Guide to Programming with Python

Yuzhen Ye (yye@indiana.edu)
School of Informatics and Computing, IUB

Objectives

- Python basics
  - How to run a python program
  - How to write a python program
- Variables
  - Basic data types: number, string, list, and dictionary
- Control flow:
  - Branching structure: if, if-else, if-elif-else
  - Loops: for loop and while loop
- Write a function & a class
- I/O (read from files and write to files)
- Write python programs with graphical user interface (GUI)
- Comments & modules (e.g., random module)
- Your first bioinformatics program in python

Algorithms & Computer programs

- An algorithm is a set of well-defined instructions for accomplishing a task (e.g., S’mores algorithm for making s’mores)
- When we write computer program, we are generally implementing a method (an algorithm) devised previously to solve some problem.
- A computer program is a sequence of instructions that are executed by a CPU
- Computer programs can be written in high-level (e.g., Python, Perl, C, C++, Java), or primitive programming languages

Hello World

- Python is an interpreted (scripting), high-level language (different from compiled programming language, such as C)
- "Hello World" program: By tradition, prints "Hello, world!"
  - Often used as first program
  - print "Hello world"
- Console window: Provides a text-based interface to Windows operating system
- Terminal application: Provides a text-based interface to Mac OS X and Linux operating systems

Setting up Python

- Linux
  - Python probably already installed
  - Test: try running python at command prompt
- Mac OS
  - IDLE, an integrated development environment for Python
  - Or from the terminal
- Remember: your computer needs a python "interpreter" to understand python codes

Introducing IDLE/vi/Emacs

- Integrated Development Environment (IDE):
  - Application that helps software developers write programs
    - Like a word processor for your code
- IDE that ships with Python
- Has two "modes": Interactive and Script (see demo)
- You need to be able to use vi or Emacs
### Interactive and Script Mode
- **Iterative mode:** You tell Python what to do, and it will do it immediately.
- **Script mode:** You write, edit, load, and save python programs (just like you write a document in Word processor, and any other types of text processors). But definitely you do not want to write your Python codes with Word!

### Start to Talk Like a Pro
- **Statement:** Single unit in programming language that performs some action
  - `print "ACGT"`
- **Expression:** Something which has a value or that can be evaluated to a single value
  - "ACGT"
  - 7 + 2
- **Code:** Sequence of programming statements
  - **Syntax error:** Error in the rules of usage; often a typo (versus logic error)
  - **Bug:** Error in programming code

### 1. Variables
- **Variable:** Represents a value; provides way to get at information in computer memory
- **Variables** allow you to store and manipulate information
- You can create variables to organize and access this information
- **Assignment statement:** Assigns a value to a variable; creates variable if necessary
  - `name = "E.coli"`
  - Stores string "E.coli" in computer memory
  - Creates variable `name`, which refers to "E.coli"

### Naming Variables
- **Rules for legal variable names**
  - Can contain only numbers, letters, and underscores
  - Can’t start with a number
  - Can’t be a keyword
- **Keyword:** Built-in word with special meaning
- **Legal Names**
  - `enzyme, dna, prot, player2, max_health`
- **Illegal Names**
  - `again, 2nd_player, print`

### Naming Variables (continued)
- **Guidelines for good variable names**
  - Choose descriptive names; `score` instead of `s`
  - Be consistent: `high_score` or `highScore`
  - Follow traditions; Names that begin with underscore have special meaning
  - Keep the length in check; `personal_checking_account_balance` - too long?
  - **Self-documenting code:** Code written so that it’s easy to understand, independent of any comments

### 2. Using the Right Types
- **Python** does not need to specify the type of a variable in advance (by contrast, C does)
- **Python:**
  - `protlen = 100`
  ```python
  C:
  int protlen = 10;
  ```
- **Important to know which data types are available**
- **Equally important to know how to work with them**
- **If not, might end up with program that produces unintended results**
- **Converting values:** e.g., `int("3") = 3`
2.1 Working with Strings

- Using quotes inside strings
  - Define with either single (’) or double quotes ("")
  - "ribosomal RNA" or "ribosomal RNA"
  - Define with one type, use other type in string
    - "Program 'Game Over' 2.0"
- Triple-quoted strings can span multiple lines
  - """I am a triple-quoted string"
- Line-continuation character \n
2.2 Working with Numbers

