First Steps Towards Programming

Python for Genomic Data Science

Interactive Mode Programming

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)]

>>> print(“Hello world!”)
Hello world!
>>>
```

NUMBERS AND STRINGS

Using Python As A Calculator

```
>>> 5+5
10
>>> 10.5-2*3
4.5
>>> 10**2
100
>>> 17.0 // 3
5.0
>>> 17 % 3
2
>>> 5 * 3 + 2
17
```

** is used to calculate powers

floor division discards the fractional part

the % operator returns the remainder after division

* takes precedence over +, -

Numbers can be different "types"

1. **Integer numbers:**

```
>>> type(5)
<type 'int'>
```

2. **Real numbers:**

```
>>> type(3.5)
<type 'float'>
>>> 12/5
2
>>> float(12)/5
2.4
>>> 12.0/5
2.4
```

What happened?

3. **Complex numbers:**

```
>>> type(3+2j)
<type 'complex'>
>>> (2+1j)**2
(3+4j)
```

Strings

- **Single quoted strings:**

```
>>> ‘atg’
‘atg’
```

- **Double quoted strings:**

```
>>> “atg”
‘atg’
```

```
>>> ‘This is a codon, isn't it?'
File "<stdin>", line 1
    'This is a codon, isn't it?'
                          ^
SyntaxError: invalid syntax
```

Why? Use double quotes instead:

```
>>> “This is a codon, isn't it?”
"This is a codon, isn't it?”
```

Or use \ to escape quotes:

```
>>> ‘This is a codon, isn\'t
it?’
"This is a codon, isn't it?”
```

Strings (cont'd)

- String can span multiple lines with triple quotes:

```
>>> """                  <-- use triple-quotes: """..."""  or '''...'''
... >dna1
... atgacgtacgtacgtacgtagctagctataattagc
... atgatdgdgtata
... >dna2
... ttggtgtcgccgcccccgcgttttaatatgcgctat
... """
'\n>dna1\natgacgtacgtacgtacgtagctagctataattagc
\natgatdgdgtata
\n>dna2\nttggtgtcgccgcccccgcgttttaatatgcgctat\n'
```

\n is a special character that signifies a new line

Escape Characters

Backslash is an "escape" character that gives the next character a special meaning:

Construct	Meaning
`\n`	Newline
`\t`	Tab
`\\`	Backslash
`\"`	Double quote

Printing Strings Nicely

- The `print()` function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>> print("""\
... >dna1
... atgacgtacgtacgtacgtagctagctataattagc
... atgatdgdgtata
... >dna2
... ttggtgtcgccgcccccgcgttttaatatgcgctat
... """)
>dna1
atgacgtacgtacgtacgtagctagctataattagc
atgatdgdgtata
>dna2
ttggtgtcgccgcccccgcgttttaatatgcgctat
```

End of lines are automatically included in the string, but it's possible to prevent this by adding a \ at the end of the line.

Basic String Operators

`+`	concatenate strings
`*`	copy string (replicate)
`in`	membership: true if first string exists inside
`not in`	non-membership: true if first string does not exist in second string

```
>>> 'atg' + 'gtacgtccgt'
'atggtacgtccgt'
>>> 'atg'*3
'atgatgatg'
>>> 'atg' in 'atggccggcgta'
True
>>> 'n' in 'atgtgggg'
False
```

VARIABLES

Variables

- *Variables* are storage containers for numbers, strings, etc.
- The equal sign (=) is used to assign a value to a variable:

```
>>> codon = 'atg'
```

Note: no result is displayed before the next interactive prompt

```
>>> dna_sequence="gtcgcctaaccgtatattttcccgt"
```

- If a variable is not "defined" (assigned a value), trying to use it will give you an error:

```
>>> dna
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'dna' is not defined
```

Variables (cont'd)

The name we associate to a value is called a "*variable*" because its value can change:

```
>>> a=4
>>> a
4
>>> b=a
>>> b
4
>>> b=b+3    #yes, this is legal
>>> b
7
>>> a
4
```

Variables – Tips And Notes

Tips:

• Give *meaningful* names to variables: e.g. `dnaSequence` is better than `s`

Notes: variable names follow specific rules:

• They are *case-sensitive*. This means that the following variables are different (i.e. they refer to different values):

```
>>> myvariable = 1
>>> MyVariable = 2
>>> MYVARIABLE = 3
```

• Variable names can ONLY consist of letters, numbers (not the first letter), and the underscore character.

