ICQB

Introduction to Computational & Quantitative Biology (G4120)
Fall 2022

Oliver Jovanovic, Ph.D.
Columbia University
Department of Microbiology & Immunology

Lecture 5: Introduction to Python and Biopython
October 11, 2022

Python

The Python programming language was released in 1991 after two years of development by Guido van Rossum, a Dutch programmer. It is distinguished by its emphasis on simplicity and readability of code, and uses whitespace indentation to delimit blocks of code.

Python is an interpreted general purpose programming language, and supports a variety of programming paradigms, including object-oriented, structured, functional and procedural programming.

The reference implementation of Python is written in C and called CPython, and is free and open source, managed by the non-profit Python Software Foundation, and supported by a large community of open source developers. The Python Package Index (PyPi), which serves as a repository for free third party Python software, currently contains over 350,000 packages.

Python has become one of the world's most popular programming languages, used heavily at Amazon, CERN, Facebook, Google and NASA, widely taught in introductory computer science courses, and is heavily used in bioinformatics.

Python 2 and Python 3

Python 2	**Python 3**
`raw_input()`	`input()`
`print "Hello"`	`print("Hello")`
`5/2 = 2`	`5/2 = 2.5`
`xrange()`	`range()`
`ASCII strings`	`UNICODE strings`

These are some of the notable differences between the versions, but others exist as well. **Python 3 is not backward compatible with Python 2.**

Python 2 is now the legacy form of the language. However, it is quite pervasive in older code and computers, and as it can take significant effort to port Python 2 code to Python 3, a large body of legacy Python 2 code is likely to continue to exist and be used.

Note that in the most recent version of macOS, Monterey, Python 3 is installed as `python3`, needs to be referenced as such to use (e.g. `#!/usr/bin/python3`).

Python Standard Library

Python features an extensive standard library that can be called to provide additional functionality using an import statement. Some potentially useful functions for bioinformatics include:

itertools
Fast, efficient looping iterator functions.

math
Basic mathematical functions.

random
Generates pseudo-random numbers.

re
Regular expression matching operations (similar to Perl).

string
Provides additional string functions and classes, including some legacy functions (note, **StringIO** can also be useful when you want to read and write large strings in memory).

sys
System specific parameters and functions (including reading command line arguments).

Python Standard Parsers

argparse
A command line option, argument and sub-command parser. See **https://docs.python.org/3/library/argparse.html** for details.

csv
A CSV file parser. See **https://docs.python.org/3/library/csv.html** for details.

fileinput
Allows for quickly looping over standard input or a list of files. See **https://docs.python.org/3/library/fileinput.html** for details.

sqlite3
A simple interface to SQLite.

urllib.parse
Parses URL strings into their components (in some cases, may need to also use **str.split**).

xml.dom.minidom
A minimal implementation of the Document Object Model interface, useful for parsing XML or SMBL files.

xml.etree.ElementTree
A simple method for parsing and storing hierarchical data structures in memory, including XML documents.

Python Input and Output

Keyboard input to a Python 3 program can be obtained using the `input` function, e.g. `i = input()` which returns whatever the user typed up to pressing **return** as a string. Numerical input has to be converted by type casting, e.g. `i = int(i)`. In Python 2, the `raw_input` function is used for strings and `input` is used for integers.

Use the `open` function to open a file object for reading (by default, `'r'`), overwriting (`'w'`), or appending (`'a'`). Once done, use the `close` method to close the file. The `readline` method reads a single line including any newline character, while `readlines` reads all the lines in a file, and returns them as a list of strings, The `write` method writes a single string (which can include newline characters) to a file, while `writelines` writes a list of strings to a file:

```
file_object = open("anybody.txt", 'w')
file_object.write("Is there anybody out there?")
file_object.close()
```

The optional `fileinput` module allows for quickly looping over standard input or a list of files:

```
import fileinput
for line in fileinput.input()
    pass    # A_placeholder_function
```

Python Command Line Arguments

Command line arguments are the values, separated by spaces, that are passed when calling a script along with the calling statement, e.g. the name of a file for the script to act upon.