- Numeric types
  - Integers: Numbers without a fractional part
    - 1, 0, 27, -100
  - Floating-Point Numbers (or Floats): Numbers with a fractional part
    - 2.376, -99.1, 1.0
- Addition, Subtraction and Division
  - print 2000 - 100 + 50 displays 1950
- Modulus (remainder of integer division)
  - print 107 % 4 displays 3

2.3 Working with Lists (and Tuples)

- List: A mutable (changeable) sequence of any type
- Creating List
  - bases = ['A', 'T', 'C', 'G']
  - Tuple: bases=('A', 'T', 'C', 'G')
- Using len() function and in operator
  - if abase in bases:
    - print "it is a base."
- Indexing and slicing bases[1], bases[1:3]
- Concatenating lists bases + bases

Working with Biological Sequences

- Biological sequences are strings of different alphabets
  - seq1 = 'gcatgcagtattccattcctgctcagpytctgtggg'
  - seq2 = 'gcatgcagtattccattcctgctcagpytctgtggg'
- Some functions you will use often
  - len(seq1)
  - 'cat' in seq1
  - count(dna, 'a')
  - replace(dna, 'a', 'A')
- Indexing/Slicing a string
  - seq[1], seq[1:-1], seq[1:], seq[3:6]

Augmented Assignment Operators

- Common to assign a value to a variable based on its original value
- Augmented assignment operators provide condensed syntax

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Is Equivalent To</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=</td>
<td>x += 5</td>
<td>x = x + 5</td>
</tr>
<tr>
<td>/=</td>
<td>x /= 5</td>
<td>x = x / 5</td>
</tr>
<tr>
<td>*=</td>
<td>x *= 5</td>
<td>x = x * 5</td>
</tr>
<tr>
<td>-=</td>
<td>x -= 5</td>
<td>x = x - 5</td>
</tr>
</tbody>
</table>

Deleting an Element or a Slice

>>> bases = ['A', 'T', 'C', 'G']
>>> del bases[2]
>>> print bases

>>> del bases[1:2]
>>> print bases
List Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append(value)</td>
<td>Adds a value to the end of a list.</td>
</tr>
<tr>
<td>sort()</td>
<td>Sorts the elements in ascending order by default.</td>
</tr>
<tr>
<td>reverse()</td>
<td>Reverses the order of a list.</td>
</tr>
<tr>
<td>count(value)</td>
<td>Returns the number of occurrences of a value in a list.</td>
</tr>
<tr>
<td>index(value)</td>
<td>Returns the index of the first occurrence of a value in a list.</td>
</tr>
<tr>
<td>pop(i)</td>
<td>Removes the element at index i and returns the removed element.</td>
</tr>
<tr>
<td>remove(value)</td>
<td>Removes the first occurrence of a value from the list.</td>
</tr>
</tbody>
</table>

Using Nested Sequences

- **Nested Sequence**: A sequence inside another sequence.
- A list can contain lists or tuples.
- A tuple can contain tuples or lists.

```python
prots = [['170079667', 'threonine synthase'], ['170079668', 'hypothetical protein']]
#gi + description
prots[1][1]  # multiple indexing
```

Unpacking a Sequence

```python
>>> gi, des = ['170079667', 'threonine synthase']
>>> print gi
>>> print des

- **Sequence unpacking**: Automatically accessing each element of a sequence.
- The list is unpacked as result of assignment statement.

Variable References

- A variable refers to a place in memory where the value (or empty) is stored.
- **Variable assignment can be**:
  - initial (creates a new box in the computer’s memory the first time a variable name is seen)
  - shared (assign lists; default for mutable items)
  - copied (numbers, strings, tuples)

```python
mike
mr_dasen
"shocka" "dress shirt" "jacket"

# All variables refer to one single list
```

Avoid Shared References

```python
gene1 = ['170079667', '170079668']
gene2 = gene1
#now how many genes does gene1 has?
gene3 = gene1[:]
gene3.append('170079690')
#now how many genes does gene1 has?

