Python programming language

A quick overview

IES San Vicente

Table of Contents

1. Introduction

Python is a programming language created at the beginning of the 90's. Its name is a tribute to Monty Python comedians. Among all its virtues, we can emphasize the following ones:

- It is very popular in the first stages of programming learning process because it is quite easy to understand.
- It is a multi-platform programming language, so you can learn it in Windows, Mac OS X or Linux platforms.
- It is interpreted (not compiled), so we can use it as a script language as well (just like PowerShell or any other script language)
- It uses dynamic typing, this is, there are no implicit data types (a variable can be used as an integer at a given point, and as a string later).
- You can use either object oriented programming or imperative programming paradigms when working with Python

1.1. Choosing an appropriate IDE

The first thing that we need to get before starting with our first program is an appropriate development environment (IDE). You can find a lot of them on the Internet, such as IDLE (free and multi-platform), or even some specific plugins to add in multi-language IDEs such as NetBeans or Eclipse. In this document, we are going to rely on either IDLE, which can be downloaded along with the corresponding version of Python, or NetBeans, which lets us install a plugin to create and run Python projects and source files. We can also use an on-line interpreter to test our Python programs, if we are only "playing" with the language.

1.1.1. Python version

There are two different Python versions that are alive at this time: Python 2.x and Python 3.x. The main differences between them are:

- Python 2.x is not going to have further improvements, but it is the most supported version (some versions of Mac and Linux do not support Python 3.x yet)
- Python 3.x is "the future" of Python, and the only one which is going to have future improvements and new libraries.

In order to follow the examples and exercises of this document, it does not matter which version to use, although there are some slight changes in the syntax. So you are free to decide.

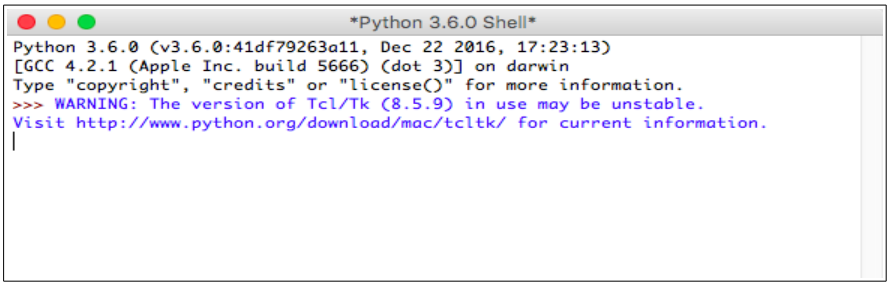
1.1.2. Using IDLE

You can IDLE environment from this link. Choose your desired version and install it. Along with Python libraries, you will be provided with IDLE development environment.

If you have Python 2.x installed in your system, you can run it by either typing "idle" in the terminal (if you are using Mac or Linux), or by choosing the corresponding shortcut from the menu in Windows. You can do the same if you used *Python 3.x,* but for some operating systems you may find some troubles. If you installed Python 3.x on Mac, then you should

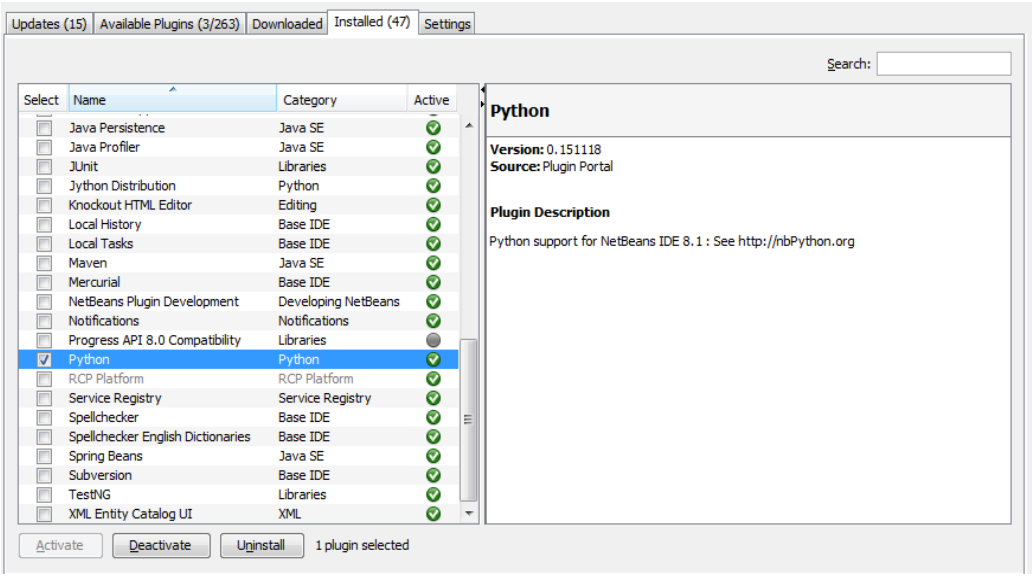
go to the *Applications* section in your Mac and unfold the *Python 3.x* folder to load IDLE from it.