`sys` **module**

The `sys` module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter, including `sys.argv`, an array for command line arguments.

`sys.argv`

Provides a list of command line arguments passed to a Python script as an array. `argv[0]` is the script name, subsequent arguments can be read as `argv[1]`, `argv[2]`, etc. Note that if the command was executed using the `-c` command line option to the interpreter, `argv[0]` is set to the string `'-c'`, and if no script name was passed to the Python interpreter, `argv[0]` is the empty string. To use `sys.argv`, first `import sys`.

Python 2 Command Line Output

```
#!/usr/bin/python

import random
import sys

def DNA(length):
    return ''.join(random.choice('acgt') for _ in xrange(length))

if len(sys.argv)==2:
    length = abs(int(sys.argv[1]))
    print(DNA(length))
elif len(sys.argv)==3:
    length = abs(int(sys.argv[1]))
    filename = sys.argv[2]
    fo = open(filename, 'w')
    fo.write(DNA(length) + "\n")
    fo.close()
else:
    length = input ("Enter length of random DNA sequence to generate: ")
    filename = raw_input ("Enter filename to save random DNA sequence to: ")
    fo = open(filename, 'w')
    fo.write(DNA(length) + "\n")
    fo.close()
```

Python 3 Command Line Output

```
#!/usr/bin/python3

import random
import sys

def DNA(length):
     return ''.join(random.choice('acgt') for _ in range(length))

if len(sys.argv)==2:
     length = abs(int(sys.argv[1]))
     print(DNA(length))
elif len(sys.argv)==3:
     length = abs(int(sys.argv[1]))
     filename = sys.argv[2]
     fo = open(filename, 'w')
     fo.write(DNA(length) + "\n")
     fo.close()
else:
     length = input ("Enter length of random DNA sequence to generate: “)
     length = int(length)
     filename = input ("Enter filename to save random DNA sequence to: ")
     fo = open(filename, 'w')
     fo.write(DNA(length) + "\n")
     fo.close()
```

Python 2 Command Line Input

```
#!/usr/bin/python

from string import *
import sys

def count_gc(dna):
      count_g = count(dna, 'g')
      count_c = count(dna, 'c')
      dna_length = len(dna)
      percent_gc= 100 * float (count_g + count_c) / dna_length
      return percent_gc

if len(sys.argv)==2:
      filename = sys.argv[1]
      with open(filename) as x: dna = x.read()
      print count_gc(dna), "percent GC in file"
else:
      dna = raw_input ("Enter a lowercase DNA sequence: ")
      print count_gc(dna), "percent GC entered"
```

Python 3 Command Line Input

```
#!/usr/bin/python3

import sys

def count_gc(dna):
      count_g = str.count(dna, 'g')
      count_c = str.count(dna, 'c')
      dna_length = len(dna)
      percent_gc= 100 * float (count_g + count_c) / dna_length
      return percent_gc

if len(sys.argv)==2:
      filename = sys.argv[1]
      with open(filename) as x: dna = x.read()
      print (count_gc(dna)), "percent GC in file"
else:
      dna = input ("Enter a lowercase DNA sequence: ")
      print (count_gc(dna)), "percent GC entered"
```

Documenting Python

Comments

Comments in Python start with a # and a single space. They should be indented to the same level as the code, and can span multiple lines. Inline comments should be used sparingly.

```
# This is a Python comment.
```

Documentation Strings

The string that appears as the first statement in a module, function, class or method definition in Python is a documentation string, or doctoring. It becomes the __doc__ special attribute of that object. By convention, triple double quotes should be used on each side of a docstring. Docstrings spanning multiple lines should start with a one line summary, followed by a blank line, followed by the rest.

```
def spam_filter():
    """Docstring for spam_filter, describes the function."""
```

Documentation Systems

For larger projects, using a Python documentation generator such as Sphinx (**http://www.sphinx-doc.org**), which uses reStructuredText markup language, can be helpful.

