MetagenomeDB Primer

Outline

1. Introduction
2. Using MetagenomeDB
3. Test cases
4. Programming in Practice
   - Choosing a language: Python
   - Executing a Python program
5. First Elements of Programming
   - Variables: Representing your data
   - Statements: Processing your data
   - Flow control: Branching and Repetitions

What is MetagenomeDB?

METAGENOMEDB is a Python[1]-based library designed to easily store, retrieve and annotate metagenomic sequences. It provide an API to create and modify two types of objects, namely sequences and collections. Behind the scene, all data are handled to a MongoDB[2] database to ensure reliability and speed.

[1] `http://www.python.org/`
[2] `http://www.mongodb.org/`

Sequences and Collections

‘Sequences’ are any sequence a metagenomic project generate; mainly, reads and contigs. Sequences can be annotated and related to other sequences (to represent similarities, or when a sequence is part of another sequence; e.g., reads that are part of a contig).

‘Collections’ are sets of sequences. Collections can also contains sub-collections to represent hierarchies (e.g., replicate sets of reads, or alternative assemblies). Collections can be annotated and related to other sequences and collections.

Content of the MetagenomeDB toolkit

METAGENOMEDB has two components:

- a library, which can be used in your own Python scripts to access the database, and
- a set of command-line tools, which automatize the addition, annotation and deletion of sequences and collections in the database.

Among the command-line tools you will find utilities to import BLAST and FASTA alignment results, FASTA-formatted sequences, and ACE reads-to-contigs mappings.

Loading the MetagenomeDB library

Once installed, METAGENOMEDB can be used in any Python program as a module using the `import` statement:

```
import MetagenomeDB
```

Alternatively, you can provide an alias to shorten the name of the library:

```
import MetagenomeDB as mdb
```

Organization of this course

Layout

- 6 weeks (maybe more)
- 2-hours sessions

Goal: to show you how to turn (*implement*) a task into a working piece of software.

Mailing list

`http://groups.google.com/group/msu-programming-course`

What is a program?

A program is a text file (a *source code*) which contains a list of instructions to manipulate information (an *algorithm*).

These instructions are written in near-natural language a computer can understand (a *programming language*).

Programming is the activity of writing such source code to achieve a specific task, using the computer's resources—data acquisition, storage, computation, and output.

Why programming?

The advantages of programming are

- You can re-use complex pieces of software experts wrote to solve specific problems
  - Extracting data from obscure or complex file formats (*parsers*)
  - Storing huge amount of data without having to care about file management nor ease of retrieval (*databases*)
  - Performing complex calculations—statistical tests, equation solving...

Why programming?

- You can automatize tasks. If you can write a piece of code to perform a task once, you can perform it as many times as you want
  - Testing various parameters of an algorithm on the same data
  - Applying a given analysis procedure on many different chunks of data

The Python programming language

For this course the programming language we will use is Python (`http://www.python.org/`).

The interests of this language are

- The syntax is very close to written english (much more than Perl or Java, for example)—easy to learn
- This is an *interpreted* language, meaning the code you write can be 'directly' executed by the computer without intermediary *compilation* step—easy to run
- There is a *shell* available, that allows you to quickly test your code or find documentation on other's code

The Python programming language

Example

Printing a simple sentence on the screen:

```
print "Hello, world!"
```

Comparing two values:

```
score_msu = get_score("Montana State University")
score_um = get_score("University of Montana")

if (score_msu <= score_um):
    print "ERROR: This shouldn't happen."
else:
    print "Everything is fine."
```

The Python interpreter

'Running' or 'executing' a program written in Python requires a Python *interpreter*.

An interpreter is a program (written in another language) that reads your instructions and convert them into orders for the computer's CPU.

The Python interpreter can work in two modes: normal, and *interactive*.

The Python interpreter

Normal mode

In the normal mode, the Python interpreter reads a file containing your instructions (a *script*) and executes them.

The interpreter is invoked on the command line. Any output from your program is displayed there.

The Python interpreter

Normal mode

For example, executing the following file:

my_first_python_code.py

```
print "Hello, world!"
```

... will be done this way:

Terminal

```
shell% python ./my_first_python_code.py
Hello, world!
```

The Python interpreter

Normal mode

In all Unix systems you can turn a Python script into an executable in two steps.

