Programming for Biologists

Manpreet S. Katari

Learning Objectives

- Quick history of Python
- How does Python compare to other programming languages.
- How to install and execute Python
- What are the parts of a Python program
- Different supported data types
- How to use the Python functions

What is Python?

A general-purpose high-level programming language with an emphasis on

- Code readability
- Coherence
- Software quality

Conceived in the late 1980s by Guido van Rossum

- First implementation released 1991
- Python 2.0 released Oct 2000
- Python 3.0 released Oct 2008

Why use Python?

- Software quality
  - Easily readable => reusable, maintainable
- Developer productivity
  - Efficient (little punctuation), no compile step => rapid prototyping
- Multi-paradigm
  - Support for “Object-oriented” and other programming styles
- Portable
  - Cross-platform

Why use Python?

- Support libraries
  - Comes with a “standard library” containing a large collection of pre-built functionality supporting many different application-level programming tasks
- Mixability
  - A variety of options for integrating Python code with components in other languages (e.g. C, C++, Java), different frameworks and network interfaces
- Extensions for numeric and scientific programming

Why use Python?

- Open source
  - Easy to learn and use
  - Growing popularity for bioinformatics (R is another very widely used language for bioinformatics/computational biology)
- Other languages used in bioinformatics:
  - C, C++: fast, used for computation (BLAST, FASTA, ClustalW), statically typed, so not as interactive as Python (which is a dynamically-typed language)
- R: interpreted, dynamic language optimised for data analysis and statistics

Python Features

- Toolbox somewhere between traditional scripting languages and systems development languages
  - Ease of use vs. software engineering tools
- Dynamic typing
- Automatic memory management / garbage collection
- Large-scale programming support
  - Modules, classes, exceptions
  - Component reuse and customization
  - Graceful event and error handling

Python Features

- Built-in object types
  - flexible data structures
  - Built-in tools
  - Convenient operation for object processing: concatenation, slicing, sorting, mapping
- Library utilities
  - From regex matching to networking
- Third-party utilities
  - XML, database access, much more

Installing Python

- The official download for python is (https://www.python.org/downloads/)
- However we recommend Anaconda by Continuum (https://docs.anaconda.com/anaconda/install/)
  - It makes it easy to install and maintain python packages and other bioinformatics tools.
  - We recommend installing the regular version (not miniconda) which ships with simple IDEs for example the notebook.
- We will be coding Python version 3.
  - There are still some applications that run using python 2, so be careful when you look for help on the web.

Running Python Programs

- Interactive (ipython and jupyter notebooks)
  - Get results right away.
- Running on command line
  - Executes a file containing a list of commands.
  - Normally have a .py extension
- Popular IDEs for developing python programs
  - Visual Studio Code
  - Pycharm
  - Sublime Text
  - Atom
  - Jupyter notebooks can be executed line by line or all together.

How Python runs prorams

- The .py script is the **source code**.
- The source code is compiled to **byte code**, a .pyc file
- A PVM, a python virtual machine, executes the byte code.
- Python does both steps at once, unlike Some lower level languages such as Java and C++, which require you to execute both steps separately.

Python terminology

- A python program often consists of several files.
- Each file is a **module**.
- Each module has **statements**.
- Each statement has **expressions**.
- Each expression results in a **value**.
- Value is data
- We save value(data) in memory using **variables**.
- Python is object oriented programming, which means every statement involves an operation on an object.
  - We will discuss this in more detail later in the semester.

Different types of numbers

The two major types are:

- Integers - whole number, no decimals, quick and precise calculations
  - In Python version 2, an integer divided by an integer always gave you an integer. This was a problem in division. This is no longer the case in python version 3
- Floating - any number with decimals.
  - Floating point error. Try 0.1 + 0.2 The result is an estimate of the value which makes floating values not precise, but very close.

Strings

Text strings can be represented with single, double, and triple quotes.

- ‘Hello world’
- “Who’s there”
  - You can use one type of quote inside the string and surround it with the other.
- ’’’Hello
  World’’’
  Triple quotes allow you to have new line character in strings
- Escape characters can be used to introduce special characters using \ before it

Link to Google colab

Google colab

Different types of collections

- Sets
  - Unordered collection of unique items
  - Individual access not allowed
  - E.g. set([1,1,2,2,3,3,4,4]) is the set of [1,2,3,4]
- Sequences
  - Ordered collections of items
  - Use numerical indices to access values
  - E.g. a[0] = 2**3; a[5]='Hi'; a=['A','C','G','T']

