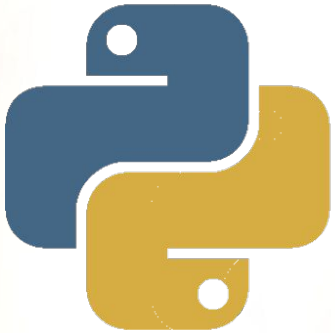
POLITECNICO
DI TORINO

Bioinformatics
a.y. 2019-2020

Introduction to



python

1

Introduction

Introduction

Extremely common, all over the world !

Top language on February 2020 PYPL index: Python is the language people are most interested in learning.

Rank	Language	Share
1	Python	29.88 %
2	Java	19.05 %
3	Javascript	8.17 %

```
def print_if_longer(sequences, limit):
    for sequence in sequences:
        if len(sequence) > limit:
            print(sequence)
```

Clear and easy to read

Python syntax is clear and easy to read: no curly braces and the interpreter forces the code to be correctly indentated.

Lots of standard and third-party libraries

State-of-the-art libraries and frameworks are available for machine learning, scientific computing, website development, etc…

Introduction

Python is a * language

INTERPRETED

*Python programs are not compiled directly to machine language and executed. Instead, they are translated into **bytecode**, which is then executed by an interpreter. An **interpreter** is a computer program that directly executes instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program.*

DYNAMICALLY TYPED

*In Python the **type of a variable refers to the value that the variable contains**. This means that if you assign an integer number to a variable, the variable is type-enforced as an integer. If you, then, assign a string to the same variable, the enforcement changes to that of a string.*

HIGH-LEVEL

*Python allows programmers to work with **powerful abstractions** and complex data types which are quite far from the original code run on the CPU. This allows the programmer to focus more on programming logic rather than on basic hardware elements like memory address and register usage.*

Introduction

Python is a *INTERPRETED* language

*Python programs are not compiled directly to machine language and executed. Instead, they are translated into **bytecode**, which is then executed by an interpreter. An **interpreter** is a computer program that directly executes instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program.*

HIGH-LEVEL

*Python allows programmers to work with **powerful abstractions** and complex data types which are quite far from the original code run on the CPU. This allows the programmer to focus more on programming logic rather than on basic hardware elements like memory address and register usage.*

DYNAMICALLY TYPED

*In Python the **type of a variable refers to the value that the variable contains**. This means that if you assign an integer number to a variable, the variable is type-enforced as an integer. If you, then, assign a string to the same variable, the enforcement changes to that of a string.*

Introduction

Python is a *DYNAMICALLY-TYPED* language

*In Python the **type of a variable refers to the value that the variable contains**. This means that if you assign an integer number to a variable, the variable is type-enforced as an integer. If you, then, assign a string to the same variable, the enforcement changes to that of a string.*

HIGH-LEVEL

*Python allows programmers to work with **powerful abstractions** and complex data types which are quite far from the original code run on the CPU. This allows the programmer to focus more on programming logic rather than on basic hardware elements like memory address and register usage.*

INTERPRETED

*Python programs are not compiled directly to machine language and executed. Instead, they are translated into **bytecode**, which is then executed by an interpreter. An **interpreter** is a computer program that directly executes instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program.*

Introduction

Python is a *HIGH-LEVEL* language

*Python allows programmers to work with **powerful abstractions** and complex data types which are quite far from the original code run on the CPU. This allows the programmer to focus more on programming logic rather than on basic hardware elements like memory address and register usage.*

INTERPRETED

*Python programs are not compiled directly to machine language and executed. Instead, they are translated into **bytecode**, which is then executed by an interpreter. An **interpreter** is a computer program that directly executes instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program.*

DYNAMICALLY TYPED

*In Python the **type of a variable refers to the value that the variable contains**. This means that if you assign an integer number to a variable, the variable is type-enforced as an integer. If you, then, assign a string to the same variable, the enforcement changes to that of a string.*

Introduction

Always refer to the official documentation

Python comes with a very large standard library that provides the user with powerful and complex abstractions. Moreover, it is evolving fast and it would be impossible to exhaustively cover all aspect of the language.