1. A line must be added as the very beginning of your file:

my_first_python_code.py

```
#!/bin/env python
print "Hello, world!"
```

The Python interpreter

Normal mode

2. The file must be set as executable:

Terminal

```
shell% chmod +x my_first_python_code.py
```

Then, you can execute your script by typing

Terminal

```
shell% ./my_first_python_code.py
```

The Python interpreter

Interactive mode

In the interactive mode, the Python interpreter reads your instructions as you enter them on the keyboard, and executes them each time you press Enter. It is similar to a Unix shell:

Terminal

```
shell% python
Python 2.6.4 (r264:75821M, Oct 27 2009)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
>>> print "Hello, world!"
Hello, world!
>>> _
```

Variables
What is a variable?

In order to manipulate data in a program we first need to *store* it in the computer memory, so it can be retrieved later. The portion of memory devoted to this data is called a *variable*.

An identifier, or variable *name*, is used to request the content of this variable or alter its value.

Variables
What is a variable?

A value is assigned to a variable using the *assignment* statement, $=$.

```
>>> a = 1
>>> print a
1
```

Note

Variable name are case sensitive, so *a* and *A* are two distinct variables.

Variables
Types of variable

Python proposes different types of variables, that are sufficient to represent any kind of information:

- *numbers*: `1`, `-0.3`, `1e-5`
- *strings*: `"a piece of text"`, `"another piece"`
  → *Strings must be quoted to distinguish them from variable names*
- *booleans*: `True`, `False`

Variables
Types of variable

Python also proposes powerful *compounds* variable types, used to group together other values.

- *lists*, also called *vectors* or *arrays*:
  `[1, "a piece of text", True]`
  → *Symbols '`[`' and '`]`' are used to represent the list*

- *dictionaries*, also called *maps* or *associative arrays*:
  `{"hello": "bonjour", "bye": "au revoir"}`
  → *Symbols '`{`', '`}`' and '`:`' are used to represent the dictionary*

Variables
Lists

Lists are a succession of values in a specific order, delimited by square brackets. `[1, 2, 3]` is not the same as `[3, 2, 1]`.

List items need not all have the same type. Lists can also contain other lists.

```
>>> a = [1, 2, 3]
>>> b = ["foo", a, "bar"]
>>> print b
['foo', [1, 2, 3], 'bar']
```

Variables
Lists

We can access any item in a list by its index, starting at 0:

```
>>> b = ["foo", [1, 3, 4], "bar"]
>>> print b[0]
'foo'
>>> print b[1]
[1, 3, 4]

>>> print b[1][2]
4
```

→ *Access to nested lists*

Variables
Lists

Note

The Python interpreter will report an error if you try to access an item outside of the list's range:

```
>>> print b[4] → This shouldn't work!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

This process of error reporting is called *exception*. Various exceptions (here, `IndexError`) are 'thrown' by the interpreter depending of the error.

Variables
Lists

Using indices we can also modify any item in a list:

```
>>> b = ["foo", [1, 3, 4], "bar"]
>>> print b
['foo', [1, 3, 4], 'bar']
>>> b[1] = True
>>> print b
['foo', True, 'bar']
```

Variables
Lists

Python allows sublist access using the indices of the first and last items (minus one). This is a feature called *slicing*.

```
>>> c = ["a", "b", "c", "d", "e"]
>>> print c[0]  → First item
'a'
>>> print c[0:3]  → Items 0 to 2 (3-1)
['a', 'b', 'c']
>>> print c[3:]  → Items from the third one
['d', 'e']
>>> print c[:4]  → Items up to the fourth
['a', 'b', 'c', 'd']
```

Variables
Lists

List indices can also be negatives. In this case they point elements at index *n* - index, with *n* the number of elements in the list.

```
>>> c = ["a", "b", "c", "d", "e"]
>>> print c[-1]
'e'
>>> print c[-3:]
['c', 'd', 'e']
>>> print c[:-2]
['a', 'b', 'c']
```

→ *Item at position 5-1 = 4*

→ *3 last items*

→ *All items except the two last ones*

Variables
Lists

Lists can be joined with the + operator:

```
>>> d = [1, 2, 3] + [True, False]
>>> print d
[1, 2, 3, True, False]
```

Strings can be manipulated as lists (lists of characters):

```
>>> e = "Montana State University"
>>> print e[:7]
'Montana'
>>> print e[8:13]
'State'
```

Variables
Dictionaries

Dictionaries are special lists, where each element is accessed to using a *key* rather than an index:

```
f = { key1: value1, key2: value2 ... }
```

Note the use of braces (‘{’ and ‘}’), colons (to separate a key from its value) and commas (to separate multiple key/value pairs, if more than one).

Variables
Dictionaries

Example:

```
>>> f = {
...         "aardvark": "a nocturnal mammal ...",
...         "zyzzyva": "any of various ..."
...       }
```

Note

Keys must be unique, so that each value is referenced by one (and only one) key. Keys can be of any immutable type (numbers, booleans, strings, tuples).

Variables
Dictionaries

Accessing a value in a dictionary requires to know its key:

```
>>> print f["aardvark"]
'a nocturnal mammal ...'

