        k = n - k
    # initialize result
    res = 1
    # Calculate value of
    # [n * (n-1) *---* (n-k + 1)] / [k * (k-1) *----* 1]
    for i in range(k):
        res = res * (n - i)
        res = res // (i + 1)
    return res
```

Solution EX 6

```
def findNext(number,n):

    # Start from the right most digit and find the first
    # digit that is smaller than the digit next to it
    for i in range(n-1,0,-1):
        if number[i] > number[i-1]:
            break

    # If no such digit found,then all numbers are in
    # descending order, no greater number is possible
    if i == 1 and number[i] <= number[i-1]:
        print "Next number not possible"
        return

    # Find the smallest digit on the right side of
    # (i-1)'th digit that is greater than number[i-1]
    x = number[i-1]
    smallest = i
    for j in range(i+1,n):
        if number[j] > x and number[j] < number[smallest]:
            smallest = j

    # Swapping the above found smallest digit with (i-1)'th
    number[smallest],number[i-1] = number[i-1], number[smallest]

    # X is the final number, in integer datatype
    x = 0
    # Converting list upto i-1 into number
    for j in range(i):
        x = x * 10 + number[j]

    # Sort the digits after i-1 in ascending order
    number = sorted(number[i:])
    # converting the remaining sorted digits into number
    for j in range(n-i):
        x = x * 10 + number[j]

    print "Next number with set of digits is",x
```

Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                    ^
SyntaxError: invalid syntax
```

Errors

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Errors

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
```

- First, the try clause (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the except clause is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.

Errors

A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Errors

The raise statement allows the programmer to force a specified exception to occur. For example:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

Exercise

Exercise

You're going to write an interactive calculator! User input is assumed to be a formula that consist of a number, an operator (+, -, etc), and another number, separated by white space (e.g. 1 + 1). Split user input using str.split(), and check whether the resulting list is valid:

- If the input does not consist of 3 elements, raise a FormulaError, which is a custom Exception.
- Try to convert the first and third input to a float. Catch any ValueError that occurs, and instead raise a FormulaError.
- If the second input is not an operator, again raise a FormulaError.
- If the input is valid, perform the calculation and print out the result.

EX errors

```
#Custom error
class FormulaError(Exception): pass

def parse_input(user_input):
    #TODO
    return n1, op, n2

def calculate(n1, op, n2):
    if op == '+':
        return n1 + n2
    if op == '-':
        return n1 - n2
    if op == '*':
        return n1 * n2
    if op == '/':
        return n1 / n2
    raise FormulaError('{0} is not a valid operator'.format(op))

while True:
    user_input = input('>>> ')
    if user_input == 'quit':
        break
    n1, op, n2 = parse_input(user_input)
    result = calculate(n1, op, n2)
    print(result)
```

Solution EX errors

```
#Custom error
class FormulaError(Exception): pass

def parse_input(user_input):
    input_list = user_input.split()
    if len(input_list) != 3:
        raise FormulaError('Input does not consist of three elements')
    n1, op, n2 = input_list
    try:
        n1 = float(n1)
        n2 = float(n2)
    except ValueError:
        raise FormulaError('The first and third input value must be numbers')
    return n1, op, n2

def calculate(n1, op, n2):
    if op == '+':
        return n1 + n2
    if op == '-':
        return n1 - n2
    if op == '*':
        return n1 * n2
    if op == '/':
        return n1 / n2
    raise FormulaError('{0} is not a valid operator'.format(op))

while True:
    user_input = input('>>> ')
    if user_input == 'quit':
        break
    n1, op, n2 = parse_input(user_input)
    result = calculate(n1, op, n2)
    print(result)
```

Common string operations / Formatting

Accessing arguments by position:

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 3.1+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')      # unpacking argument sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')   # arguments' indices can be repeated
'abracadabra'
```

Common string operations / Formatting

Accessing arguments by name:

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.8
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

Accessing arguments' items:

```
>>> coord = (3, 5)
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3;  Y: 5'
```

Common string operations / Formatting

Aligning the text and specifying a width:

```
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:^30}'.format('centered')
'           centered           '
>>> '{:*^30}'.format('centered')  # use '*' as a fill char
'***********centered***********'
```

Specifying a sign:

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # show it always
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14)  # show a space for positive numbers
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14)  # show only the minus -- same as '{:f}; {:f}'
'3.140000; -3.140000'
```

Common string operations / Formatting

Number	Format	Output	Description
3.1415926	{:.2f}	3.14	Format float 2 decimal places
3.1415926	{:+.2f}	+3.14	Format float 2 decimal places with sign
-1	{:+.2f}	-1.00	Format float 2 decimal places with sign
2.71828	{:.0f}	3	Format float with no decimal places
5	{:0>2d}	05	Pad number with zeros (left padding, width 2)
5	{:x<4d}	5xxx	Pad number with x's (right padding, width 4)
10	{:x<4d}	10xx	Pad number with x's (right padding, width 4)
1000000	{:,}	1,000,000	Number format with comma separator
0.25	{:.2%}	25.00%	Format percentage
1000000000	{:.2e}	1.00e+09	Exponent notation
13	{:10d}	13	Right aligned (default, width 10)
13	{:<10d}	13	Left aligned (width 10)
13	{:^10d}	13	Center aligned (width 10)

Common string operations / Formatting

This example uses a precision variable to control how many decimal places to show:

```
pi = 3.1415926
precision = 4
print( "{:.{}f}".format( pi, precision ) )
~~ 3.1415
```

This example formats the number 21 in each base:

```
print("{0:d} - {0:x} - {0:o} - {0:b} ".format(21))
~~ 21 - 15 - 25 - 10101
```

You can use .format as a function to separate text and formatting from code:

```
## defining formats
email_f = "Your email address was {email}".format

## use elsewhere
print(email_f(email="bob@example.com"))
```

Exercise

Ex 1

Write a format string that will take the following four element tuple: (2, 123.4567, 10000, 12345.67) and produce: 'file_002 : 123.46, 1.00e+04, 1.23e+04'

Ex 2

Write a function that takes as input a tuple of integers and returns a string of the integers separated by a '+' sign.

Ex 3

Write some Python code to print a table of several rows, each with a name, an age and a cost. Make sure some of the costs are in the hundreds and thousands to test your alignment specifiers.

Frame Title

```
def EX1():
    t = (2, 123.4567, 10000, 12345.67)
    print('file_{:0>3d} :{:9.2f}, {:.2e}, {:.3g}'.format(*t))
```

```
def EX2():
    in_tuple = (1,2,3,4,5)
    num = str(len(in_tuple))
    form_string = ""
    for x in range(0,len(in_tuple)):
        form_string = form_string + "{:d} + "
    print(form_string.format(*in_tuple))
```

```
def EX3():
    l = [['Superman', 29, 10000],
         ['Wonder Woman', 5000, 1000],
         ['Spiderman', 15, 100]]
    for row in l:
        print('{:>{width}s} {:>{width}d} {:>{width}d}'.format(*row, width=15))
```

Practice : crude integrale approximation

Write a function `f(a, b, c)` that returns $a^b - c$. Form a 24x12x6 array containing its values in parameter ranges `[0,1] x [0,1] x [0,1]`.

Approximate the 3-d integral

$$\int_0^1 \int_0^1 \int_0^1 (a^b - c) da\, db\, dc$$

over this volume with the mean. The exact result is: $\ln 2 - \frac{1}{2} \approx 0.1931\ldots$ – what is your relative error?

(Hints: use elementwise operations and broadcasting. You can make `np.ogrid` give a number of points in given range with `np.ogrid[0:1:20j]`.)

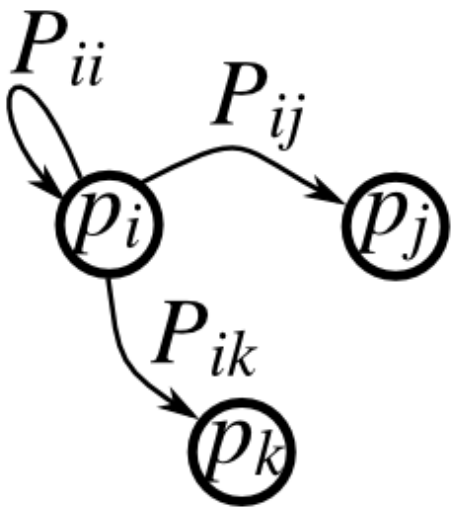
Reminder Python functions:

```
def f(a, b, c):
    return some_result
```

Solution crude integrale approximation

```
import numpy as np
from numpy import newaxis

def f(a, b, c):
    return a**b - c

a = np.linspace(0, 1, 24)
b = np.linspace(0, 1, 12)
c = np.linspace(0, 1, 6)

samples = f(a[:,newaxis,newaxis],
            b[newaxis,:,newaxis],
            c[newaxis,newaxis,:])

integral = samples.mean()

