Bioinformatics
2018/2019

Introduction to Python

Pedro G. Ferreira
[dCC] FCUP

Outline

- Introduction to Python language
  - Variables and pre-defined functions
  - Python code structure
  - Development of programs in python
  - Functions and predefined methods

Python Features

- Python is “a high-level scripting language that allows for interactivity” Guido van Rossum
- Interpreted language that runs in 2 modes: script or in interactive.
- We will use version 3.x. Python 3’s interpreter can be easily downloaded and installed from https://www.python.org/downloads/.
- Jupyter notebooks http://jupyter.org/ provides a web based extended version of the Python command line. Allows development and documentation/ reproducibility.

Python Features

- Combines features from different programming paradigms including imperative, scripting, object-oriented, and functional languages.
- Main features of the language:
  - Concise and clear syntax.
  - Code indentation.
  - Set of high-level and powerful data types.
  - Simple, but effective, approach to object-oriented programming.
  - Modularity.
- Popular Integrated Development Environments (IDEs): Spyder, PyCharm or IDLE.
- Package Manager: *Anaconda*, *pip*, *canopy*.

Variables and Strings

- Variables are used to store values.

Data type	Complexity	Order	Mutable	Indexed	Heterogeneous
int	primitive	–	yes	–	–
float	primitive	–	yes	–	–
complex	primitive	–	yes	–	–
Boolean	primitive	–	yes	–	–
string	container	yes	no	yes	no
list	container	yes	yes	yes	yes
tuple	container	yes	no	yes	yes
set	container	no	no	no	yes
dictionary	container	no	yes	no	yes

From Bioinformatics Algorithms, Rocha & Ferreira

- Sets and dictionaries represent collection of unordered elements.
- Strings are a series of characters enclosed by single or double quotes.

```
01. print ("Bioinformática @ DCC")
02.
03. # string as a variable
04. txt = "Bioinformática @ DCC"
05. print (txt)
06.
07. # combine and concatenate strings
08. p1 = "Bioinformatics"
09. p2 = "Course"
10. p = p1 + " " + p2
11. print (p)
```

Lists

- Lists store a series of variables in a particular order. Lists can be traversed using indices or loops.

```
01. # Create a list
02. nucleotides = ['A', 'C', 'G', 'T']
03. # Get the first item in a list
04. first_nuc = nucleotides[0]
05. # Get the last item in a list
06. last_nuc = nucleotides[-1]
07. # Looping through a list
08. for nuc in nucleotides:
09.     print(nuc)
10.
11. # Adding items to a list
12. nucleotides = []
13. nucleotides.append('A')
14. nucleotides.append('C')
15. nucleotides.append('G')
16. nucleotides.append('T')
17.
18. # range
19. # Making numerical lists
20. values = []
21. for x in range(1, 11):
22.     values.append(x**2)
23.
24. #range produces a sequence of integers from start (inclusive) to stop (exclusive) by step: range (start, stop, step)
25.
26. # length of list
27. print (len(values))
28.
29. # copying a list
30. values_copy = copy[:]
31.
32. # slicing
33. values= [1, 4, 9, 16, 25, 36, 49]
34. # slicing indices 1 to 3
35. values[1:3]
36. # slicing from 0 until 2
37. values[:3]
38. # slicing from 3 to end
39. values[3:]
40. # every three elements
41. values[::3]
42. # reversing a list
43. values[::-1]
44. #list comprehension
45. new_values = [x**2 for x in range(1,8)]
```

List comprehension

- The generation of new lists with elements that follow a mathematical or a logical concept is a frequent task in programming.
- List comprehension syntax: convenient way to create new lists from existing ones.

  General form:

  [expression for obj in iterable]

  It can also include a conditional statement:

  [expression for obj in iterable if condition]

```
01. # list of multiples of 10 smaller or equal than 200
02. multiples_ten = []
03. for x in range(1, 21):
04.     multiples_ten.append(x * 10)
05.
06. # with list comprehension
07. multiples_ten = [10 * x for x in range(1,21)]
08.
09. # extract all sub-strings of length 3 from a given sequence
10. seq = "ATGCTAATGTACATGCA"
11. seq_substrings = [(seq[x:x+3]) for x in range(0, len(seq)-2)]
```