Finally, you should get this window:



1.1.3. Using NetBeans

If you choose NetBeans as your environment, then you can install a plugin called *Python*, that lets us create Python projects with several files or modules, and run them from NetBeans. To install this plugin, just go to *Tools* > *Plugins* menu, search for *Python* in the *Available Plugins* and install it, along with all the other needed packages.



Then, after restarting NetBeans, you can create new Python projects (either with existing sources or not).

1.1.4. Online interpreters

The third (but also useful) option to try Python relies on online interpreters that are available in many web sites, such as this one, or this one. You just type your code in the code panel and then run your app. You can also interact with the program in an embedded console:

1.2. First steps. Our "hello world" program

Once we have Python properly installed, we can start with our first program.

In order to print a "Hello" message in Python, we can just type the following instruction, either with parentheses (Python 3) or without them (Python 2):

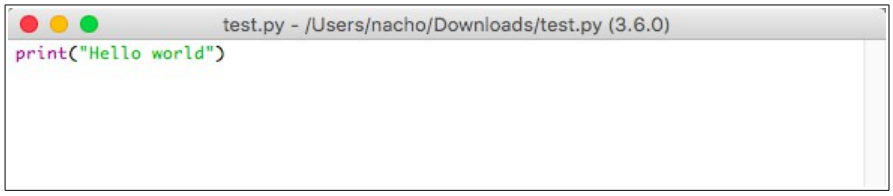
```
print("Hello world")
```

If we are using IDLE as our Python editor, we can just type this instruction in the terminal, and we will get an immediate response.

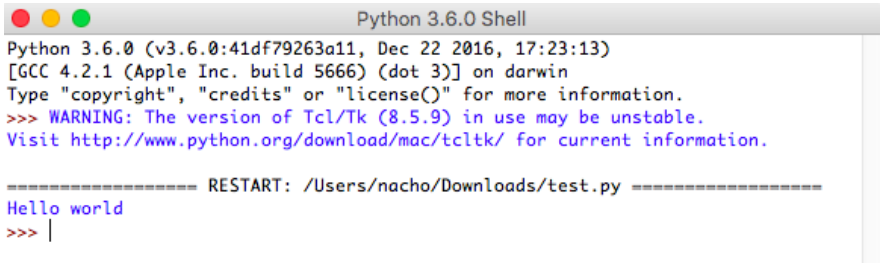
1.2.1. Creating a source code file

We can also store Python instructions in a source code file (just like we do in other programming languages), so that we can run this file whenever we want. To do this:

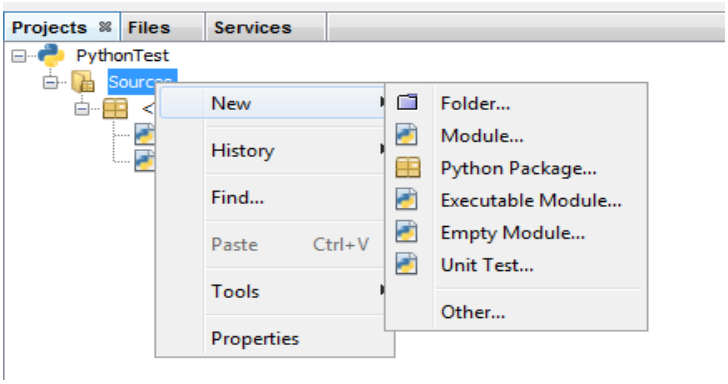
- If we are using IDLE, we need to start IDLE editor from menu *File* > *New File*. A new window will be shown, and we can add our Python instructions there:



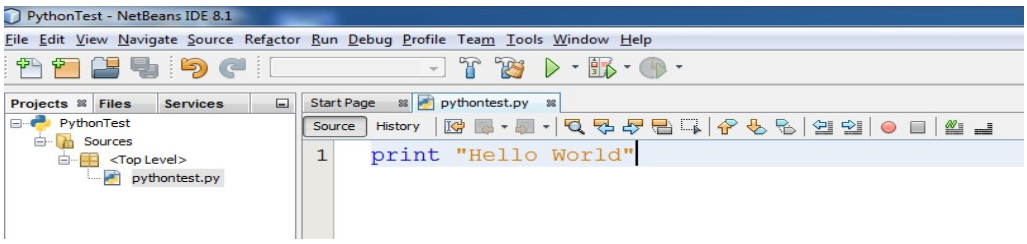
  Then, we can save the document in our preferred folder with an appropriate name, and with a *.py* extension (for instance, *example.py*), and run it from the *Run* > *Run module* menu option (or by pressing F5). When you run the file, you can see the results in IDLE terminal:



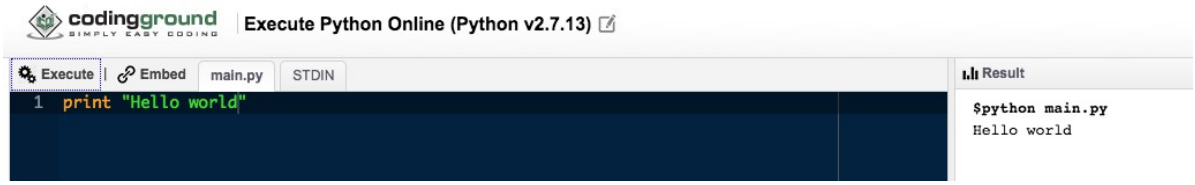
- If we are using NetBeans, then we create a new module to our project:



  Then, we type all the instructions in this source file, and run it with right click > *Run File*.



Finally, if we are using an online interpreter, we can just type the instruction in the code area, and press the *Run* or *Execute* button:



2. Basic elements of Python

Let's see some basic elements that are present in every programming language, such as comments, variable declaration and initialization, user input, screen output...

2.1. Comments

Comments in Python are written in lines starting by '#' symbol. For instance:

```
#We print a standard message
print "Hello world"
```

2.2. Basic data types

We have 3 basic types in Python:

- *Numbers*, that can be either integers (3), real (4.33) or complex (5 + 3j)
- *Strings*, such as "Hello world". We can put them between either double quotes ("Hello world") or single quotes ('Hello world'), and we can use some escape sequences inside them, such as \n or \t ("Hello\tworld")
- *Boolean*, that can be *True* or *False*

2.2.1. Variable declaration and initialization

In order to declare the variable and assign a value to it, we just put the variable name and its value assigned. For instance:

```
age = 23
message = "Hello World"
```

2.2.2. Data conversion

If we want to convert from one data type to another, there are some useful built-in functions, such as:

- *int(value)* converts to an integer the *value* passed as an argument. This *value* can be either a numeric value (another integer, even in a different base, or a floating point number...), or a string.
- *float(value)* works like *int* function, but it converts to float or real number.
- *str(value)* gets a string representation of the specified *value*

2.3. Operators

2.3.1. Arithmetic operators

We have these arithmetic operators available in Python:

- + for addition
- - for substraction
- * for multiplication

- ** for power
- / for division (the result will be of the same type than the numbers involved)
- // for integer division
- % for module

2.3.2. Relational operators

In order to compare two elements, we can use these operators: **==, !=, <, <=, >, >=.** As you can see, they are more or less the same that we can see in many other languages.