Python Testing

When developing large or complex software packages, automated software testing procedures can save a great deal of time and effort. Unit testing involves testing individual units of code with a set of appropriate test cases.

doctest
Python features a simple automated Python session testing framework called doctest which searches for examples of tests embedded in docstring documentation, runs them, and verifies the results.

unittest
Python features a full unit testing framework called unittest, which loads and runs individual test cases or suites of tests, then reports the results. It is particularly suited for use with large, complex projects.

Installing Python Packages

Installing a large Python package manually can be a complex procedure, as many pieces may need to be installed in specific locations, the Python search path needs to be correctly configured, portions of the package may need to be correctly compiled, and the package may have certain dependencies (other packages that need to be properly installed for it to function). Thus, it is often far easier to use a prepackaged installation, or a Python package management system installer, such as **pip,** which automates installing packages from the Python Package Index (PyPI).

pip installation
In the most recent version of macOS, Monterey, **pip** is preinstalled as `pip3`. You can simply install packages from PyPI by running `pip3 install package_name` (you may need to use `sudo`, e.g. `sudo pip3 install biopython`). **pip** will attempt to resolve dependencies and download and install any other required packages. **pip** often comes preinstalled as `pip` (instead of `pip3`). It can also be downloaded and installed from **https://pip.pypa.io**.

List installed packages: `pip3 list`
Search for a package: `pip3 search query_string`
Install a package: `pip3 install package_name`
Uninstall a package: `pip3 uninstall package_name`
Show installed package details: `pip3 show package_name`
List outdated packages: `pip3 list --outdated`
Upgrade an installed package: `pip3 install --upgrade package_name`
Upgrade to the latest version of pip: `python3 -m pip install --upgrade pip`

Python and Bioinformatics

iPython
An enhanced interactive shell for Python programming: **ipython.org**

NumPy and SciPy
Scientific computing packages for Python: **www.numpy.org**

matplotlib
A simple 2D plotting library for Python: **matplotlib.org**

Cython
Allows you to embed compiled optimized bits of C or C++ code in a Python program: **cython.org**

SQLAlchemy
A SQL toolkit and object relational mapper for SQL databases in Python: **sqlalchemy.org**

Django
A rapid back end web development framework for Python: **www.djangoproject.com**

Pandas
A high-performance data structure and data analysis toolkit for Python: **pandas.pydata.org**

Biopython
A bioinformatics and biological computation toolkit for Python: **biopython.org**

NumPy and SciPy

A set of packages that add expanded scientific computing capabilities to Python including:

Fast N-dimensional array objects

Defining and storing arbitrary data types

Database integration

Tools for C, C++ and Fortran code integration

Linear algebra, Fourier transform and random number generation functions

Statistical functions and other mathematical routines, solvers and optimizers

Source: https://www.numpy.org

Pandas

Pandas provides a set of particularly powerful data structures and functions for working with structured data. It is named after **panel data,** which in statistics and econometrics refers to multi-dimensional data that frequently changes over multiple time periods.

DataFrames
The primary data structure in pandas is a **DataFrame**, a two dimensional column oriented structure with row and column labels that can be thought of as a table of data, similar to the R programming language **data.frame** object. Pandas also supports one dimensional array like structures called a **Series**, containing an array of data and an associated array of labels.

Pandas allows for data to be loaded into very large **DataFrame** structures and quickly and efficiently manipulated in a variety of ways: cleaned, transformed, merged, reshaped, pivoted, etc. It also offers high-level plotting functions that supplement those offered by **matplotlib**, and simplifies the visualization of large, complex data sets.