Valid names: `name, _str, DNA , sequence1`

Invalid names: `1string, name#, year@20`

More String Operators

[x] *indexing*: gives the character at position **x** in a string
[x:y] *slicing*: gives a substring between positions **x** and **y** in a string

```
>>> dna="gatcccccgatattatttgc"
>>> dna[0]
'g'
>>> dna[-1]
'c'
>>> dna[-2]
'g'
>>> dna[0:3]
'gat'
>>> dna[:3]
'gat'
>>> dna[2:]
'tcccccgatattatttgc'
```

the position in the string is called its *index* and the first character has index 0

indices may also be negative numbers, which start counting from the right, beginning at -1

the start is always included, and the end always excluded

an omitted first index defaults to zero

an omitted second index defaults to the size of the whole string being sliced

A Useful String Function

The built-in function `len()` returns the length of a string:

```
>>> dna="acgctcgcgcggcgatagctgatcgatcggcgcgctttttttttaaaag"
>>> len(dna)
49
```

Note. Python has a number of functions and types built into it that are always available. Some other examples:

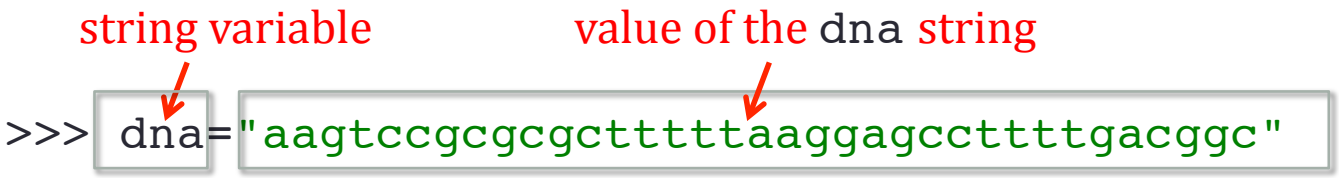
`type()`

`print()`

You can find more information about any built-in function by using the pydoc command in a shell or the help function of the interpreter. Try:

```
>>> help(len)
```

Strings as Objects



String variables are things that know more than their *values*. They are *objects*. They can also perform specific actions (functions), called *methods* that only they can perform:

```
>>> dna.count('c')
9
```

the . (dot) operator can be interpreted as: "ask object `dna` to do something" such as: count how many `'c'` characters it contains.

The method `count()` can also count substrings:

```
>>> dna.count('gc')
5
```

More Useful String Functions

```
>>> dna
'acgctcgcgcggcgatagctgatcgatcggcgcgctttttttttaaaag'
>>> dna.upper()
'ACGCTCGCGCGGCGATAGCTGATCGATCGGCGCGCTTTTTTTTTAAAAG'
>>> dna
'acgctcgcgcggcgatagctgatcgatcggcgcgctttttttttaaaag'
>>> dna.find('ag')
16
>>> dna.find('ag',17)
47
>>> dna.rfind('ag')
47
```

there is also a `lower()` function

Note that the value of the string `dna` did not change.

only the first occurrence of `'ag'` is reported

we need to tell the method where to start looking

same as `find()`, but search backwards in string

You can find all methods for the type `str`:

```
>>> help(str)
```

More Useful String Functions (cont'd)

```
>>> dna
'acgctcgcgcggcgatagctgatcgatcggcgcgctttttttttaaaag'
>>> dna.islower()
True
>>> dna.isupper()
False
```

Another useful string method for bioinformatics programming is:

```
>>> dna.replace('a','A')
'AcgctcgcgcggcgAtAgctgAtcgAtcggcgcgctttttttttAAAAg'
```

A First Program In Python

```
read DNA sequence from user
>>> dna = 'acgctcgcgcggcgatagctgatcgatcggcgcgctttttttttaaaag'
count the number of C's in DNA sequence
>>> no_c=dna.count('c')
count the number of G's in DNA sequence
>>> no_g=dna.count('g')
determine the length of the DNA sequence
>>> dna_length=len(dna)
compute the GC%
>>> gc_percent=(no_c+no_g)*100.0/dna_length
print GC%
>>> print(gc_percent)
53.06122448979592
```

The .0 after 100 is only required in Python 2.x

Executing Code From A File

Write the code of your program in a file and then pass the file to the Python interpreter:

gc.py ← It is a convention for files containing python code to have a **py** extension

```
dna = 'acgctcgcgcggcgatagctgatcgatcggcgcgctttttttttaaaag'
no_c = dna.count('c')
no_g = dna.count('g')
dna_length = len(dna)
gc_percent = (no_c + no_g) * 100.0 / dna_length
print(gc_percent)
```

Now pass this file to the interpreter:

```
~/Work/courses/python>python gc.py
53.06122448979592
```

Executing Code From a File (cont'd)

gc.py

```
#!/usr/bin/python
```

← You can make your file executable if you put this line at the beginning of your file

```
dna = 'acgctcgcgcggcgatagctgatcgatcggcgcgctttttttttaaaag'
no_c=dna.count('c')
no_g=dna.count('g')
dna_length=len(dna)
gc_percent=(no_c+no_g)*100/dna_length
print(gc_percent)
```

