Python for EVOP 2018

Sofia Robb Ph.D.
Genomic Scientist @ Stowers Institute for Medical Research
https://planosphere.stowers.org, PubMed
https://cuttingclass.stowers.org, The American Biology Teacher
https://simrbase.stowers.org, PubMed
https://smedgd.stowers.org, PubMed

Why use biopython

Why Python?

It really isn't a question of why are you learning Python, but why are you learning to script. Learning one programming language make it easier for you to learn another. The basic concepts, like conditional statements and loops are the same in most languages. These are the concepts that *some* students struggle with at first. Popular thought is that Python is the best language for students to learn programming. I think you should learn what ever language you can get someone to help you learn. My personal favorite is Perl. I have been using it for the longest and can write a script very quickly. Python might be better a better choice if you are writing code with others on a large project. The modularity, classes, and object orientness of Python make this easier.

Why Script?

As biologist we often need to run customized analyses or to reformat files. It is nice if we can do this when we want, rather than wait for someone else to do it for us. We also often have to repeat tasks over and over. Why do something hundreds of times when we can write one script to do these tasks for us. Scripting allows us to be independent, efficient, and have reproducible results.

Teaching Format

We only have 2 days. I am going to teach you the basics and more importantly teach you how to teach yourself. I am going to point you to online references to find information about data types, methods, functions. We will also have short blocks of lecturing followed by simple hands on exercises.

Installing Python

1. Install Anaconda
2. Install Python

Python References

Books

1. Python for Biologist
2. Learn Python in One Day and Learn it Well

Websites

1. TutorialsPoint
2. Anaconda
3. BioPython

Python Overview

Python is a scripting language. It is useful for writing medium-sized scientific coding projects. When you run a Python script, the Python program will generate byte code and interpret the byte code. This happens automatically and you don't have to worry about it. Compiled languages like C, C++ will run much faster, but are much much more complicated to program. Languages like java (which also gets compiled into byte code) are well suited to very large collaborative programming projects, but don't run as fast as C and are more complex that Python.

Python has

- data types
- functions
- objects
- classes
- methods

Data types are just different types of data which are discussed in more detail later. Examples of data types are integer numbers and strings of letters and numbers (text). These can be stored in variables.

Functions do something with data, such as a calculation. Some functions are already built into Python. You can create your own functions as well.

Objects are a way of grouping a set of data and functions (methods) that act on that data

Classes are a way to encapsulate (organize) variables and functions. Objects get their variables and methods from the class they belong to.

Methods are just functions that belong to a class. Objects that belong to a class can use methods from that class.

Running Python

There are two versions of Python: Python 2 and Python 3. We will be using 3. This version fixes some of the problems with Python 2 and breaks some other things. A lot of code has already been written for Python 2 (it's older), but going forwards, more and more new code development will use Python 3.

Interactive Interpretor

Python can be run one line at a time in an interactive interpretor. You can think of this as a Python shell. To launch the interpreter type the following into your terminal window:

`% python3`

or

`% ipython`

Note:

- '%' indicates the command line prompt
- '>>>' indicates the interpreter

First Python Commands:

```
1  >>> print("Hello, EVOP2018!")
2  Hello, EVOP2018!
```

> *Note:* `print` *is a function. Function names are followed by (), so formally, the function is* `print()`

1. Open the interactive interpreter. Type `python3` in the terminal window (`ipython` is another interactive terminal).
2. Use the `print()` function to print something to the screen. Make sure to use parenthesis `()`, quotes `""`, and a semi-colon `;` like

in the example.

Python Scripts

- The same code from above is typed into a text file using a text editor.
- Python scripts are always saved in files whose names have the extension '.py' (i.e. the filename ends with '.py').

File Contents:

```
print ("Hello, EVOP2018!")
```

Running Python Scripts

Typing the Python command followed by the name of a script makes Python execute the script. Recall that we just saw you can run an interactive interpreter by just typing `python` on the command line

Execute the Python script like this (% represents the prompt)

```
% python3 test.py
```

This produces the following result:

```
Hello, EVOP2018!
```

A quicker/better way to run python scripts

If you make your script executable, you can run it without typing `python3` first. Use `chmod` to change the permissions on the script like this

`chmod +x test.py`

You can look at the permissions with

```
% ls -l test.py
-rwxr-xr-x  1 srobb  staff  60 Oct 16 14:29 test.py

```

The first field of -, r, w and x characters define the permissions of the file. The three 'x' characters means anyone can execute or run this script.