print("Approximation:", integral)
```

Practice : Markov Chain



$$\sum_j P_{ij} = 1$$

Markov chain transition matrix `P`, and probability distribution on the states `p`:

1. `0 <= P[i,j] <= 1`: probability to go from state `i` to state `j`
2. Transition rule: $p_{new} = P^T p_{old}$
3. `all(sum(P, axis=1) == 1)`, `p.sum() == 1`: normalization

Write a script that works with 5 states, and:

- Constructs a random matrix, and normalizes each row so that it is a transition matrix.
- Starts from a random (normalized) probability distribution `p` and takes 50 steps => `p_50`
- Computes the stationary distribution: the eigenvector of `P.T` with eigenvalue 1 (numerically: closest to 1) => `p_stationary`

Remember to normalize the eigenvector — I didn't...

- Checks if `p_50` and `p_stationary` are equal to tolerance 1e-5

Toolbox: `np.random.rand`, `.dot()`, `np.linalg.eig`, reductions, `abs()`, `argmin`, comparisons, `all`, `np.linalg.norm`, etc.

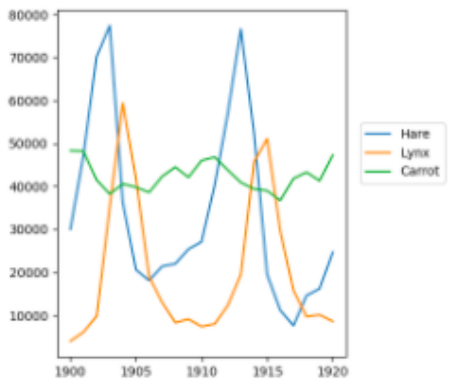
Solution Markov Chain

```python
import numpy as np

np.random.seed(1234)

n_states = 5
n_steps = 50
tolerance = 1e-5

# Random transition matrix and state vector
P = np.random.rand(n_states, n_states)
p = np.random.rand(n_states)

# Normalize rows in P
P /= P.sum(axis=1)[:,np.newaxis]

# Normalize p
p /= p.sum()

# Take steps
for k in range(n_steps):
    p = P.T.dot(p)

p_50 = p
print(p_50)

# Compute stationary state
w, v = np.linalg.eig(P.T)

j_stationary = np.argmin(abs(w - 1.0))
p_stationary = v[:,j_stationary].real
p_stationary /= p_stationary.sum()
print(p_stationary)

# Compare
if all(abs(p_50 - p_stationary) < tolerance):
    print("Tolerance satisfied in infty-norm")

if np.linalg.norm(p_50 - p_stationary) < tolerance:
    print("Tolerance satisfied in 2-norm")
```

Practice : Data statistics

The data in **`populations.txt`** describes the populations of hares and lynxes (and carrots) in northern Canada during 20 years:

```
>>> data = np.loadtxt('data/populations.txt')
>>> year, hares, lynxes, carrots = data.T  # trick: columns to variables
>>> import matplotlib.pyplot as plt
>>> plt.axes([0.2, 0.1, 0.5, 0.8])
<matplotlib.axes...Axes object at ...>
>>> plt.plot(year, hares, year, lynxes, year, carrots)
[<matplotlib.lines.Line2D object at ...>, ...]
>>> plt.legend(('Hare', 'Lynx', 'Carrot'), loc=(1.05, 0.5))
<matplotlib.legend.Legend object at ...>
```



Computes and print, based on the data in `populations.txt`...

1. The mean and std of the populations of each species for the years in the period.
2. Which year each species had the largest population.
3. Which species has the largest population for each year. (Hint: `argsort` & fancy indexing of `np.array(['H', 'L', 'C'])`)
4. Which years any of the populations is above 50000. (Hint: comparisons and `np.any`)
5. The top 2 years for each species when they had the lowest populations. (Hint: `argsort`, fancy indexing)
6. Compare (plot) the change in hare population (see `help(np.gradient)`) and the number of lynxes. Check correlation (see `help(np.corrcoef)`).

... all without for-loops.

Solution Data statistics

```
import numpy as np

data = np.loadtxt('../../../data/populations.txt')
year, hares, lynxes, carrots = data.T
populations = data[:,1:]

print("       Hares, Lynxes, Carrots")
print("Mean:", populations.mean(axis=0))
print("Std:", populations.std(axis=0))

j_max_years = np.argmax(populations, axis=0)
print("Max. year:", year[j_max_years])

max_species = np.argmax(populations, axis=1)
species = np.array(['Hare', 'Lynx', 'Carrot'])
print("Max species:")
print(year)
print(species[max_species])

above_50000 = np.any(populations > 50000, axis=1)
print("Any above 50000:", year[above_50000])

j_top_2 = np.argsort(populations, axis=0)[:2]
print("Top 2 years with lowest populations for each:")
print(year[j_top_2])

hare_grad = np.gradient(hares, 1.0)
print("diff(Hares) vs. Lynxes correlation", np.corrcoef(hare_grad, lynxes)[0,1])

import matplotlib.pyplot as plt
plt.plot(year, hare_grad, year, -lynxes)
plt.savefig('plot.png')
```