2.3.3. Logical operators

We have the operators **and, or** and **not** to represent these logical operators. For instance:

```
3 > 2 and 5 > 1
```

2.4. String manipulation

Apart from using escape sequences within the strings, we can also do some other operations, such as concatenation with + operator:

```
text = "I have " + 23 + " years"
```

or repeating a string a number of times, with * operator

```
text2 = "Hello" * 3    #Stores "HelloHelloHello"
```

There are also some old friends from other languages, such as:

- *split* method, that returns an array (list) of elements splitting the string by the specified delimiter:

  ```
  parts = text.split(",")
  ```

- *replace* method, that replaces all the ocurrences of the first argument with the second argument, and returns a new string with this replacement:

  ```
  replacedString = text.replace("old", "new")
  ```

- *lower* and *upper* methods, convert all the string to lowercase or uppercase, respectively, and return the new string as a result:

  ```
  textToUpper = text.upper()
  ```

2.5. Basic input/output

2.5.1. Output with *print*

We have seen how *print* instruction works to show messages on the screen. We can just print a single message in the screen...

```
print "Hello world"
```

Or combine multiple elements, separated by commas:

```
print "The sum of ", number1, " and ", number2, " is ", result
```

We can also provide some output format, in a similar way than *printf* instruction does in other languages, such as C or Java. For instance, we can define a horizontal gap for a given number:

```
print "My name is Nacho and I am %3d years old" % (age)
```

Or specify how many integer and decimal digits must be printed in a floating point number:

```
print "The final result is %3.2f" % (result)
```

If we need to format more than one number, we can specify some data within the parentheses, separated by commas:

```
print "The sum of %d and %d is %5d" % (number1, number2, result)
```

As happens with C or Java, we can use special symbols such as %d, %f or %s to represent basic data types such as integers, floating point numbers or strings, respectively.

2.5.2. Getting user input

In order to get user input, we can choose between two options:

- If we want to get a number, we have to use *input* function, so that we can do arithmetic operations with it:

  ```
  age = input()
  ```

- Otherwise, we can use *raw_input* function:

  ```
  userName = raw_input()
  ```

Exercise 1

Do the following simple Python programs:

1. A program called *Percentages.py* that asks the user to introduce the total number of boys and girls that are in a classroom, and then calculates the percentages of each one. This is an example of how it should workd:

   ```
   Enter the number of boys:
   12
   Enter the number of girls:
   8
   There are 60 % boys and 40 % girls
   ```

2. A program called *Greeting.py* that asks the user to introduce his name and age, and then prints a message saying hello to him, this way:

   ```
   Hello Nacho, you are 40 years old
   ```

3. A program called *Average.py* that asks the user to introduce four integer numbers, and calculates the total (real) average. This average must be shown in the screen using 3 decimal digits.

3. Control structures

Control structures allows us to create programs with multiple paths to follow, depending on some conditions. The control structures present in Python are very usual in other programming languages: *if, if..else, while* or *for*. There are, however, some old friends missing in Python, such as *switch/case* structures.

3.1. if, if..else, if..elif..else

3.1.1. Basic *if* clause

If we want to run a block of instructions depending on a given condition, we put this condition in an ***if* clause**. The block of instructions involved must be indented from this *if*. For instance:

```
number = input()
if number > 0:
     number = number + 1
     print "The following number is", number
print "End of the program"
```

3.1.2. *if..else* clause

We can also distinguish between two possible paths with an ***if..else* clause**. We must indent the block of instructions for each clause as well:

```
number = input()
if number > 0:
     number = number + 1
     print "The following number is", number
else:
     print "The number is not positive"
print "End of the program"
```

3.1.3. Nesting *if..else* clauses

In case we need to have more than two different paths, we can nest *if..else* clauses inside another previous ones:

```
number = input()
if number > 0:
     number = number + 1
     print "The following number is", number
else:
     if number < -10:
          print "Number is too low"
```

```
     else:
          print "The number is not positive"
print "End of the program"
```

3.1.4. *if..elif* clause

Instead of nesting *if..else* structures inside another ones, we can use an *if..elif* structure to specify more than one condition block. Previous example could have been written this way:

```
if number > 0:
     number = number + 1
     print "The following number is", number
elif number < -10:
     print "Number is too low"
else:
     print "The number is not positive"
print "End of the program"
```

3.2. Loops

3.2.1. *while* clause

The *while* structure has a sintax similar to the *if* clause:

```
number = input()
while number > 0:
     print "You typed", number
     number = input()
print "End"
```

There is no *do..while* structure in Python, since it can always be implemented with a *while* structure with some minor changes.