- List slicing can create a new copy of a list and avoid shared references (but NOT for nested sequences)
  ```python
  a = [1, 2, [3, 4]]
  b = a[1]
b[1] = "22"
b[2][0] = "33"
  ```

Using copy.deepcopy()

- **Module**: copy
  - ref: http://docs.python.org/library/copy.html

```python
import copy
b = copy.copy(a)  # shallow copy, => b = a[:]
b = copy.deepcopy(a)  # deep copy of an object

# A deep (shallow) copy constructs a new compound object and then, recursively, inserts copies
# (references) into it of the objects found in the original.

Example: sokodu = copy.deepcopy(sokodu)
```
Accessing Elements of a Nested Sequence Using for Loop

```python
prots = [('170079667', 'threonine synthase'), ('170079668', 'hypothetical protein')]
for entry in prots:
    gi, des = entry
    print gi, des
```

Sequence unpacking: Automatically accessing each element of a sequence as a result of assignment statement

3. Control the flow

```python
baddna = 'tgagaatataugatattcnnn'
gooddna = ''
for base in baddna:
    if base in 'atcgATCG':
        gooddna += base
    elif base == 'u':
        gooddna += 't'
    elif base == 'U':
        gooddna += 'T'
    else:
        gooddna += '*'
print "baddna:  ", baddna
print "cleandna: ", gooddna
```

3.1 Branching structures
- Make choices based on conditions – to selectively execute certain portions of the code
  - Use if to execute code based on a condition
  - Use if-else to make a choice based on a condition
  - Use if-elif-else structures to make a choice based on a series of conditions
- Plan programs

3.2 while Statement

```python
from string import *
dna = 'tgagaattctatgaattc'
enz = 'gaattc'
print "dna", dna
site = find (dna, enz)
while site != -1:
    print "restriction site %s at position %d" % (enz, site)
    site = find (dna, enz, site + 1)
```

3.3 Using for Loops
- for loop
  - Like while loop, repeats a loop body
  - Unlike while loop, doesn’t repeat based on condition
  - Repeats loop body for each element in a sequence
  - Ends when it reaches end of the sequence
  - e.g., go through sequence of game titles and print each

3.4 Understanding conditions
- Condition: Expression that is True or False
  - True and False are values of type boolean
  - password == "secret" is condition - True or False
    - If variable password is equal to string "secret" condition evaluates to True
    - Otherwise, condition evaluates to False
  - The password program
- Often create conditions by comparing values
  (==, !=, >, <, >=, <=)
Treating Values as Conditions

- Any value can be interpreted as True or False when used as a condition
  - Any empty (None) or zero value is False
    - So, 0, "", and None are False
  - Any other value is True
    - So for example, -10, 2.5, "banana" are True
- if money:
  - money is treated as a condition
  - True when money not 0; False when money is 0

Using Compound Conditions

- Can create more complex conditions by joining simple conditions seen so far with logical operators to create a compound condition
- Logical operator: An operator (and, or, not) that joins conditions to form a large condition
  - dna = ""
    - while not dna:
      dna = raw_input("dna: ")
- Compound condition: A larger condition formed by joining simpler conditions
  - A or B and C
  - # be aware of the precedence of logical operators
    - # not > and > or

4. Working with Files!

- A typical sequence file in fasta format
  ```
  >seq1
  STATGPNEDKRVMLVIPGPNT
  ```

Opening and Closing a Text File

```python
# Opening a file
file = open("read_it.txt", "r")

# Reading file
line1 = file.readline()
line2 = file.readline()
line3 = file.readline()

# Closing a file
file.close()
```

- Must open before read (or write); then you read from and/or write to the file by referring to the file object
- Always close file when done reading or writing
- Can open a file for reading, writing, or both

File Access Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;r&quot;</td>
<td>Read from file. If the file doesn't exist, Python will complain with an error.</td>
</tr>
<tr>
<td>&quot;w&quot;</td>
<td>Write to a file. If the file exists, its contents are overwritten. If the file doesn't exist, it is created.</td>
</tr>
<tr>
<td>&quot;a&quot;</td>
<td>Appends to a file. If the file exists, new data is appended to it. If the file doesn't exist, it is created.</td>
</tr>
<tr>
<td>&quot;x&quot;</td>
<td>Creates a file. If the file doesn't exist, Python will complain with an error. Writes to and read from file. If the file exists, its contents are overwritten. If the file doesn't exist, it's created.</td>
</tr>
<tr>
<td>&quot;+&quot;</td>
<td>Appends and reads from a file. If the file exists, new data is appended to it. If the file doesn't exist, it's created.</td>
</tr>
</tbody>
</table>

Files can be opened for reading, writing, or both.