If you need details regarding a function, a class, or any language construct, always refer to the official documentation. It can be accessed from the following links:

Main page : https://docs.python.org/3/index.html
Library reference : https://docs.python.org/3/library/index.html
Language reference : https://docs.python.org/3/reference/index.html

Introduction

Python code can be executed in two ways...

Script

Implement the logic of the program in one or more files and pass the entry point as input to the interpreter.


```
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 23:03:10)
[MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more
information.
>>>
>>> print('Hello World')
Hello World
>>> exit()
```

Interactively

Call the Python interpreter without providing input files and instruction manually, one by one.

Introduction

Reference problem: *pattern matching*

Pattern matching has to do with finding the occurrences of a substring, called "*query*", in a larger string, called "*text*". It is a recurrent problem in computer science and in bioinformatics, especially in *alignment* or *assembly* tools. It can be stated as follow:

*Given a text **T** of length **n** and a pattern **P** of length **m** ≤ **n**, retrieve all positions **i** where pattern **P** occurs in text **T**, such that **i** ≤ **n**−**m***

2

Variables and Flow Control

Variables and Flow Control

Pattern Matching - version 1

“Write a program that prints all positions where a pattern occurs in a given text”

Variables and Flow Control

```
01 txt = 'ACGTACGGGACGTA'
02 qry = 'CG'
03 i = 0
04 while i < len(txt) - len(qry) + 1:
05     j = 0
06     found = True
07     while j < len(qry) and found:
08         if qry[j] != txt[i+j]:
09             found = False
10         j += 1
11     if found:
12         print(f'{qry} found at position {i}')
13     else:
14         print(f'{qry} NOT found at position {i}')
15     i += 1
```

Variables and Flow Control - variables

```
01  txt = 'ACGTACGGGACGTA'
02  qry = 'CG'
03  i = 0
05  j = 0
06  found = True
09  found = False
```

REMEMBER !

*Python is a dynamically typed language. Variables do **NOT** need to be declared with their type. You can just assign them a value, and use them.*

"*txt*" and "*qry*" are string variables

Strings are immutable collection of characters.

- Compute string length with the **len** function.

```
>>> len('Hello World')
11
```

- Access any character with square brackets.

```
>>> s = 'Hello World'
>>> s[3]
l
```

- Access any substring using slices.

```
>>> s = 'Hello World'
>>> s[1:4]
ell
```

- Concatenate strings.

```
>>> 'Hello' + 'World'
HelloWorld
```

Variables and Flow Control - variables

```
01  txt = 'ACGTACGGGACGTA'
02  qry = 'CG'
03  i = 0
05  j = 0
06  found = True
09  found = False
```

REMEMBER !

*Python is a dynamically typed language. Variables do **NOT** need to be declared with their type. You can just assign them a value, and use them.*

- Remove whitespaces from the beginning/end of a string with the "*strip*" method

```
>>> s = 'Hello World\n'
>>> s.strip()
Hello World
```

- Join a list of strings using a delimiter with the "*join*"method.

```
>>> s = '-'.join(['Bio', 'Course', '2020'])
>>> s
Bio-Course-2020
```

- Split a string using a delimiter with the "*split*"method.

```
>>> s = 'Hello World'
>>> s.split(' ')
['Hello', 'World']
```

Variables and Flow Control - variables

```
01 | txt = 'ACGTACGGGACGTA'
02 | qry = 'CG'
03 | i = 0
05 | j = 0
06 | found = True
09 | found = False
```

REMEMBER !

*Python is a dynamically typed language. Variables do **NOT** need to be declared with their type. You can just assign them a value, and use them.*

"*i*" and "*j*" are numeric variables

There are three distinct numeric types: integers, floating point numbers, and complex numbers. Integers have unlimited precision.