Dictionaries

- Dictionaries represent hash tables and store connections between pieces of information. Each item in a dictionary is a key-value pair. The key provides the indexing and the value stores the associated value.

```
01. # A dictionary
02. info = {'color': 'blue', 'points': 5, 'weight': 75}
03. #Accessing a value
04. print("The color is " + info['color'])
05. #Adding a new key-value pair
06. info['height'] = 178
07.
08. #Looping through all key-value pairs
09. age = {'Rui': 17, 'Luis': 24, 'Isabel':32}
10. for name, number in age.items():
11.     print(name + ' is ' + str(number) + ' old')
12.
13. #Looping through all the values
14. for number in age.values():
15.     print(str(number) + ' years' )
```

Tuples and Sets

- Tuples represent a type of ordered sequences. The values are separated by commas and within the container (). They share several features of lists but are immutable.

```
01. # Create a tuple
02. t1 = (1, "a", 2, [10, 100, 100])
03. # slicing
04. t1[1]
05. # accessing elements
06. info = ("Rui", 24, "Master in Bioinformatics", "Porto")
07. name, age, course, city = info
```

- Sets are unordered sets of immutable elements. Particularly useful for testing membership and removing duplicates from lists. They implement the mathematical concept of set and the different operators on sets apply. Defined with **set()** operator.

```
01. a = set([1,2,3])
02. b = set([3,4,5])
03. a & b
04. a | b
05. a - b
```

If Statements

- If statements are used to test for particular conditions and respond appropriately.

Conditional tests:

```
# equals
x == 70
# not equal
x != 70
# greater than or equal to
x >= 70
# less than or equal to
x <= 70
# Conditional test with lists
'A' in nucleotides
'X' not in nucleotides

# Assigning boolean values
end_of_gene = True
promoter = False

# A simple if test
if len(sequence) >= 200:
    print("Enough length!")

# If-elif-else statements
if score < 50:
    grade = "Fail"
elif score >= 50 and score < 75:
    grade = "pass"
else:
    grade = "Very good"
```

Indentation

- In order to deliver more concise and clear coding Python adopts a set of indentation syntax rules. The indentation blocks are syntactically relevant and may affect the logic of the program. The rules are the following:
  - Code begins in the first column of the file.
  - All lines in a block of code are indented in the same way, i.e. aligned by a fixed spacing.
  - No brackets are required to delimit the beginning and the end of the block.
  - A colon (:) opens a block of code.
  - Blocks of code can be defined recursively within other blocks of code.

```
01. # Several nested blocks of code
02. if a > 10:
03.     if b > 100:
04.         print (b)
05.     else:
06.         if c < 1000:
07.             print (c)
08. else:
09.     if x == 10:
10.         print (x**2)
```

For and While loops

- A **while** loop repeats a block of code as long as a certain condition is true.

```
01. a = 0
02. while a < 10:
03.     print (a)
04.     a += 10
```

- A **for** loop is an iterative loop that is executed a fixed number of times. It scans through and iterable object (e.g. strings, lists or range).

```
1.  # iterate over a string
2.  seq = "ATATTCTAT"
3.  seq_len = 0
4.  for c in seq:
5.      seq_len += 1
6.  print ("len of sequence " + seq + " is " + str(seq_len))
7.
8.  # iterate over a range
9.  a = 0
10. for x in range(1, 11):
11.     a += x*10
```

User Input

- Programs can prompt the user for input. All input is stored as a string.

```
01. #Prompting for a value
02. sequence = input("Insert the sequence? ")
03. print(">" + sequence )
04.
05. #Prompting for numerical input
06. seq_len = input("Sequence length?")
07. seq_len = int(seq_len)
08. prot_pot = input("Protein potential? ")
09. prot_pot = float(prot_pot)
```

Working with files

- Python programs can read from files and write to files. Files are opened in read mode ('r') by default, but can also be opened in write mode ('w') and append mode ('a'). This is a three-step procedure:
  1. Open the stream to the file given its name and path to obtain a file handler and access the contents of the file.
  2. Read or write blocks or lines of text.
  3. Close the connection (file handler) to the file.

```
1.  # read file line by line and add a prefix to each line
2.  file_name = "test.txt"
3.  prefix = ">"
4.  with open (file_name) as fh:
5.      for line in fh:
6.          print (prefix + line)
7.  fh.close()
8.
9.  # write a line to end of file; open in append mode
10. file_name = "test.txt"
11. fh = open(file_name, "a")
12. fh.write("\nadd this line to end of file")
13. fh.close
14.
15. # write contents of list to a file; one per line
16. new_values = [str(x**2) + "\n" for x in range(1,8)]
17. file_name = "squares.txt"
18. fh = open(file_name, "w")
19. fh.writelines(new_values)
20. fh.close()
```