We also need to add a line at the beginning of the script that tells the shell to run python3 to interpret the script. This line starts with #, so it looks like a comment to python. The '!' is important as is the space between `env` and `python3`. The program `/usr/bin/env` looks for where `python3` is installed and runs the script with `python3`. The details may seem a bit complex, but you can just copy and paste this 'magic' line.

The file test.py now looks like this

```
1  #!/usr/bin/env python3
2  print ("Hello, EVOP2018!")
```

Now you can simply type the name of the script to run it. Like this

```
1  % ./test.py
2  Hello, EVOP2018!
3
```

1. Open your text editor and add `#!/usr/bin/env python3` to the top of your file. There cannot be any white space above or before this line.
2. Use the `print()` function to print something to the screen.
3. Save your script.
4. Make it executable with `chmod +x`. (You only have to do this one time per script.)
5. Run the script on the command line `./yourScript.pl`

Syntax

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters or other special characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, `seq_id` and `seq_ID` are two different identifiers in Python.

Naming conventions for Python Identifiers

- The first character is lowercase, unless it is a name of a class. Classes should begin with an uppercase characters. (ex. Seq)
- Private identifiers begin with an underscore. (ex. `_private`)
- Strong private identifiers begin with two underscores. (ex. `__private`)
- Language-defined special names begin and end with two underscores. (ex. `__special__`)

Reserved Words

The following is a list of Python keywords. These are special words that already have a purpose in python and therefore cannot be used as identifier names.

```
and       exec      not
as        finally   or
assert    for       pass
break     from      print
class     global    raise
continue  if        return
def       import    try
del       in        while
elif      is        with
else      lambda    yield
except

```

Lines and Indentation

Python denotes blocks of code by line **indentation**. Incorrect line spacing and/or indention will cause an error or could make your code run in a way you don't expect. You can get help with indentation from good text editors or Interactive Development Environments (IDEs).

The number of spaces in the indentation need to be consistent but a specific number is not required. All lines of code, or statements, within a single block must be indented in the same way. For example:

```
#!/usr/bin/env python3
for x in (1,2,3,4,5):
    if x > 4:
        print("Hello")
    else:
        print(x)
print('All Done!')
```

Comments

Comments are an essential programming practice. Making a note of what a line or block of code is doing will help the writer and readers of the code. This includes you!

Comments start with a pound or hash symbol `#`. All characters after this symbol, up to the end of the line are part of the comment and are ignored by Python.

The first line of a script starting with `#!` is a special example of a comment that also has the special function in unix of telling the unix shell how to run the script.

```
#!/usr/bin/env python3

# this is my first script
print ("Hello, EVOP2018!") # this line prints output to the screen
```

Blank Lines

Blank lines are also important for increasing the readability of the code. You should separate pieces of code that go together with a blank line to make 'paragraphs' of code. Blank lines are ignored by the Python interpretor.

Data Types and Variables

This is our first look at variables and data types. Each data type will be discussed in more detail in subsequent sections.

The first concept to consider is that Python data types are either immutable (unchangeable) or not. Literal numbers, strings and tuples cannot be changed. Lists, dictionaries and sets can be changed. So can individual (scalar) variables. You can store data in memory by putting it in different kinds variables. You use the `=` sign to assign a value to a variable.

Numbers and Strings

Numbers and strings are two common data types. Literal numbers and strings like this `5` or `'my name is'` are immutable. However, their values can be stored in variables and then changed.

For Example:

```
gene_count = 5
gene_count = 10
```

> *You should give your variables names that help you understand what they store. gene_count, expression, sequences are all good identifiers or variable names. k, x, data, var1, var2 are bad because you can't tell what they store. This means it's harder to understand the script and to spot errors or bugs in your script.*

Different types of data can be assigned to variables, i.e., integers (1,2,3), floats (floating point numbers, 3.1415), and strings (text).

For Example:

```
count   = 10    # this is an integer
average = 2.5   # this is a float
message = "Welcome to Python" # this is a string
```

10, 2.5, and "Welcome to Python" are singular pieces of data being stored in an individual variables.

Collections of data can also be stored in special data types, i.e., tuples, lists, sets, and dictionaries. Generally, you should try to store like with like, so each element in the data type should be the same kind of data, like an expression value from RNA-seq or a count of how many exons are in a gene or a read sequence.