- Arithmetic operations are supported.

```
>>> 3.5 + 5 * (-6) / 2
-11.5
```

- Bitwise operations are supported.

```
>>> 5 | 2 # bitwise OR
7
>>> 5 & 4 # bitwise AND
4
>>> 7 ^ 2 # bitwise XOR
5
```

Variables and Flow Control - variables

```
01 txt = 'ACGTACGGGACGTA'
02 qry = 'CG'
03 i = 0
05 j = 0
06 found = True
09 found = False
```

REMEMBER !

*Python is a dynamically typed language. Variables do **NOT** need to be declared with their type. You can just assign them a value, and use them.*

"*found*" is a boolean variables

Any object can be tested for truth value, for use in an if or while condition or as operand of the Boolean operations below.

- Logic boolean operators are *and*, *or*, *not*

```
>>> a = True
>>> b = False
>>> a and b
False
>>> a or b
True
>>> not a
False
```

Variables and Flow Control - if/elif/else

```
08  if qry[j] != txt[i+j]:
09      found = False
11  if found:
12      print(...)
13  else:
14      print(...)
```

REMEMBER !

*The Python interpreter forces code indentation. Bodies of if/elif/else constructs **MUST** be correctly indented.*

"if" construct drives the execution flow

In Python the if construct can appear in three fashions:

- *"if"*

```
>>> n = 5
>>> if n < 10:
>>>     print(n)
5

>>> n = 12
>>> if n < 10:
>>>     print(n)
>>>
```

Variables and Flow Control - if/elif/else

```
08  if qry[j] != txt[i+j]:
09      found = False
11  if found:
12      print(...)
13  else:
14      print(...)
```

REMEMBER !

*The Python interpreter forces code indentation. Bodies of if/elif/else constructs **MUST** be correctly indented.*

"if" construct drives the execution flow

In Python the if construct can appear in three fashions:

- *"if"/"else"*

```
>>> n = 5
>>> if n % 2 == 0:
>>>     print('even')
>>> else:
>>>     print('odd')
odd
```

Variables and Flow Control - if/elif/else

```
08  if qry[j] != txt[i+j]:
09      found = False
11  if found:
12      print(...)
13  else:
14      print(...)
```

REMEMBER !

*The Python interpreter forces code indentation. Bodies of if/elif/else constructs **MUST** be correctly indented.*

"*if*" construct drives the execution flow

In Python the if construct can appear in three fashions:

- *"if"/"elif"/"else"*

```
>>> n = 5
>>> if n > 0 and n < 4:
>>>     print('small')
>>> elif n >= 4 and n < 10:
>>>     print('medium')
>>> else:
>>>     print('large')
medium
```

Variables and Flow Control - while loop

```
04  while i < len(txt) - len(qry) + 1:
05      j = 0
06      found = True
07      while j < len(qry) and found:
08          if qry[j] != txt[i+j]:
09              found = False
10          j += 1
```

REMEMBER !

*The Python interpreter forces code indentation. Bodies of while loops **MUST** be correctly indented.*

"while" enables loops

The "while" keyword is used for implementing while-loops.

```
>>> limit = 5
>>> while limit > 0:
>>>     print(limit)
>>>     limit -= 1
5
4
3
2
1
```

Variables and Flow Control - print

```
12  print(f'{qry} found at position {i}')
14  print(f'{qry} NOT found at position {i}')
```

"print" prints on standard output

- Basic printing

```
>>> print(12)
12
>>> print('Hello World')
Hello World
>>> n = 10
>>> print(n**2)
100
>>> print('Hello' + 'World')
HelloWorld
```

- Formatted printing (*f-strings*)

```
>>> name = 'Emanuele'
>>> print(f'My name is {name}')
My name is Emanuele
>>> name = 'Bob'
>>> print(f'My name is {name}')
My name is Bob
```

Variables and Flow Control

```
01 txt = 'ACGTACGGGACGTA'
02 qry = 'CG'
03 i = 0
04 while i < len(txt) - len(qry) + 1:
05     j = 0
06     found = True
07     while j < len(qry) and found:
08         if qry[j] != txt[i+j]:
09             found = False
10         j += 1
11     if found:
12         print(f'{qry} found at position {i}')
13     else:
14         print(f'{qry} NOT found at position {i}')
15     i += 1
```