Reading a Line from a File

```python
# Reading a line
file = open("read_it.txt", "r")
line1 = file.readline()
line2 = file.readline()
line3 = file.readline()
```

- `readline()` file object method
  - Returns the entire line if no value passed
  - Once read all of the characters of a line (including the newline), next line becomes current line

```python
file.readline(number_of_characters) # a little confusing
```
Reading All Lines into a List

```
    text_file = open("read_it.txt", "r")
    lines = text_file.readlines()  # lines is a list!
```

- `readlines()` method reads a text file into a list
- Returns a list of strings
- Each line of the file becomes a string element in the list

Compared to: `read()`, which reads the entire file into a string (instead of a list of strings)

Looping Through a Text File

```
>>> text_file = open("read_it.txt", "r")
>>> for line in text_file:
    print line
Line 1
This is line 2
That makes this line 3
```

- Can iterate over open text file, one line at a time

Strip() & Split()

```
e.g., read_it.txt:

Hunter 98
Nathan 67

# The following lines for reading names and scores:

    text_file = open("read_it.txt", "r")
    for line in text_file:
        line = line.strip()  # line.split()
        (name, score) = line.split()  # sep, maxsplit)

str.split([sep[, maxsplit]]) -- Return a list of the words in the string, using sep as the delimiter string. If sep is not specified or None, any whitespace string is a separator. '1<>2<>3'.split('<>') returns ['1', '2', '3']

str.strip([chars]) -- Return a copy of the string with the leading and trailing characters removed. 'spacious '.strip() returns 'spacious'
```

Writing to Text Files

```
    text_file = open("write_it.txt", "w")
    text_file.write(">seq1\n")
    text_file.write("ACGTTGAACTATGGC\n")
```

- `write()` method writes new characters to file open for writing

```
    text_file = open("write_it.txt", "w")
    lines = [">seq1\n", "ACGTTGAACTATGGC\n"]
    text_file.writelines(lines)
```

- `writelines()` method writes list of strings to a file

5. Use and Write Functions

- Divide and conquer: divide complicated tasks into simpler and more manageable tasks.
- Avoid writing redundant program code (many programs require that a specific function is repeated many times)
- Enhance the readability of codes (a code can be broken up into manageable chunks; easy to follow the flow of the program)
- Testing and correcting errors is easy because errors are localized and corrected
- A single function written in a program can also be used in other programs also (software reuse)

User-defined Functions

```
def usage():
    """Display instructions."""
    print "Welcome to use ...!"
```

- Functions make programs easier to read, write and maintain
- **Function definition**: Code that defines what a new function does
- **Function header**: First line of a function definition
- Give function name that conveys what it does or produces
### Calling a Function

- **usage()**
  - Call tells the computer to execute function
  - Call works just like call to built-in function
  - Tells the computer to execute previously-defined function

### Encapsulation

- **Encapsulation**: A technique of keeping independent code separate by hiding the details
- **Variables created in a function cannot be directly accessed outside the function**
- **Parameters and return values allow for information exchange**
  - Functions with no arguments and no return values.
  - Functions with arguments and no return values.
  - Functions with no arguments and return values.
  - Functions with arguments and return values.

### Receiving and Returning Values

```python
import random
def RandDNA(totbp):
    alphabets = ['A', 'C', 'G', 'T']
    seq = ''
    for i in range(totbp):
        seq += alphabets[random.randrange(4)]
    return seq
```

- Receives one value and returns another
- **Parameter**: A variable name inside the parentheses of a function header that can receive a value
- **Argument**: A value passed to a parameter
- **Sample call**: ```seq = RandDNA(50)```  

### Positional & Keyword Arguments

```python
def AddProt(gi, des):
    print "You added a protein, gi=", gi, "des=", des
AddProt("170079667", "threonine synthase")
AddProt("hypothetical protein", "170079668")
```

- **Positional arguments**: Arguments passed to the parameters in order
- **Keyword argument**: Argument passed to a specific parameter using the parameter name
- **Keyword argument benefit**: The biggest benefit of using keyword arguments is clarity.