1. In the interpreter, create and assign (`=`) values to variables with the following names:
    1. name
    2. age
    3. institute
    4. birth_country (example of variable name using a_underscore)
    5. favoriteColor (example of variable name using camelCase)

2. Use the `print()` function to print each variable to the screen.

Lists

- Lists are used to store an ordered, *indexed* collection of data.
- Lists are mutable: the number of elements in the list and what's stored in each element can change
- Lists are enclosed in **square brackets** and items are separated by commas

```
1  [ 'atg' , 'aaa' , 'agg' ]
```

INDEX	VALUE
0	atg
1	aaa
2	agg

> *The list index starts at 0*

The contents of a list can be assigned to a variable

```
codons = [ 'atg' , 'aaa' , 'agg' ]
```

1. In the interpreter, create a list and assign it to a variable.
    1. Play close attention to the the square brackets `[]` and the quotes `''`. (Either single or double quotes can be used here.)
    2. Be sure to give the variable a descriptive name.
2. Use the `print()` to print the list to the screen.

Tuples

- Tuples are similar to lists and contain ordered, *indexed* collection of data.
- **Tuples are immutable: you can't change the values or the number of values**
- A tuple is enclosed in **parentheses** and items are separated by commas.

```
( 'Jan' , 'Feb' , 'Mar' , 'Apr' , 'May' , 'Jun' , 'Jul' , 'Aug' , 'Sep' , 'Oct' , 'Nov' , 'Dec' )
```

INDEX	VALUE
0	Jan
1	Feb
2	Mar
3	Apr

4	May
5	Jun
6	Jul
7	Aug
8	Sep
9	Oct
10	Nov
11	Dec

Tuples can also be assigned to a variable:

```
months = ( 'Jan' , 'Feb' , 'Mar' , 'Apr' , 'May' , 'Jun' , 'Jul' , 'Aug' , 'Sep' , 'Oct' , 'Nov' , 'Dec' )
```

Dictionary

- Dictionaries are good for storing data that can be represented as a two-column table.
- They store unordered collections of key/value pairs.
- A dictionary is enclosed in **curly braces**.
- A colon is written between each key and value.
- Commas separate key:value pairs.

```
{ 'TP53' : 'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC' , 'BRCA1' : 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA' }
```

KEY	VALUE
TP53	GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC
BRCA1	GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA

Dictionaries can also be assigned to a variable:

```
genes = { 'TP53' : 'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC' , 'BRCA1' : 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA' }
```

1. In the interpreter create a dictionary and assign it to a variable.
    1. Play close attention to the curly braces (`{}`) and the quotes.
2. Use the `print()` function to print the contents to the screen.

Command line parameters: A Special Built-in List

Command line parameters follow the name of a script or program and have spaces between them. They allow a user to pass information to a script on the command line when that script is being run. Python stores all the pieces of the command line in a special list called `sys.argv` .

You need to import the sys module at the beginning of your script like this

```
#!/usr/bin/env python3
import sys
```

Let's imagine we have a script called friends.py. If you write this on the command line:

```
% friends.py Joe Anita
```

This happens inside the script:

> *the script name 'friends.py', and the strings 'Joe' and 'Anita' appear in a list called* `sys.argv` .

> *These are the command line parameters, or arguments you want to pass to your script.*
> `sys.argv[0]` *is the script name.*
> *You can access values of the other parameters by their indices, starting with 1, so* `sys.argv[1]` *contains 'Joe' and* `sys.argv[2]` *contains 'Anita'. You access elements in a list by adding square brackets and the numerical index after the name of the list. If you wanted to print a message saying these two people are friends, you might write some code like this*

```
#!/usr/bin/env python3
import sys
friend1 = sys.argv[1] # get first command line parameter
friend2 = sys.argv[2] # get second command line parameter
# now print a message to the screen
print(friend1,'and',friend2,'are friends')
```

The advantage of getting input from the user from the command line is that you can write a script that is general. It can print a message with any input the user provides. This makes it flexible. The user also supplies all the data the script needs on the command line so the script doesn't have to ask the user to input a name and wait til the user does this. The script can run on its own with no further interaction from the user. This frees the user to work on something else. Very handy!

1. Using your text editor, create a new python script. Be sure to include `#!/usr/bin/env python3` on the very first line. Make sure you give the script a name that ends with '.py'
2. Import the sys module by typing `import sys` .
3. Create a variable called favAnimal and assign the first command line argument to this variable, using `sys.argv[1]` . Need help? Google it.
4. Create a variable called favGene and assign the second command line argument to this variable using `sys.argv[2]` .