Variables and Flow Control

```
01  txt = 'ACGTACGGGACGTA'
02  qry = 'CG'
03  i = 0
04  while i < len(txt) - len(qry) + 1:
05      j = 0
06      found = txt[i:i+len(qry)] == qry
07      while j < len(qry) and found:
08          if qry[j] != txt[i+j]:
09              found = False
10          j += 1
11      if found:
12          print(f'{qry} found at position {i}')
13      else:
14          print(f'{qry} NOT found at position {i}')
15      i += 1
```

Variables and Flow Control

Pattern Matching - version 2

“Write a program that keeps track of all positions where a pattern occurs in a given text”

Variables and Flow Control

```
01  txt = 'ACGTACGGGACGTA'
02  qry = 'CG'
03  positions = []
04  for i in range(len(txt) - len(qry) + 1):
05      found = txt[i:i+len(qry)] == qry
06      if found:
07          positions.append(i)
08  print(s'Found {len(positions) occurrences}')
09  for p in positions:
10      print(f'{qry} in position {p}')
11
12
13
14
15
```

Variables and Flow Control - lists

```
03 | positions = []
08 | positions.append(i)
```

“*positions*” is a list variable

Lists are mutable and heterogeneous collections of objects.

- *Lists* are created with square-brackets.

```
>>> l = [1, ‘bioinformatics’]
>>> l
[1, ‘bioinformatics’]
```

- *Lists* length is computed with *len*.

```
>>> l = [1, 2, 3]
>>> len(l)
3
```

- A new element can be added to the list with the “*append*” method.

```
>>> l = []
>>> l.append(‘string’)
>>> l
[‘string’]
```

- Elements can be accessed with the square-brackets notation.

```
>>> l = [5, 2, 3, 0]
>>> l[1:3]
[2, 3]
```

- Check element presence using “in”.

```
>>> 12 in [4, 12, 6, 7]
True
```

Variables and Flow Control - for loop

```
09 | for p in positions:
```

“for”/”in” iterates on collection items

In Python, for allows iterating over elements of a collection through an auxiliary variable.

```
>>> collection = [1, 2, 3]
>>> for element in collection:
>>>     print(element)
1
2
3
```

```
>>> for e in ‘bio-course’.split(‘-‘):
>>>     print(e)
bio
course

>>> threshold = 20
>>> for e in [50, 3, 20]:
>>>     if e < threshold:
>>>         print(’smaller’)
>>>     elif e == threshold:
>>>         print(‘equal’)
>>>     else:
>>>         print(’bigger’)
bigger
smaller
equal
```

Variables and Flow Control - range

04 | `for i in range(...):`

The "range" type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in for loops.

```
>>> for i in range(3):
>>>     print(i)
0
1
2
```

```
>>> for i in range(5, 8):
>>>     print(i)
5
6
7
```

```
>>> for i in range(3, 12, 2):
>>>     print(i)
3
5
7
9
11
```

Variables and Flow Control

```
01  txt = 'ACGTACGGGACGTA'
02  qry = 'CG'
03  positions = []
04  for i in range(len(txt) - len(qry) + 1):
05      found = txt[i:i+len(qry)] == qry
06      if found:
07          positions.append(i)
08  print(s'Found {len(positions) occurrences}')
09  for p in positions:
10      print(f'{qry} in position {p}')
11
12
13
14
15
```

Variables and Flow Control

Pattern Matching - version 3

“Write a program that keeps track of all positions where a set of patterns occur in a given text”

Variables and Flow Control

```
01  txt = 'ACGTACGGGACGTA'
02  qry = ['AC', 'GT']
03  positions = {}
04  for q in qry :
05      positions[q] = []
06      for i in range(len(txt) - len(qry) + 1):
07          found = txt[i:i+len(q)] == q
08          if found:
09              positions[q].append(i)
10  for q in positions:
11      print(s'Found {len(positions[q])} {q} occurrences}')
12      for p in positions[q]:
13          print(f'{q} in position {p}')
14
15
```

Variables and Flow Control - dictionaries

```
03 | positions = {}
05 | positions[q] = []
```

“*positions*” is a dict variable

Dictionaries implement associative arrays. Such a data structure allows storing arbitrary objects, called “values”, and retrieve them by other objects, called “keys”.