3.2.2. *for* clause

As in many other languages, Python has a *for* clause, although it can be used in some different ways. We can, for instance, explore a given set of values:

```
for value in [2, 4, 5]:
     print value
```

In this example, *value* variable will take the different values of the list (2, 4 and 5).

We can also iterate from a given initial value to a given final value (this is, the "classical" use of *for* clause):

```
for value in range(1, 5):
     print value
```

In this example, numbers from 1 to 4 will be printed (the right limit of the range is excluded). If only a number is specified in the range, then *for* loop goes from 0 to this number (excluding the number). So, `for value in range(5)` goes from 0 to 4.

Finally, we can specify an increment different than 1, specifying a third parameter inside the *range* option. For instance, if we want to count from 0 to 100 with an increment of 10 (this is, 0, 10, 20, 30...), we can do this:

```
for value in range(0, 101, 10):
     print value
```

3.2.3. Using *break*

There is a *break* sentence, similar to the one found in many other languages, that lets us get out of a loop at a given point.

```
number = input()
while number > 0:
     if number == 10:
          break
     print number
     number = input()
```

Exercise 2

Implement the following programs in Python:

1. A program called *Marks.py* that asks the user to enter three marks, and then calculate the final mark according to the following rules:
    - If no mark is greater than 4, then the final mark must be 0
    - If some marks are greater than 4 (but not all of them), then the final mark must be 2
    - If all marks are greater than 4, then the final mark must be 30% of first mark, 20% of second mark and 50% of third mark
2. A program called *Receipt.py* that asks the user to introduce different prices for a receipt, until he enters 0. Then, the total must be printed with 2 decimal digits
3. A program called *HighestLowest.py* that asks the user to enter a set of N numbers (first, the user must specify how many numbers he is going to enter). At the end of the process, the program must tell the user which is the highest and the lowest number that he has entered. For instance:

```
Tell me how many numbers are you going to enter:
3
Enter the 3 numbers:
3
7
2
The highest number has been 7
The lowest number has been 2
```

4. Functions

Functions let us group our code and make it reusable. To define a function in Python, we start it with the word *def,* followed by the function name and the arguments between parentheses. As happens with *if* or *while* clauses, all the code within a function must be indented. Besides, every function must have a *return* statement. If we want the function to return a value, we put this value next to the *return* statement. Otherwise, we leave the *return* statement empty (this would be equivalent to a *void* function).

Let's see some examples. This function takes two numbers and returns the highest of them

```
def maxNumber(num1, num2):
     if num1 > num2:
          return num1
     else:
          return num2
```

This function takes a text as a parameter and prints it:

```
printText(text)
     print text
     return
```

In order to call these functions from other parts of the program, we do it as in many other languages:

```
print "Enter two numbers:"
number1 = input()
number2 = input()
print "The maximum is", maxNumber(number1, number2)
print "Enter a text:"
text = raw_input()
printText(text)
```

4.1. Pass by reference

In Python, every argument is passed by reference. This means that, as long as we keep the same reference, argument's value can be changed inside the function, and this change persists outside the function. For instance, if we use this function:

```
def appendValues(list):
     list.append([30, 40])
     print "Values inside", list
     return
```

and we call it this way:

```
theList = [10, 20]
appendValues(theList)
print "Values outside", theList
```

Then variable *theList* stores the following values either inside or outside the function: [10, 20, 30, 40]. However, if we have this function:

```
def appendValues(list):
     list = [30, 40]
     print "Values inside", list
     return
```

and we call it in the same way as shown before, then the list has the values [30, 40] inside the function, and [10, 20] outside the function, because we have changed the reference of the variable inside the function, and thus we have created a new reference apart from the original one.