Source: https://pandas.pydata.org

Biopython

Biopython is an extensive package of Python tools, classes and functions for bioinformatics and computational biology. It was first released in 2000, and now contains over 300 modules for dealing with biological data. The current version, 1.79, was released in June of 2021, and requires Python 3.6 or later. A previous version, 1.76, supports Python 2.7 to 3.5.

In Biopython, sequence data is represented by a **Seq** class, which includes biological sequence methods such as **transcribe** or **translate**, and specifies the sequence alphabet used. The **SeqRecord** class describes sequences, with features described by **SeqFeature** objects.

Biopython handles importing and exporting biological data from a wide variety of formats, including Clustal, DNA Strider, FASTA, GenBank, mmCIF, Newick, NEXUS, PDB, PHYLIP and phyloXML using **Bio.SeqIO** and other modules. The **Bio.Entrez** module can download and import data directly from various NCBI databases. Phylogeny data can be imported into **Tree** and **Clade** objects and traversed and analyzed using the **Bio.Phylo** module. Molecular structure data can be imported into **Structure** objects and examined and analyzed using the **Bio.PDB** module.

Other Biopython features include a **GenomeDiagram** module for visualizing sequence and genome data, a **Bio.PopGen** module for interacting with Genepop, support for the BioSQL model and schema, and a number of command line wrappers which allow for Python interaction with commonly used bioinformatics tools such as BLAST, Clustal and EMBOSS.

Source: http://www.biopython.org

Object-oriented Programming (OOP)

In object-oriented programming (OOP), a class serves as a blueprint for creating an instance called an object. The class defines the data and behavior of the object. Each object created from a class can have its own set of properties.

Properties are applied to variables inside a class.

For example, the BioPython `Seq` class is defined as:
`class Bio.Seq.Seq(data, alphabet=Alphabet())`
An object derived from this class will contain a string with an `alphabet` **property**, which defines whether it is DNA, RNA or protein.

Methods define the behavior of a class.

For example, the BioPython `Seq` class has built in **methods** for common sequence operations such as: `complement(self)`, `reverse_complement(self)`, `transcribe(self)`, `back_transcribe(self)` and `translate(self, table='Standard', stop_symbol='*', to_stop=False, cds=False, gap=None)` as well as standard string manipulation methods. Depending on the alphabet **property**, not all the methods may be available, e.g. `transcribe(self)` is limited to DNA sequences. Dot notation is used to access methods, e.g. `seq.transcribe()`.

Key Concepts of Object-oriented Programming

Inheritance
Objects of one class can derive their behaviors from another class. When a class inherits from another, the inheriting child class is considered a subclass, and the parent class it inherits from is considered its superclass. Python allows for multiple inheritance, where objects of one class can derive behavior from multiple base classes.

Polymorphism
Objects of different classes can be used interchangeably. When the same interface can be used for different data types and functions, it greatly simplifies programming. In Python, all classes inherit from the object class implicitly, and the language supports Method Overriding, which allows you to modify methods in a child class inherited from a parent.

Encapsulation
Objects keep their internal data private. Instead of directly manipulating an object's data, other objects send requests to the object, in the form of messages, which the object may respond to by altering its internal state. This practice can simplify programming. Python supports encapsulation, but does not strictly enforce it.

Basic Biopython

```
pip3 install biopython
```

```
python3
```

```
>>> from Bio.Seq import Seq

>>> my_seq = Seq("ATGCATTAG")

>>> print (my_seq)

>>> print (my_seq.complement())

>>> print (my_seq.reverse_complement())

>>> print (my_seq.translate())
```