Now you can run your python program like this:

```
~/Work/courses/python>./gc.py
53.06122448979592
```

Executing Code From a File (cont'd)

- Make your python file executable by setting the x execution bit.

```
chmod a+x gc.py
```

- You can find where the Python interpreter is located by running `which`:

```
~/Work/courses/python>which python
/usr/bin/python
```

Now you can run your python program like this:

```
~/Work/courses/python>./gc.py
53.06122448979592
```

Adding Comments

gc.py

```
#! /usr/bin/python
"""                    ← multiple line comments are included in between """..."""
This is my first Python program.
It computes the GC content of a DNA sequence.
"""
# get DNA sequence:    ← everything that follows a # is ignored up to the line end
dna = 'acgctcgcgcggcgatagctgatcgatcggcgcgctttttttttaaaag'
no_c=dna.count('c') # count C's in DNA sequence
no_g=dna.count('g') # count G's in DNA sequence
dna_len=len(dna) # get the length of the DNA sequence
gc_perc=(no_c+no_g)*100.0/dna_len # compute gc percentage
print(gc_perc) # print GC% to screen
```

Reading Input

Python 2.x Python 3.x

```
>>> dna=     input("Enter a DNA sequence, please:")
Enter a DNA sequence, please:agtagcatgaggagggacttc
>>> dna
'agtagcatgaggagggacttc'
```

`input()` (Python 3) and `raw_input()` (Python 2) *always* return strings:

```
>>> my_number=raw_input("Please enter a number:")
Please enter a number:10
>>> type(my_number)
<type 'str'>
>>> actual_number=int(my_number)
>>> type(actual_number)
<type 'int'>
```

use the built-in function `int()` to transform the string into an integer

Some Conversion Functions

Function	Description
`int(x [,base])`	converts x to an integer
`float(x)`	converts x to a floating-point (real) number
`complex(real [,imag])`	creates a complex number
`str(x)`	converts x to a string
`chr(x)`	converts an integer to a character

```
>>> chr(65)
'A'
>>> str(65)
'65'
```

← every number <256 has a unique character associated to it

Fancier Output Formatting

leave out the parentheses in Python 2.x

```
>>> print("The DNA sequence's GC content is",gc_perc,"%")
The DNA sequence's GC content is 53.06122448979592 %
```

The value of the `gc_perc` variable has many digits following the dot which are not very significant. You can eliminate the display of too many digits by imposing a certain format to the printed string:

note the double % to print a % symbol

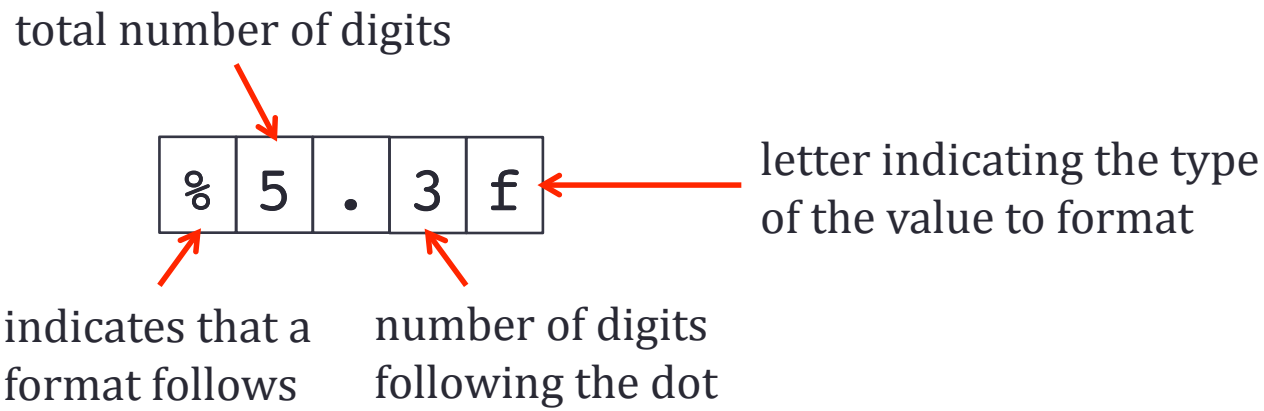
```
>>> print("The DNA sequence's GC content is %5.3f %%" % gc_perc)
The DNA sequence's GC content is 53.061 %
```

formatting string

value that is formatted

percent operator separating the formatting string and the value to replace the format placeholder

The Formatting String



Other formatting characters:

```
>>> print("%d" % 10.6) # 10.6 is first transformed into an integer
10
>>> print("%3d" % 10.6) # notice the space in front of 10
 10
>>> print("%o"% 10) # use %o for an octal/%x for a hexadecimal integer
12
>>> print("%e" % 10.6) # 'E' notation uses powers of 10
1.060000e+01
>>> print("%s" % dna)
gtataggagatatattag
```