4.2. Argument types

Arguments defined in a function can be:

- **Required**: this is the normal usage. If we just declare each argument's name, they are all required, and must be placed in the same order. Here you can see an example (same as shown before):

  ```
  def maxNumber(num1, num2):
       if num1 > num2:
            return num1
       else:
            return num2
  ```

- **Keyword**: we can use original argument's name when calling the function. In this case, we don't need to follow the same argument order than the original definition. For instance:

  ```
  def printData(name, age):
       print "Your name is", name, "and you are", age
       return
  ...
  printData(age = 28, name = "John")
  ```

- **Default**: when we declare the function, we can set a default value for all/some arguments, by assigning a default value to them. For instance:

  ```
  def printData(name, age = 0):
       print "Your name is", name, "and you are", age
       return
  ```

  In this case, if we need to call the function, we can omit all/any of these default values.

```
printData("John") # Prints "Your name is John and you are 0"
```

- **Variable-length**: we can specify a last argument of a special type that allows us to pass as many arguments as we want to our function. For instance:

```
def printAllData(number1, *numbers):
     print "First number:", number1
     for num in numbers:
          print num
     return
```

Exercise 3

Create these Python functions in the same source file called *Functions.py*:

1. A function called *fibonacci* that gets an integer *n* as a parameter, and returns the *n-th* number of Fibonacci series
2. A function called *isPrime* that gets a number as a parameter and returns a boolean indicating if this number is prime or not.

From the main program, call the *fibonacci* function to print the 5th and 10th numbers of Fibonacci series. Then, use *isPrime* function to print the prime numbers from 1 to 50

5. Lists

Lists are (maybe) the most popular way of dealing with collections in Python. A list in Python is a sequence of elements (they can be from different types). We represent it with the elements separated by commas, and between square brackets. For instance:

```
data = ['Programming', 'Web design', 2015, 2013]
```

We can also initialize an empty list by calling the *list* method:

```
data = list()
```

5.1. Accessing and showing elements

Elements within a list are referenced by an index, starting by zero. So in previous example, if we print *data[1]*, we would print *Web design*.

We can use *print* command to print the data contained in a list, this way:

```
print data[2] # Result is 2015
print data  # Result is ['Programming', 'Web design', 2015, 2013]
```

5.2. Adding, updating and deleting values

If we want to **add new values** to a list, we can use the *append* method from the list object. For instance:

```
data.append(2017)
```

It is also possible to add elements in the middle of the list, by calling the *insert* method and specifying position where the new element must be placed, and the value of this element:

```
data.insert(3, 1996)
```

We can re-assign or **update the value** of any element from the list, just by indicating its index, and the new value. For instance:

```
data[2] = 2016
```

If we want to **remove a value** from a list, we use *del* instruction followed by the element to be removed. For instance:

```
del data[2]
```

We can also use *remove* method from the list object:

```
data.remove(2)
```

5.3. Some other useful operations

There are some other operations with lists that can be very useful, such as:

- *len(list)* returns the number of elements in the list
- *list(value)* returns a list from a set of iterable items. For instance, if we pass a string such as `list('123')`, then it returns the list `['1', '2', '3']`
- *list* + *list* concatenates both lists' elements

- *list * N* generates a new list where all the elements of the original *list* are repeated N times
- *n in list* returns a boolean indicating if value *n* is contained in *list*
- *max(list)* and *min(list)* return the maximum or minimum value (respectively) from the list
- *list.count(obj)* returns the number of occurrences of object *obj* in the list
- *list.index(obj)* returns the lowest index in the list in which *obj* appears
- *list.reverse()* reverses objects' order in the list. This method does NOT return a new list, but modifies the existing one
- *list.sort(function)* sorts the list according to the criteria specified in the function provided (if any, because this argument is optional). As with *reverse*, this function does NOT return any new list, it modifies the existing one.

Exercise 4

Implement these programs using Python:

1. A program called *ReverseList.py* that asks the user to enter a set of names separated by commas, and then prints the same list in reverse order.
2. A program called *Lottery.py* that asks the user to enter the 6 numbers of the lottery (separated by whitespaces), then creates a list with them, sorts the list in ascending order and prints it in the screen. Besides, the program must tell if it is a valid list (i.e. it has no repeated numbers nor numbers < 1 or > 49). Here is an example of usage:

```
Enter the 6 numbers separated by whitespaces
1 20 12 20 6 50
[1, 6, 12, 20, 20, 50]
The list is NOT valid
There are repeated numbers
There are numbers lower than 1 or higher than 49
```

6. Comparing languages

In this section we are going to see how a problem can be solved in Python, and how short and compact the code is if we compare it with a traditional language, such as C# or Java.