5. Print the two variables to the screen.

What kind of object am I working with?

You have an identifier in your code called `data`. Does it represent a string or a list or a dictionary? Python has a couple of functions that help you figure this out.

FUNCTION	DESCRIPTION
`type(data)`	tells you which class your object belongs to
`dir(data)`	tells you which methods are available for your object

We'll cover `dir()` in more detail later

```
>>> data = [2,4,6]
>>> type(data)
<class 'list'>
>>> data = 5
>>> type(data)
<class 'int'>
```

1. In the interpreter create a list, assign it to a variable named 'experiment'.
2. Use the `type()` function to help you determine what kind of object you have.
3. Overrite the contents of 'experiment' with another value.
4. Use the `type()` function to help you determine what kind of object you have.

Operators

An operator in a programming language is a symbol that tells the compiler or interpreter to perform specific mathematical, relational or logical operation and produce a result. Here we explain the concept of operators.

Arithmetic Operators

In Python we can write statements that perform mathematical calculations. To do this we need to use operators that are specific for this purpose. Here are arithmetic operators:

OPERATOR	DESCRIPTION	EXAMPLE	RESULT
`+`	Addition	`3+2`	5
`-`	Subtraction	`3-2`	1
`*`	Multiplication	`3*2`	6
`/`	Division	`3/2`	1.5
`%`	Modulus (divides left operand by right operand and returns the remainder)	`3%2`	1
`**`	Exponent	`3**2`	9
`//`	Floor Division (result is the quotient with digits after the decimal point removed. If one of the operands is negative, the result is floored, i.e., rounded away from zero	`3//2` ; `-11//3`	1 ; -4

1. In the interactive interpreter try a few of the above examples with new values.
2. How would you use modulus '(%)' to determine if a number is odd or even? Try `3%2` and `10%2`. Need help? Google It.

Assignment Operators

We use assignment operators to assign values to variables. You have been using the `=` assignment operator. Here are others:

OPERATOR	EQUIVALENT TO	EXAMPLE	RESULT EVALUATES TO
`=`	`a = 3`	`result = 3`	3
`+=`	`result = result +`	`result = 3 ; result +=`	5

	`result = result + 2`	`result = 3 ; result += 2`	5
`-=`	`result = result - 2`	`result = 3 ; result -= 2`	1
`*=`	`result = result * 2`	`result = 3 ; result *= 2`	6
`/=`	`result = result / 2`	`result = 3 ; result /= 2`	1.5
`%=`	`result = result % 2`	`result = 3 ; result %= 2`	1
`**=`	`result = result ** 2`	`result = 3 ; result **= 2`	9
`//=`	`result = result // 2`	`result = 3 ; result //= 3`	1

1. In the interactive interpreter try a few of the above examples with new values.
2. What are the two ways to add 4 to to this variable named number? (Use '+' and '+=')

   `number = 3`

Comparison Operators

These operators compare two values and returns true or false.

OPERATOR	DESCRIPTION	EXAMPLE	RESULT
`==`	equal to	`3 == 2`	False
`!=`	not equal	`3 != 2`	True
`>`	greater than	`3 > 2`	True
`<`	less than	`3 < 2`	False

>=	greater than or equal	3 >= 2	True
<=	less than or equal	3 <= 2	False

1. In the interactive interpreter try a few of the above examples with new values.

Logical Operators

Logical operators allow you to combine two or more sets of comparisons. You can combine the results in different ways. For example you can 1) demand that all the statements are true, 2) that only one statement needs to be true, or 3) that the statement needs to be false.

OPERATOR	DESCRIPTION	EXAMPLE	RESULT
and	True if left operand is True and right operand is True	3>=2 and 2<3	True
or	True if left operand is True or right operand is True	3==2 or 2<3	True
not	Reverses the logical status	not False	True

1. In the interactive interpreter try a few of the above examples with new values.

Membership Operators

You can test to see if a value is included in a string, tuple, or list. You can also test that the value is not included in the string, tuple, or list.