### Global versus Local Scopes

**Scopes**: Different areas of a program that are separate from each other
- Every function has its own scope
- Functions can't directly access each other's variables
- But can exchange information through parameters and return values

### Shadowing/Changing a Global Variable from Inside a Function

```python
def demo():
    global value1  # full access of global variable value1
    value1 = -value1
    value2 = -20  # a new variable with same name (shadow)
    print "Inside local scope":, value1, value2, value3
    value1 = 10
    value2 = 20
    value3 = 30
    print "In the global scope":, value1, value2, value3
demo()  # value1 is changed; value2 and value3 not
```

- **Shadow**: To hide a global variable inside a scope by creating a new local variable of the same name
- **Not a good idea**: Not a good idea to shadow a global variable
6. Object-Oriented Programming

- **OOP** allows representation of real-life objects as software objects (e.g., a dictionary as an object)
- **Object:** A single software unit that combines attributes and methods
- **Attribute:** A "characteristic" of an object; like a variable associated with a kind of object
- **Method:** A "behavior" of an object; like a function associated with a kind of object
- **Class:** Code that defines the attributes and methods of a kind of object (A class is a collection of variables and functions working with these variables)

```python
class Dna(object):
    def __init__(self, name, seq):
        self.name = name
        self.seq = seq
    def translate(self):
        ...

dna1 = Dna("gene1", "atcggtttgact")
dna2 = Dna("gene2", "ttcgatcgcttgct")
dna2.translate()
```

Creating Classes for Objects

- **Class:** Code that defines the attributes and methods of a kind of object
- **Instantiate:** To create an object (an Instance)
- `dna2.translate()` invokes translate() method of Dna object dna2

Using Constructors

- **Constructor:** A special method that is automatically invoked right after a new object is created
- Usually write one in each class
- Usually sets up the initial attribute values of new object in constructor

Printing an Object (How?)

```python
class Seq(object):
    ...
    def __str__(self):
        rep = "Seq object
        rep += "id: " + self.id + "\n"
        ...
seq1 = Seq()
print seq1
```

- __str__ is a special method that returns string representation of object

7. Handling Exceptions

```python
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in -toplevel-
  ZeroDivisionError: integer division or modulo by zero
```

- **Exception:** An error that occurs during the execution of a program
- **Exception is raised** and can be **caught** (or trapped) then handled
- Unhandled, halts program and error message displayed

try-except Statement

```python
try:
    file = open("seq.txt", "r")
except IOError:
    sys.exit("open file error! ")
```

- try statement sections off code that could raise exception
- Instead of raising exception, except block run
- If no exception raised, except block skipped
8. Using Modules

- Module imported using filename, just like built-in modules

```python
from Bio import SeqIO
handle = open("ls_orchid.gbk")
for seq_record in SeqIO.parse(handle, "genbank") :
    print seq_record.id
```

Only use Biopython after you are getting quite familiar with Python!!

9. Debugging Your Programs

- Logic errors are difficult to find
- Uses examples (from simple to complex ones) and check if your program gives correct results
- Using print statements to trace the progress (especially for those jobs that are going to take a while to finish)

```python
name = raw_input("type in your file name: ")
num = int(raw_input("enter your seed number: "))
```

You may use it for debugging purpose (to pause the program)

There is no easy way to learn programming

- Python basics
  - Variables
  - Basic data types: number, string, list, and dictionary
  - Control flow:
    - Branching structure: if, if-else, if-elif-else
    - Loops: for loop and while loop
  - Write function & class
  - I/O (read from files and write to files)
- Learning programming takes a lot of practice! And crappy codes are ok (in the beginning), as long as they work!

More about Python

- BioPython
  - You don’t use BioPython for the assignments
- Python 2 or 3?
  - Python 2 in most Linux servers
- Python as a glue language
  - Write the performance-critical parts of the application in C++ or Java, and use Python for higher-level control and customization
  - Call out from Python code to a fast, machine-code routine (e.g. compiled using C/C++ or Fortran)
    - writing an extension module
    - calling a shared-library subroutine using the ctypes module
    - A simple way: commands.getstatusoutput(whatever-command)
- The compiled python?