6.1. The problem to solve: Cooking the books

This problem has been extracted from Facebook Hacker Cup 2015 Qualification Round. Its original title is *Cooking the books.*

Every business can make use of a good accountant and, if they're not big on following the law, sometimes a bad one. Bad accountants try to make more money for their employers by fudging numbers without getting caught.

Sometimes a bad accountant wants to make a number larger, and sometimes smaller. It is widely known that tax auditors will fail to notice two digits being swapped in any given number, but any discrepancy more egregious will certainly be caught. It's also painfully obvious when a number has fewer digits than it ought to, so a bad accountant will never swap the first digit of a number with a 0.

Given a number, how small or large can it be made without being found out?

Input

Input begins with an integer T, the number of numbers that need tweaking. Each of the next T lines contains a integer N.

Output

For the ith number, print a line containing "Case #i: " followed by the smallest and largest numbers that can be made from the original number N, using at most a single swap and following the rules above.

Constraints

1 ≤ T ≤ 100

0 ≤ N ≤ 999999999

N will never begin with a leading 0 unless N = 0

Example input	***Example output***
`5`	
`31524`	`Case #1: 13524 51324`
`897`	`Case #2: 798 987`
`123`	`Case #3: 123 321`
`10`	`Case #4: 10 10`
`5`	`Case #5: 5 5`

6.2. Possible solution in C# or Java

If we try to solve this problem in C# (or Java) in a "traditional way", we could get something like this:

```
using System;
using System.IO;
using System.Text;

class CookingTheBooks
{
      // Generates each line of the answer
    public static void ProcessAndAnswer(int c, int n1, int n2)
    {
        Console.WriteLine("Case #{0}: {1} {2}", c+1, n1, n2);
    }

    // Swaps two digits from a given number (those in positions i and j)
    public static void SwapDigits(ref StringBuilder number, int i, int j)
    {
           char c1 = number[i];
           char c2 = number[j];
           number[i] = c2;
           number[j] = c1;
    }

    // Orders the digits of the number in ascendent (true) or descendent
    // (false) order. 0 in 1st position is forbidden unless number is 0
    public static void OrderNumber(ref StringBuilder number, bool order)
    {
           for (int i = 0; i < number.Length - 1; i++)
                 for (int j = i+1; j < number.Length; j++)
                 {
                       if ((number[i] > number[j] && order &&
                            (i != 0 || number[j] != '0')) ||
                            (number[i] < number[j] && !order &&
                            (i != 0 || number[j] != '0')))
                       {
                             SwapDigits(ref number, i, j);
                       }
                 }
    }

    // Main function
    public static void Main(string[] args)
    {
           int c, i, j, k, t;

           // We read the total amount of numbers (T)
           t = Convert.ToInt32( Console.ReadLine() );

           // We read and process each number (N)
           for(i=0; i<t; i++)
           {
                 // Read the number
                 // We will use this variable to set the minimum number
                 StringBuilder number =
                     new StringBuilder(Console.ReadLine());

                 // If the number is 0, then return 0 0
                 if (Convert.ToInt32(number.ToString()) == 0)
```

```
            ProcessAndAnswer(i, 0, 0);
        else
        {
            // Copy the number into another variable
            // This will be the maximum number
            StringBuilder numberMax =
                new StringBuilder(number.ToString());

            // Copy number into another variable to order it
            StringBuilder numberToOrder =
                 new StringBuilder(number.ToString());

            // Finding the minimum:

            OrderNumber(ref numberToOrder, true);

            c = number.Length - 1;
            for (j = 0; j < number.Length; j++)
                 if (number[j] != numberToOrder[j])
                 {
                      c = j;
                      break;
                 }

            char lowest = '9';
            int posLowest = c;
            for (k = c+1; k < number.Length; k++)
            {
                 if (number[k] <= lowest &&
                     (c != 0 || number[k] != '0'))
                 {
                      lowest = number[k];
                      posLowest = k;
                 }
            }
            if (posLowest > c)
                 SwapDigits(ref number, c, posLowest);

            // Finding the maximum:

            OrderNumber(ref numberToOrder, false);

            c = numberMax.Length - 1;
            for (j = 0; j < numberMax.Length; j++)
                 if (numberMax[j] != numberToOrder[j])
                 {
                      c = j;
                      break;
                 }

            char highest = '0';
            int posHighest = c;
            for (k = c+1; k < numberMax.Length; k++)
            {
                 if (numberMax[k] >= highest &&
                     (c != 0 || numberMax[k] != '0'))
```

```
                        {
                              highest = numberMax[k];
                              posHighest = k;
                        }
                  }
                  if (posHighest > c)
                        SwapDigits(ref numberMax, c, posHighest);

                  // Generate the case result
                  ProcessAndAnswer(i,
                      Convert.ToInt32(number.ToString()),
                      Convert.ToInt32(numberMax.ToString()));
            }
      }
  }
}
```