Functions

- An user can define its own functions. A function is simply defined by the **def** keyword, followed by the name, a list of arguments and a block of statements after the colon. The return keyword is used to return the result of the function.

```
01. # simple function
02. def print_seq():
03.     my_seq = "ACTGCT"
04.     print(my_seq)
05.
06. # function that receives a parameter and returns a value
07. def square(x):
08.     return x * x
```

Exceptions

- Python offers a **try-except** blocks to handle exceptions and unexpected behaviors during the execution of a program. If errors occurs this block structures offer an alternative way to handle these errors. There are nearly 50 exception types.

```
01.  x = 5
02.  y = 0
03.  try:
04.      r = x / y
05.  except ZeroDivisionError:
06.      print ("Division by zero detected")
07.  else:
08.      print ("ratio: " + r )
```

Built-in Functions

- Methods applicable to containers (lists, tuples, dictionaries,...):

Function	Description
len(*c*)	number of elements in container *c*
max(*c*)	maximum value from elements in container *c*
min(*c*)	minimum value from elements in container *c*
sum(*nc*)	sum of the numerical values in container *nc*
sorted(*c*)	list of sorted values in container *c*
value **in** *c*	membership operator **in**. Returns a Boolean value

Function	Description
range(*x*)	iterable object with *x* integer values from 0 to *x-1*
enumerate(*c*)	iterable object with *(index, value)* tuples
zip(*c1, c2, ..., cn*)	creates an iterable object that joins elements from *c1, c2, ... cn* to create tuples
all(*c*)	returns `True` if all elements in *c* are evaluated as true, and `False` otherwise
any(*c*)	returns `True` if at least one element in *c* is evaluated as true, and `False` otherwise

- Methods applicable to ordered sequence containers:

Function	Description
c * *n*	replicates *n* times the container *c*
c1 + *c2*	concatenates containers *c1* and *c2*
c.**count**(*x*)	counts the number of occurrences of *x* in container *c*
c.**index**(*x*)	index of the first occurrence of x in container *c*
reversed(*c*)	an iterable object with elements in *c* in reverse order

From Bioinformatics Algorithms, Rocha & Ferreira

```
01. # enumerate through list
02. for e in enumerate(["a","b","c"]):
03.     print (e)
04.
05. # create tuples from lists
06. for z in zip ([1,2,3],["a","b","c"], [7,8,9]):
07.     print (z)
08.
09. # test logical value
10. all([1, 1, 1])
11. any([0, 1, 0, 0])
12.
13. a = [1, 2, 3]
14. a * 3
15. b = [4, 5, 6]
16. ab = a + b
17. c.count(1)
18. c.index(3)
19. ra = reversed(a)
```

- Methods over lists:

Function	Description
lst.**append**(*obj*)	append *obj* to the end of *lst*
lst.**count**(*obj*)	count the number of occurrences of *obj* in the list *lst*
lst.**index**(*obj*)	returns the index of the first occurrence of *obj* in *lst*. Raises **ValueError** exception if the value is not present
lst.**insert**(*idx, obj*)	inserts object *obj* in the list in position *idx*
lst.**extend**(*ext*)	extend the list with sequence with all elements in *ext*
lst.**remove**(*obj*)	remove the first occurrence of *obj* in the list. Raises **ValueError** exception if the value is not present
lst.**pop**(*idx*)	removes and returns the element at index *idx*. If no argument is given, the function returns the element at the end of the list. Raises **IndexError** exception if list is empty or *idx* is out of range
lst.**reverse**()	reverses the list *lst*
lst.**sort**()	sorts the list *lst*

From Bioinformatics Algorithms, Rocha & Ferreira

Built-in Functions

- Methods over strings:

Function	Description
s.**upper**(), *s*.**lower**()	creates a new string from *s* with all chars in upper or lower case
s.**isupper**(), *s*.**islower**()	returns `True` when in *s* all chars are in upper or lower case, and `False` otherwise
s.**isdigit**(), *s*.**isalpha**()	returns `True` when in *s* all chars are digits or alphanumeric, and `False` otherwise
s.**lstrip**(), *s*.**rstrip**(), *s*.**strip**()	returns a copy of the string *s* with leading/trailing/both whitespace(s) removed
s.**count**(*substr*)	counts and returns the number of occurrences of sub-string substr in *s*
s.**find**(*substr*)	returns the index of the first occurrence of sub-string *substr* in *s* or −1 if not found
s.**split**(*[sep]*)	returns a list of the words in *s* split using *sep* (optional) as delimiter string. If *sep* is not given, default is any white space character
s.**join**(*lst*)	concatenates all the string elements in the list *lst* in a string where *s* is the delimiter