- *Dicts* are created with curly-brackets.

```
>>> d = {}
>>> d
{}
>>> d = {‘k1’: 34, ‘k2’: ‘hello’}
>>> d
{‘k1’: 34, ‘k2’: ‘hello’}
```

- Add a new *key-value* pair to a dictionary using square-brackets.

```
>>> d = {}
>>> d[‘hello’] = ‘world’
>>> d
{‘hello’: ‘world’}
```

- Elements can be accessed with the square-brackets notation.

```
>>> d = {‘name’: ‘Bob’, ‘age’: 26}
>>> d[‘name’]
Bob
```

- Check key presence using “in”.

```
>>> ‘name’ in {‘age’: 26}
False
```

Variables and Flow Control - dictionaries

- *Dicts* can be used in “for”/”in” constructs for iterating on keys.

```
>>> d = {‘k1’: 34.98, ‘k2’: 2.2}
>>> for k in d:
>>>     print(f’{k} -> {d[k]}’)
k1 -> 34.98
k2 -> 2.2
```

- The “*values*” method can be used for iterating on values.

```
>>> d = {‘k1’: 34.98, ‘k2’: 2.2}
>>> for v in d.values():
>>>     print(f’{v}’)
34.98
2.2
```

- The “*items*” method can be used for iterating on *key-value* pairs.

```
>>> d = {‘k1’: 34.98, ‘k2’: 2.2}
>>> for k, v in d.items():
>>>     print(f’{k} -> {v}’)
k1 -> 34.98
k2 -> 2.2
```

Variables and Flow Control

```
01  txt = 'ACGTACGGGACGTA'
02  qry = ['AC', 'GT']
03  positions = {}
04  for q in qry :
05      positions[q] = []
06      for i in range(len(txt) - len(qry) + 1):
07          found = txt[i:i+len(q)] == q
08          if found:
09              positions[q].append(i)
10  for q in positions:
11      print(s'Found {len(positions[q])} {q} occurrences}')
12      for p in positions[q]:
13          print(f'{q} in position {p}')
14
15
```

3

Functions

Functions

Pattern Matching - version 4

“Implement a function that gets a query as input and returns its complement. Then, implement a function that prints all positions where a pattern, or its complement, occurs in a given text. Assume the pattern and the text are passed as input argument.”

Functions

```
01  def get_complement(qry):
02      compl_map = {'a': 't', 'c': 'g', 'g': 'c', 't': 'a'}
03      compl_qry = ''
04      for e in qry:
05          compl_qry += compl_map[e]
06
07      return compl_qry
08
09  def pattern_matching(txt, qry, complement=False):
10      if complement:
11          qry = get_complement(qry)
12      for i in range(len(txt) - len(qry) + 1):
13          found = txt[i:i+len(qry)] == qry
14          if found:
15              print(f'{qry} found at position {i}')
```

Functions

```
01 | def get_complement(qry):
09 | def pattern_matching(txt, qry, complement=False):
```

“*get_complement*” and “*pattern_matching*” are functions

- *Functions* are defined with the **def** keyword.

```
>>> def f():
>>>     print('Hello World')
>>> f
<function f at 0x7f7163989bf8>
```

- *Functions* are called with brackets.

```
>>> def f():
>>>     print('Hello World')
>>> f()
Hello World
```

- *Functions* returns values to the caller with the **return** keyword

```
>>> def f():
>>>     return [1, 2, 3]
>>> x = f()
>>> x
[1, 2, 3]
```

Functions

- *Functions* may accepts one or more input arguments.

```
>>> def f(a, b):
>>>     return a ** b
>>> x = f(4, 3)
>>> x
64
```

- Python *functions* support default arguments.

```
>>> def f(a, b=2, c=1):
>>>     return a ** b + c
>>> x = f(10)
>>> x
101
>>> x = f(10, b=3)
>>> x
1001
>>> x = f(10, c=3)
>>> x
103
>>> x = f(10, b=3, c=3)
>>> x
1003
```