We could, of course, have a shorter code. This is an example of what someone sent in Java:

```
import java.io.*;
import java.math.*;
import java.text.*;
import java.util.*;

public class A {
      private static BufferedReader br;
      private static StringTokenizer st;
      private static PrintWriter pw;

      public static void main(String[] args) throws IOException    {
            br = new BufferedReader(new InputStreamReader(System.in));
            pw = new PrintWriter(new BufferedWriter(
                              new OutputStreamWriter(System.out)));
            final int MAX_NUM_CASE = readInt();
            for(int qq = 1; qq <= MAX_NUM_CASE; qq++)    {
                  pw.print("Case #" + qq + ": ");
                  String s = nextToken();
                  TreeSet<Long> set = new TreeSet<Long>();
                  set.add(Long.parseLong(s));
                  for(int i = 0; i < s.length(); i++) {
                        for(int j = i+1; j < s.length(); j++) {
                              String next = s.substring(0,i) + s.charAt(j)
                                  + s.substring(i+1, j) + s.charAt(i) +
                                  s.substring(j+1);
                              if(next.charAt(0) != '0') {
                                    set.add(Long.parseLong(next));
                              }
                        }
                  }
                  pw.println(set.first() + " " + set.last());
            }
            pw.close();
      }

      private static long readLong() throws IOException{
            return Long.parseLong(nextToken());
```

```
	}

	private static double readDouble() throws IOException  {
		return Double.parseDouble(nextToken());
	}

	private static int readInt() throws IOException  {
		return Integer.parseInt(nextToken());
	}

	private static String nextToken() throws IOException  {
		while(st == null || !st.hasMoreTokens())  {
			if(!br.ready())  {
				pw.close();
				System.exit(0);
			}
			st = new StringTokenizer(br.readLine().trim());
		}
		return st.nextToken();
	}
}
```

6.3. Possible solution in Python

If we take advantage of some of the Python capabilities, we can solve this problem in a few lines:

```
for t in range(input()):
    x = raw_input()
    a = b = int(x)
    x = list(x)
    for i in range(len(x)):
        for j in range(i):
            x[i],x[j] = x[j],x[i]
            if x[0] != '0':
                v = int("".join(x))
                a = min(a, v)
                b = max(b, v)
            x[i],x[j] = x[j],x[i]
    print "Case #%d: %d %d" % (t+1, a, b)
```

There are some concepts that have not been explained before. For instance, notice how Python can swap values between two variables without any auxiliary variable:

```
x[i],x[j] = x[j],x[i]
```

Another unknown element is *join* function. It generates a string by joining all the elements of the iterable set *x* (in this case, a list of digits), separating each element from the following with the separator specified (empty string, in this case). The result is a string that is then converted into an integer with the *int* function.

Exercise 5 (to deliver)

In order to practice with some of the capabilities of Python, try to solve the following challenges:

1. Create a file called *Challenge252.py* to solve this challenge from *Acepta el reto*. At least, it must print the following output for this input:

- `Acaso hubo buhos aca => SI`
- `Querido muerto esta tarde llegamos => NO`
- `Dabale arroz a la zorra el abad => SI`
- `Abalaba => SI`
- `Abracadabra => NO`

2. Create a file called *Challenge383.py* to solve this challenge from *Acepta el reto*. At least, it must print the corresponding output for the inputs shown in the challenge web page

7. Other Python features

There are some other interesting Python features that are not covered in this (short) tutorial, such as:

- File reading and writing
- Importing modules
- Object-oriented programming in Python
- Exception handling
- Regular expressions