>>> print f["!?"] → This shouldn't work!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: '!?'
```

Dictionaries

As for regular lists, any value of a dictionary can be modified:

```
>>> f["python"] = "a large heavy-bodied ..."
>>> print f["python"]
'a large heavy-bodied ...'
```

Variables
Altogether

Using combinations of strings, numbers, booleans, lists and dictionaries it is possible to describe and manipulate virtually any type of information in Python:

```
>>> saccharomyces_cerevisiae = {
...     "Number of genes": 6400,
...     "Kingdom": "Fungi",
...     "Genes": {
...         "15S_RRNA": { "description": "..." },
...         "ZWF1": { "locus": "YNL241C" }
...      }
...   }
```

Statements

A program is written as a succession of *statements* or *commands*.

A statement is a basic instruction that is given to the Python interpreter for execution.

A statement is a combination of *keywords*, *values* or *variables*, and possibly some symbols.

Statements

Only 29 keywords are used by Python: `and`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `exec`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `not`, `or`, `pass`, `print`, `raise`, `return`, `try`, `while`, and `yield`.

Among the symbols used: `:`, `=`, `[`, `]`, `{`, `}`, and `#`.

Statements
Types of statements

It exists at least 8 types of statements:

- **assignments** (=), to assign a value to a *variable*
- **arithmetic operators** (`+`, `-`, `*`, `/`, `**`, `%`), to combine numbers
- **boolean operators** (`and`, `or`, `not`, ∧, `in`, `==`, `!=` **...**), to compare values

Statements
Types of statements

- **calls** (*print*, *raise*, *exec* ...), to execute a set of instructions (called *function*, *method* or *sub-program*)
- **returns** (*return*, *yield*), to return a value from within a function to the main program
- **branching** (*if*, *elif*, *else*, *try*, *except*, *finally*), to jump to other parts of the program based on the value of a variable
- **iterations** (*for*, *while*), to execute a group of instructions several times
- **special flow controls** (*pass*, *continue*, *break*), to arbitrarily stop the flow of instructions or skip iterations

Statements
Arithmetic operators

Python supports all the four basic operators: addition, subtraction, multiplication and division.

```
>>> 1 + 1
2
>>> 6 / -2
-3
>>> 3 * 5.2
15.600000000000001
>>> "bob" + "cat"
'bobcat'
```

Statements
Arithmetic operators

Note

The division of two numbers returns by default the floor. Hence,

```
>>> 5 / 2
2
```

It is possible to return the real value however, by telling Python that either the numerator or denominator is a real number:

```
>>> 5.0 / 2
2.5
>>> 5 / 2.0
2.5
```

Statements
Arithmetic operators

Additional operators are modulo (`%`) and power (`**`):

```
>>> 26 * 4 + 3
107
>>> 107 / 4
26
>>> 107 % 4
3
>>> 2 ** 4
16
>>> 10 ** 3
1000
```

Statements
Arithmetic operators

Parentheses can be used to specify the order of operations (and also, to improve readability):

```
>>> 3 + 4 * 2
11
>>> (3 + 4) * 2
14
```

Python evaluates arithmetic operators in the following order: power, then multiplication, division, and modulo, then addition and subtraction, then boolean operators.

Statements
Boolean operators

Python can perform logical operations on booleans (i.e., values that are either true or false).

AND

```
>>> True and True
True
>>> True and False
False
>>> False and False
False
>>> True and False
False
```

Other syntax: `&`

Statements
Boolean operators

OR

```
>>> True or True
True
>>> True or False
True
>>> False or False
False
>>> False or True
True
```

Other syntax: |

Statements
Boolean operators

XOR

```
>>> True ^ True
False
>>> True ^ False
True
>>> False ^ False
False
>>> False ^ True
True
```

Statements
Boolean operators

NOT

```
>>> not True
False
>>> not False
True
```

Note: Python evaluates boolean operators in the following order: negation, logical and, logical or.

Statements
Equalities

Value comparisons (equality '==', inequality '!=', less than '<', greater than '>', less than or equal to '<=', greater than or equal to '>=') can be performed, and combined with boolean operators:

```
>>> 1 >= 2
False
>>> (10 > 0) and (10 < 100)
True
>>> (10 + 1) == 11
True
>>> "bob" != "cat"
True
```

Statements
Subgroups

The `in` operator tests if a value is part of a list or a dictionary:

```
>>> a = [1, 2, 3]
>>> b = { "blue": "bleu", "red": "rouge" }
>>> 1 in a
True
>>> "1" in a
False
>>> "blue" in b
True
>>> blue in b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'blue' is not defined
```

Statements
Subgroups

Remember that strings are also considered as lists:

```
>>> "Montana" in "Montana State University"
True
```

Statements
Calls

Python provides numerous pieces of code, called *functions*, to perform basic operations.

Any function has three properties: it is given a *name*, it may accept *inputs* or *arguments* (data to process), and it may produce *outputs* (results of the processing).

Note

The term ‘function’ stems from mathematics. A mathematical function is a relation between inputs and outputs (e.g., graph equation).

Statements
Calls

A function is called by typing its name, followed by any input it accepts (if any) between parentheses. It may or may not return a value.

```
>>> a = range(5)
>>> print a
[0, 1, 2, 3, 4]
```

The function `range` is provided by Python to generate a list of n numbers starting (by default) from 0. Here, we called this functions with $n = 5$, and stored its output in the variable a.

Statements
Calls

Note

The `print` statement is actually a function, which purpose is to display the value of its inputs on the screen.

For practical purpose, the Python authors decided not to ask programmers to use parentheses to delimitate the input(s) of this function. Thus, `print 1` should really be `print(1)`.

Flow control

A program is a list of instructions that the Python interpreter will read sequentially.

However, it is possible to alter the flow of execution of these instructions in three ways:

- By calling a function (the program 'jumps' to the set of instructions contained in this function, then goes back to the point this function was called from)
- By *branching*; i.e., by choosing to execute a set of instructions over others based on the result of a statement
- By *repeating*; i.e., by choosing to execute a set of instructions a given number of times

Flow control

Branching: if, else

The `if` and `else` keywords allow for the conditional execution of sets of instructions based on the evaluation of a statement:

`if` *`statement`*`:`
→ *Set of instructions executed if the statement returns True*
`else:`
→ *Set of instructions executed if the statement returns False*

Flow control
Branching: elif

The `elif` keyword allows to test additional statements in case the previous one(s) return False:

`if` *`statement #1`*`:`
→ *Set of instructions executed if statement #1 returns True*

`elif` *`statement #2`*`:`
→ *Set of instructions executed if statement #1 returns False and statement #2 returns True*

`else:`
→ *Set of instructions executed if statement #1 and #2 return False*

Flow control
Branching

Two syntax subtleties that you have to remember.

1. There is a colon at the end of both `if`, `else` and `elif` statements. If you forget it, the Python interpreter will complain.
2. The sets of instructions that are executed have to be *indented* using *tabulations*. This is a Python specificity.

Flow control
Branching: Example

```
score_msu = get_score("Montana State University")
score_um = get_score("University of Montana")

if (score_msu < score_um):
    print "How did this happen!?"
    go_back_in_shame()
elif (score_msu == score_um):
    print "They were lucky."
else:
    print "Yeah, that was obvious."
    taunt()
```

Flow control
Repetition

The `for`, `in` and `while` keywords allow for the repeat execution of a set of instructions based either on the content of a variable (`for` and `in`) or the result of a statement (`while`):

`for` `variable` **`in`** *`iterator`*`:`
→ *Set of instructions to execute for any value found in the variable* `iterator`*, or returned by a function*

`while` *`statement`*`:`
→ *Set of instructions to execute as long as the statement returns True*

Flow control

Repetition: for ... in ...

An iterator is either a *variable* that contains a list of items, or a *function* that returns a list of items. The variable name used between the `for` and `in` keywords will receive each of these items, one at a time:

```
>>> for i in ["a", "b", "c"]:
...     print i → Do not forget the tabulation!
...
a
b
c
>>>
```

Flow control

Repetition: for ... in ...

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

Flow control
Repetition: while

The `while` keyword is useful when a set of instructions have to be executed until some conditions are satisfied:

```
a = 0
while (a < 5):
    print a
    a = a + 1
```

Flow control
Special keywords

Two keywords can be used to control a flow from *within* a `for` or `while` loop: `continue`, and `break`.

The `continue` keyword asks Python to skip the remaining instructions, and to execute the next iteration.

The `break` keyword asks Python to stop the iteration.

Flow control

Special keywords: continue

```
for i in range(5):
  if (i == 3):
    continue
  print i
```

... will result in showing 0, 1, 2, and 4. Because when *i* equals 3, the program is told to skip all other instructions in the loop, including `print i`.

Flow control

Special keywords: break

```
a = 0
while (True): → Will loop forever!
  if (a == 10):
    break
  print a
  a = a + 1
```

... will print all numbers from 0 to 9, then stop.

End of session 1.