Different types of collections

- Mappings
  - Unordered collection of items
  - Use keys to access values
  - E.g. a = {'A':'adenine','C':cytosine}
- Streams
  - Sequences of input, such as text from keyboard or a file
  - Objects are handled in different modes (e.g. read, write…)

Operations for ALL collections

Operation	Returns
x in coll	True if coll contains x
x not in coll	True if coll does not contain x
any(coll)	True if any item in coll is true, otherwise false
all(coll)	True if every item in coll is true, otherwise false
len(coll)	The number of items in coll (not supported by streams)
max(coll, key=function)	Maximum item in coll which may not be empty
min(coll, key=function)	Minimum item in coll which may not be empty
sort(coll[, keyfn][, reverseflag])	A list containing the elements of coll, sorted by comparing elements

Sets

- A **set** is an unordered collection of items that contains no duplicates.
- **frozenset** is an immutable set.
- You create a set by calling it as a function set() or use braces:

```
set("TCAGTTAT")
```

```
{'A', 'C', 'G', 'T'}
```

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
basket
```

```
{'apple', 'banana', 'orange', 'pear'}
```

Set comparison operations

Operation	Method	Returns
	set1.isdisjoint(coll)	True if the set and the collection argument (coll) have no elements in common
set1 <= set2	set1.issubset(coll)	True if every element of set1 is also in set2 (coll in method)
set1 < set2		True if every element in set1 is also in set2 (coll) and set2 is larger than set1
set1 >= set2	set1.issuperset(coll)	True if every element of set2 (coll) in method is also in set1
set1 > set2		True if every element of set2 is in set1 and set1 is larger than set2
set1 | set2	set1.union(set2)	New set with elements from both sets
set1 & set2	set1.intersection(set2)	New set with elements common to both sets
set1 - set2	set1.difference(set2)	New set with elements in set1 but not in set2
set1 ^ set2	set1.symmetricdifference(set2)	New set with elements in either set1 or set2 but not both

Set update operations (these will not work if set1 is a frozenset)

Operation	Method	Returns
set1 |= set2	set1.update(set2)	Updates set1 by adding the elements in set2
set1 &= set2	set1.intersection_update(set2)	Updates set1 to keep only the elements that are in both set1 and set2
set1 -= set2	set1.difference_update(set2)	Updates set1 to keep only the elements that are in set1 but not in set2
set1 ^= set2	set1.symmetric_difference_update(set2)	Updates set1 to keep only the elements that are in either set1 or set2
	set1.add(item)	Add item to set1
	set1.remove(item)	Remove item from set1; raises an error if item is not in the set
	set1.discard(item)	Removes item from set1 if present; no error if not present

Sequences

- Sequences are ordered collections that may contain duplicate elements
  - Since they are ordered, they can be referenced by position (index)
- There are different types of sequences:
  - str - strings (text), not mutable
  - bytes - 8 bit bytes, not mutable
  - bytearray - mutable 8 bit bytes
  - range - integers, non mutable, uses little memory
  - tuple - any type, non mutable
  - list - any type, mutable

Summary of sequence slicing

Operation	Returns
seq[i:j]	Elements of seq from i up to but not including j
seq[i:]	Elements of seq from i through end of seq
seq[:j]	Elements of seq from first up to but not including j
seq[:-1]	Elements of seq from first up to but not including last
seq[:]	All the elements of seq
seq[i:j:k]	Every kth element of seq, from i up to, but not including j If k is negative the steps go in reverse

Some useful functions and methods for str

Operation	Returns
str()	Returns an empty string
str(obj)	Printable representation of obj
str1.isalpha()	True if str1 is not empty and all characters are alphabetic
str1.numeric()	True if str1 is not empty and all characters are numeric
str1.isupper()	True if string contains at least one “cased” character and all “cased” characters are upper case, else False
str1.startswith(str2[,startpos, [endpos]])	Returns true if str1 starts with str2
str1.find(str2[,startpos, [endpos]])	Returns lowest index at which str2 is found, else returns -1
str1.count(str2[,startpos, [endpos]])	Returns the number of occurrences of str2 in str1
str1.upper()	Returns a string with all of its characters as uppercase

Some useful functions and methods for Ranges (series of integers)

Operation	Returns
range(stop)	Creates a range representing integers from zero up to, but not including, stop.
range(start,stop)	Creates a range from start up to but not including stop
range(start, stop, step)	Creates a range from start up to but not including stop, in increments of step.