Functions

```
01  def get_complement(qry):
02      compl_map = {'a': 't', 'c': 'g', 'g': 'c', 't': 'a'}
03      compl_qry = ''
04      for e in qry:
05          compl_qry += compl_map[e]
06
07      return compl_qry
08
09  def pattern_matching(txt, qry, complement=False):
10      if complement:
11          qry = get_complement(qry)
12      for i in range(len(txt) - len(qry) + 1):
13          found = txt[i:i+len(qry)] == qry
14          if found:
15              print(f'{qry} found at position {i}')
```

4

File I/O

File I/O

Pattern Matching - version 5

“Implement a function that gets as input the path of a file storing a query and a text string and returns all positions where the pattern occurs in a given text.”

File I/O

```
01  def pattern_matching(txt, qry_path):
02      fp = open(qry_path, 'r')
03      qry = fp.readline()
04      qry = qry.strip()
05      positions = []
06      for i in range(len(txt) - len(qry) + 1):
07          found = txt[i:i+len(qry)] == qry
08          if found:
09              positions.append(i)
10      fp.close()
11
12      return positions
13
14
15
```

File I/O

```
02 | fp = open(qry_path, 'r')
03 | qry = fp.readline()
10 | fp.close()
```

"*fp*" is a file-object

File object is the way Python mediates reading from, or writing to, files.

- *File-objects* are created with the open functions. By default, they are opened in read-only mode.

```
>>> fp = open('file.txt')
```

- It is possible to add more arguments to the *"open"* function for supporting operations other than reading.
- Once the file is no more required, it should always be closed using the "*close*" method.

```
>>> fp = open('file.txt', 'w')
>>> # Write on file…
>>> fp.close()
```

File I/O

```
02 | fp = open(qry_path, 'r')
03 | qry = fp.readline()
10 | fp.close()
```

Let's assume that the file "file.txt" has the following content:

```
 First line
Second line
Third line
```

- *File-objects* can be read line-by-line with the "*readline*" method.

```
>>> fp = open('file.txt')
>>> line = fp.readline()
>>> line
' First line\n'
>>> line.strip()
'First line'
>>> fp.readline()
'Second line\n'
```

- When each line of the file is correctly read, the "*readline*" method does not fail, but returns an empty string.

File I/O

Pattern Matching - version 6

“Implement a function that gets as input the path of a file storing a set of queries and a text string and write all positions where each pattern occurs in the text in another file.”

File I/O

```
01  def pattern_matching(txt, qry_path):
02      fp_r = open(qry_path, 'r')
03      fp_w = open('results.txt', 'w')
04      for line in fp:
05          qry = line.strip()
06          positions = []
07          for i in range(len(txt) - len(qry) + 1):
08              found = txt[i:i+len(qry)] == qry
09              if found:
10                  positions.append(i)
11          for p in positions:
12              fp_w.write(str(p) + ' ')
13          fp_w.write('\n')
14      fp_r.close()
15      fp_w.close()
```

File I/O

```
04 | for line in fp:
12 | fp_w.write(str(p) + ' ')
13 | fp_w.write('\n')
```

File-objects are iterable

File-object can be seen as a list of lines in the file they point to. The easiest way for reading a file line-by-line in Python is iterating over its file-object.

Let's assume that the file "file.txt" has the following content:

```
 First line
Second line
Third line
```

- Read the content of a file and print it.

```
>>> fp = open('file.txt', 'r')
>>> for line in fp:
>>>     print(line)
 First line
Second line
Third line
```

File I/O

```
04 | for line in fp:
12 | fp_w.write(str(p) + ' ')
13 | fp_w.write('\n')
```

- Write a string to a file.

```
>>> lines = ['Hello', 'World']
>>> fp = open('file.txt', 'w')
>>> for l in lines:
>>>     print(l + '\n')
>>> lines = ['Hello\n', 'again\n']
>>> fp.writelines(lines)
```

The content of the file "file.txt" after the writing loop is the following:

```
Hello
World
Hello
again
```