OPERATOR	DESCRIPTION
`in`	True if a value is included in a list, tuple, or string
`not in`	True if a value is absent in a list, tuple, or string

For Example:

```
>>> dna = 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
>>> 'TCT' in dna
True
>>>
>>> 'ATG' in dna
False
>>> 'ATG' not in dna
True
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
>>> 'atg' in codons
True
>>> 'ttt' in codons
False
```

1. Use the Interactive Interpretor to test to see if you can find an 'CAA' in the following DNA string:

```
GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA
```

2. How about 'GGG'?

3. Use the `in` operator to test to see if the codon 'ata' in this list? How about 'agg'?

   `codons = [ 'atg' , 'aaa' , 'agg' ]`

Operator Precedence

Operators are listed in order of precedence. Highest listed first. Not all the operators listed here are mentioned above.

OPERATOR	DESCRIPTION
`**`	Exponentiation (raise to the power)
`~` `+` `-`	Complement, unary plus and minus (method names for the last two are +@ and -@)
`*` `/` `%` `//`	Multiply, divide, modulo and floor division
`+` `-`	Addition and subtraction
`>>` `<<`	Right and left bitwise shift
`&`	Bitwise 'AND'
`^` `|`	Bitwise exclusive 'OR' and regular 'OR'
`<=` `<` `>` `>=`	Comparison operators
`<>` `==` `!=`	Equality operators
`=` `%=` `/=` `//=` `-=` `+=` `*=` `**=`	Assignment operators
`is`	Identity operator
`is not`	Non-identity operator
`in`	Membership operator
`not in`	Negative membership operator
`not` `or` `and`	logical operators

1. Without using the computer, what is the value of this expression:
   `3+5*2-5`
2. Test your answer in the interpreter

Truth

Lets take a step back, What is truth?

Everything is true, except for:

EXPRESSION	TRUE/FALSE
`0`	FALSE
`None`	FALSE
`False`	FALSE
`''` (empty string)	FALSE
`[]` (empty list)	FALSE
`()` (empty tuple)	FALSE
`{}` (empty dictionary)	FALSE

Which means that these are True:

EXPRESSION	TRUE/FALSE
`'0'`	TRUE
`'None'`	TRUE
`'False'`	TRUE
`'True'`	TRUE
`' '` (string of one blank space)	TRUE

Test for truth

`bool()` is a function that will test if a value is true.

```
>>> bool(True)
True
>>> bool('True')
True
>>>
>>>
>>> bool(False)
False
```

```
>>> bool('False')
True
>>>
>>>
>>> bool(0)
False
>>> bool('0')
True
>>>
>>>
>>> bool('')
False
>>> bool(' ')
True
>>>
>>>
>>> bool(())
False
>>> bool([])
False
>>> bool({})
False
```

1. In the interpretor use `bool()` to test a variety of values like '', 0, 0.0, FALSE, false, True, true, 'True', 'False' to see if they evaluate to True or False.

Logic: Control Statements

Control Statements are used to direct the flow of your code and create the opportunity for decision making. Control statements foundation is build on truth.

If Statement

- Use the `if` Statement to test for truth and to execute lines of code

if true.
- When the expression evaluates to true each of the statements **indented** below the `if` statment, also known as the nested statement block, will be executed.

if

```
if expression :
  statement
  statement
```

For Example:

```
dna = 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
if 'AGC' in dna:
  print('found AGC in your dna sequence')
```

Returns:

```
found AGC in your dna sequence

```

1. In your text editor create a script that prints 'FOUND IT!!' `if` this string of nucleotides: 'TTCGTATT', is found `in` this string of DNA: 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'

else

- The `if` portion of the if/else statement behave as before.
- The first indented block is executed if the condition is true.
- If the condition is false, the second indented else block is executed.

```
dna = 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
if 'ATG' in dna:
  print('found ATG in your dna sequence')
else:
  print('did not find ATG in your dna sequence')
```

Returns:

```
did not find ATG in your dna sequence

```

1. Using a text editor, write a script that
    - Assigns a value to a variable
    - Has a if/else statment in which:
        - It prints out a confirmation of truth if the value is true
        - It prints out "Not True" if the value is not true.

if/elif

- The if condition is tested as before and the indented block is executed if the condition is true.
- If it's false, the indented block following the elif is executed if the first elif condition is true.
- Any remaining elif conditions will be tested in order until one is found to be true. If none is true, the else indented block is executed.

```
count = 60
if count < 0:
  message = "is less than 0"
  print(count, message)
elif count < 50:
  message = "is less than 50"
  print (count, message)
elif count > 50:
  message = "is greater than 50"
  print (count, message)
else:
  message = "must be 50"
  print(count, message)
```

Returns:

```
60 is greater than 50

```

1. Change the variable count to 20, which statement block gets executed?
2. Change count to 50, what happens?
3. CHALLENGE QUESTION: Create a script that has a if/elif/else statement that
    - Tests to see if a number is positive or negative (use modulus)
    - if it is positive also test if it is bigger or smaller than 50
        - if it is smaller also test if it is an even number
        - if it is larger also test if it is divisible by 3 (use modulus).