```
list(range(5))
```

```
[0, 1, 2, 3, 4]
```

```
list(range(0,5))
```

```
[0, 1, 2, 3, 4]
```

```
list(range(0,5,2))
```

```
[0, 2, 4]
```

Tuples (immutable sequence of any type)

- Commonly used as simple representation of x,y coordinates.
- A one-element tuple must be written with a comma after the single element.

Tuples

```
('TGAC','UCAG')   # a two element tuple
```

```
('TGAC', 'UCAG')
```

```
('TGAC',)         # a one element tuple
```

```
('TGAC',)
```

```
()                # an empty tuple
```

```
()
```

```
('TGAC')          # not a tuple; it's a string!
```

```
'TGAC'
```

```
tuple('TGAC')     # convert a string to tuple,
                  # becomes a tuple of characters+
```

```
('T', 'G', 'A', 'C')
```

Lists (mutable sequences of any type)

- Anything you can do with a tuple, you can do with a list, but there's a lot more you can do with a list.
- Index and slice expressions of lists can appear on the left-hand side of an assignment, where they are replaced by the expression on the right-hand of the statement.

Lists (mutable sequences of any type)

Operation	Returns
lst[n]=x	Replace the nth element of lst with x
lst[i:j]=coll	Replace the ith through jth elements of lst with the elements of coll
lst[i:j]=any_empty_container	Delete the ith through jth elements of lst
lst[n:n]=coll	Insert the elements of coll before the nth element of lst
lst[len(lst):len(lst)]=[x]	Add x to the end of lst
lst += coll	Add the elements of coll to the end of lst

Operation	Returns
del lst[n]	Remove the nth element form lst
del lst[i:j]	Remove ith through jth element of lst
del lst[i:j:k]	Remove every kth element of from i up to j from lst
lst.append(x)	Add x to end of lst
lst.extend(x)	Add elements of x to lst
lst.insert(i, x)	Insert x before the ith element of lst
lst.remove(x)	Remove the first occurrence of x from lst
lst.pop([i])	Remove ith element of lst; if i is not specified, remove the last element
lst.reverse	Reverse the list
lst.sort([reverseflag[,keyfn])	Sort the list by comparing elements. If keyfn is provided, then comparison is done based on it. If reverse flag is True, then reverse sort is performed.

```
list1 = [1,2,3]
list1
```

```
[1, 2, 3]
```

```
list2 = [4,5]
list2
```

```
[4, 5]
```

```
list1 + list2 # concatenates list1 and list2
              # but does not modify either list
```

```
[1, 2, 3, 4, 5]
```

```
list1 # list1 is the same as before
```

```
[1, 2, 3]
```

```
list1.extend(list2) # list1 modified directly by extend
list1
```

```
[1, 2, 3, 4, 5]
```

```
list2 # ...but not list2
```

```
[4, 5]
```

```
list1 += list2 # the plus-equals operator
list1 # also appends items to list1
```

```
[1, 2, 3, 4, 5, 4, 5]
```

```
list2 # ...but list2 still stays the same
```

```
[4, 5]
```

Mappings

- A mapping is a mutable unordered collection of key/value pairs.
  - Also known as associative arrays, or hash tables
- A dictionary is the only mapping type in python.
- When using the dict function to create a dictionary, the elements must be tuples (lists of two elements).
- Or you can simply use the curly brackets with a comma separating each key/value pair, and the elements of each pair by a colon.

Mappings

```
dict(( ('A', 'adenine'),
      ('C', 'cytosine'),
      ('G', 'guanine'),
      ('T', 'thymine')
      ))
```

```
{'A': 'adenine', 'C': 'cytosine', 'G': 'guanine', 'T': 'thymine'}
```

```
# note that the following is a set, not a dict:
{'A','adenine','C','cytosine','G','guanine','T','thymine'}
```

```
{'A', 'C', 'G', 'T', 'adenine', 'cytosine', 'guanine', 'thymine'}
```

Operation	Returns
d[key]	Value associated with key
d[key]=value	Associates value with key (either by adding a new key/value pair or replacing value associated with key
d[key] .= value	Augment assignment, error if value does not exist or value is not numeric
del d[key]	Delete key from dictionary, error if key does not exist
d.get(key[,default_value])	Similar to d[key] but if key does not exist it returns default value (None)
d.setdefault(key[,default_value])	Like d[key], if key does not exist,adds default value to key, which is None
d.pop(key[,default_value])	Like del d[key] but does not cause error if key does not exist, instead returns default_value
d1.update(d2)	For each key in d2, sets d1[key] to d2[key], replacing the values
d.keys()	Sequence like object containing keys
d.values()	Sequence like object containing values
d.items()	Sequence like object containing (key,value) tuples