From Bioinformatics Algorithms, Rocha & Ferreira

- Methods over sets:

Function	Description
s.**update**(s2)	updates the set *s* with the union of itself and set *s2*
s.**add**(*obj*)	adds *obj* to set
s.**remove**(*obj*)	removes *obj* from the set. If *obj* does not belong to set raises an exception **KeyError**
s.**copy**()	returns a shallow copy of the set
s.**clear**()	removes all elements from the set
s.**pop**()	removes the first element from the set. Raises the exception **KeyError** if the set *s* is empty
s.**discard**(*obj*)	removes *obj* from the set *s*. If *obj* is not present in the set, no changes are performed

From Bioinformatics Algorithms, Rocha & Ferreira

- Methods over dictionaries:

Function	Description
d.**clear**()	removes all elements from dictionary *d*
d.**keys**()	returns list of keys in dictionary *d*
d.**values**()	returns list of values in dictionary *d*
d.**items**()	returns list of key-value pairs in *d*
d.**has_key**(k)	returns `True` if *k* is present in the list of keys, and `False` otherwise
d.**get**(*k*,*[defval]*)	returns the value corresponding to key *k*, or default value if *k* does not exist as key
d.**pop**(*k*,*[defval]*)	removes entry corresponding to key *k* and returns respective value (or default value if key does not exist)

From Bioinformatics Algorithms, Rocha & Ferreira

Lambda functions

- The lambda function can be used to create small, one-time and anonymous function.
- Basic syntax

  lambda arguments : expression

```
01. def add(x, y):
02.     return x + y
03.
04. print add(2, 3)
05.
06. # with lambda function
07. add2 = lambda x, y : x + y
08. print add2(2, 3)
```

Modules

- Modules provide a good way to keep code organized. They allow more efficient program maintenance and code reusability.
- The import statement can then be used to load to the current program specific functions or all the functions of the module. Once loaded, these functions can be called as if they were part of the program. Both modules developed in-house or by other developers can be imported.

```
01. # syntax for importing all functions from module
02. import module_name1
03.
04. # specific functions from a module can be imported instead of the entire module
05. from module_name import function_name1, function_nameX
```

- Whenever a Python script is run, the code from the imported modules is interpreted and executed. To prevent immediate execution of the imported code within a module, a conditional statement can be used. With the code below, when the module is run directly, the function main() is called and the respective code executed. When, on the other hand, the module is imported by some other program, the execution of the module's code is prevented. This feature is particularly useful for testing purposes.

```
01. if __name__ == "__main__":
02.     main()
```

- Modules and packages can be easily installed with package manager software: *pip*, *anaconda*, *setuptools* are examples of package managers.

Running code

- Once the code is written it should be saved in a Python script file, with a *.py* extension (e.g. my_script.py) It can then be run from command line as:

```
> python my_script.py
```

- Under Unix based operating systems the first line in our script can be used to indicate the path to the Python interpreter.

```
#!/path_in_my_os/bin/python
```

- The script can then be run from command line as:

```
> my_script.py
```

Exercises

- Install and explore the Jupyter Notebooks environment, running some of the examples.
- Install IDE (e.g. Spyder) and run some of the examples from the previous exercise.
- Write small programs, with the input-process-output structure, for the following tasks:
  - Reads a value of temperature in Celsius degrees (oC) and converts it into a temperature in Fahrenheit degrees (oF): F = 32 + 9C/5.
  - Reads a string and converts it to capital letters, printing the result.
  - Adapt the previous program to read the string from a file, whose name is entered by the user.
  - Reads a string and check if it is a palindrome, i.e. if it reads the same when it is reversed. Implement different versions using functions over strings, and cycles (for/while).
  - Reads several positive numerical values from the standard input (until a negative value is found), and calculates the largest and the smallest value.
  - Read an alphanumeric string (long enough) from the input and report the frequencies of each symbol in the sequence.
  - Given the two lower sides of a right-angled triangle calculate its hypotenuse.
  - Given a numeric interval defined by a lower and upper bound, calculate the sum of all integers included in that interval.
- Rewrite the previous code as functions and call them from your code.