Biopython and Sequences

```
#!/usr/bin/python3

from Bio import SeqIO
from Bio.SeqUtils import GC

for sr in SeqIO.parse ("test.fasta", "fasta"):
   print (sr.id)
   print (repr(sr.seq))
   print (len(sr))
   print (sr.seq)
   print (GC(sr.seq))
   print (sr.seq.transcribe())
   print (sr.seq.translate())
   print (sr.seq.translate(to_stop=True))
```

bpfasta.py

Biopython and Parsing

genbank.py

```
#!/usr/bin/python3
from Bio import Entrez
Entrez.email = "mi@columbia.edu"
handle = Entrez.efetch(db="nucleotide", rettype="gb", retmode="text",
id="2765658")
save_file = open("2765658.gbk", 'w')
save_file.write(handle.read())
handle.close()
save_file.close()
```

convert.py

```
#!/usr/bin/python3
from Bio import SeqIO
SeqIO.convert("2765658.gbk", "genbank", "2765658.fasta", "fasta")
```

parsefasta.py

```
#!/usr/bin/python3
from Bio import SeqIO
recs = SeqIO.parse("cosmids1.fasta", "fasta")
for rec in recs:
    print (rec.id)
```

Python for RNA-seq

HTSeq is a Python based framework for processing and analyzing data from high-throughput sequences assays, e.g. RNA-seq. Some of the functions HTSeq can perform include:

- Quality assessment of sequencing runs by providing statistical summaries of quality scores and plotting base calls and base-call qualities by position in the read.
- Reading annotation data from General Feature Format (GFF) files.
- Counting how many reads cover a particular section of a chromosome or genome and plotting this data.
- Counting how many reads fall into the exon regions of each gene in a RNA-seq run.

Source: https://htseq.readthedocs.io/

Anaconda

Anaconda is a free open source data science platform powered by the Python and R programming languages that includes over 100 of the most popular packages for data science, including NumPy, Pandas, SciPy, Matplotlib and the Jupyter Notebook.

Anaconda includes the **conda** package, dependency and environment manager, which can easily install over 1,000 additional data science packages in a variety of languages, as well as the **pip** package manager. Anaconda also includes a graphical user interface, Anaconda Navigator, and supports a variety of Integrated Development Environments (IDEs) including Eclipse/PyDev and Spyder.

Anaconda allows you to run multiple versions of Python in isolated environments. It easily allows you to run Python 3 on older Macs alongside Python 2.

Source: https://www.anaconda.com

Anaconda, Biopython and BLAST

```
#!/usr/bin/python3

from Bio.Blast import NCBIWWW
result_handle = NCBIWWW.qblast("blastn", "nt", "8332116")
from Bio.Blast import NCBIXML
blast_record = NCBIXML.read(result_handle)
E_VALUE_THRESH = 0.04
for alignment in blast_record.alignments:
    for hsp in alignment.hsps:
        if hsp.expect < E_VALUE_THRESH:
            print('\n****Alignment****')
            print('*Sequence:', alignment.title)
            print('*Length:', hsp.align_length)
            print('*Identities:', hsp.identities)
            id = (100.00 * hsp.identities / hsp.align_length)
            print ('*Precent identity:', id)
            print('*E-value:', hsp.expect)
            print(hsp.query[0:75] + '...')
            print(hsp.match[0:75] + '...')
            print(hsp.sbjct[0:75] + '...')
```

blast3.py

Jupyter Lab and Notebook

Jupyter Lab and Notebook are free open source web applications that let you create and share documents that contain live code, data visualizations, text and equations. Jupyter fully supports both Python and R, and is particularly useful for interactive scientific programming and visualization.

Anaconda includes Jupyter Notebook. Once Anaconda is installed, Notebook can be run from Terminal (on Macs) or Command Prompt (on Windows) by typing `jupyter notebook`. It can also be installed using **pip**, but installing with Anaconda instead is highly recommended.

Source: https://jupyter.org

References

Python 3 Documentation free at:
https://docs.python.org/3/

Biopython Tutorial and Cookbook free at:
http://biopython.org/DIST/docs/tutorial/Tutorial.html

Biopython Documentation free at:
https://biopython.org/wiki/Documentation

Introduction to Computation and Programming Using Python
by John V. Guttag

Python for Data Analysis: Data Wrangling with Pandas, NumPy and iPython
by Wes McKinney