Numbers

Python recognizes 3 types of numbers: integers, floating point numbers, and complex numbers.

integer

- known as an int
- an int can be positive or negative
- and **does not** contain a decimal point or exponent.

floating point number

- known as a float
- a floating point number can be positive or negative
- and **does** contain a decimal point (`4.875`) or exponent (`4.2e-12`)

complex number

- known as complex
- is in the form of a+bi where bi is the imaginary part.

Conversion functions

Sometimes one type of number needs to be changed to another for a function to be able to do work on it. Here are a list of functions for converting number types:

FUNCTION	DESCRIPTION
`int(x)`	to convert x to a plain integer
`float(x)`	to convert x to a floating-point number
`complex(x)`	to convert x to a complex number with real part x and imaginary part zero
`complex(x, y)`	to convert x and y to a complex number with real part x and imaginary part y

```
>>> int(2.3)
2
>>> float(2)
2.0
>>> complex(2.3)
(2.3+0j)
>>> complex(2.3,2)
(2.3+2j)
```

Numeric Functions

Here are a list of functions that take numbers as arguments.

FUNCTION	DESCRIPTION
`abs(x)`	The absolute value of x: the (positive) distance between x and zero.
`round(x[,n])`	x rounded to n digits from the decimal point. round() rounds to an even integer if the value is exactly between two integers, so round(0.5) is 0 and round(-0.5) is 0. round(1.5) is 2. **Rounding to a fixed number of decimal places can give unpredictable results.**
`max(x1, x2,...)`	The largest positive argument is returned
`min(x1, x2,...)`	The smallest argument is returned

```
>>> abs(2.3)
2.3
>>> abs(-2.9)
2.9
>>> round(2.3)
2
>>> round(2.5)
2
>>> round(2.9)
3
>>> round(-2.9)
-3
>>> round(-2.3)
-2
```

```
>>> round(-2.009,2)
-2.01
>>> round(2.675, 2)  # note this rounds down
2.67
>>> max(4,-5,5,1,11)
11
>>> min(4,-5,5,1,11)
-5
```

Many numeric functions are not built into the Python core and need to be included in our script if we want to use them. To include them at the tip of the script type: `import math`

These next functions are found in the math module and need to be imported. To use these functions, prepend the function with the module name, i.e, `math.ceil(15.5)`

MATH.FUNCTION	DESCRIPTION
`math.ceil(x)`	return the smallest integer greater than or equal to x is returned
`math.floor(x)`	return the largest integer less than or equal to x.
`math.exp(x)`	The exponential of x: e^x is returned
`math.log(x)`	the natural logarithm of x, for x > 0 is returned
`math.log10(x)`	The base-10 logarithm of x for x > 0 is returned
`math.modf(x)`	The fractional and integer parts of x are returned in a two-item tuple.
`math.pow(x, y)`	The value of x raised to the power y is returned
`math.sqrt(x)`	Return the square root of x for x >= 0

```
>>> import math
>>>
>>> math.ceil(2.3)
3
>>> math.ceil(2.9)
3
>>> math.ceil(-2.9)
-2
>>> math.floor(2.3)
2
```

```
>>> math.floor(2.9)
2
>>> math.floor(-2.9)
-3
>>> math.exp(2.3)
9.974182454814718
>>> math.exp(2.9)
18.17414536944306
>>> math.exp(-2.9)
0.05502322005640723
>>>
>>> math.log(2.3)
0.8329091229351039
>>> math.log(2.9)
1.0647107369924282
>>> math.log(-2.9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>>
>>> math.log10(2.3)
0.36172783601759284
>>> math.log10(2.9)
0.4623979978989561
>>>
>>> math.modf(2.3)
(0.2999999999999998, 2.0)
>>>
>>> math.pow(2.3,1)
2.3
>>> math.pow(2.3,2)
5.289999999999999
>>> math.pow(-2.3,2)
5.289999999999999
>>> math.pow(2.3,-2)
0.18903591682419663
>>>
>>> math.sqrt(25)
5.0
>>> math.sqrt(2.3)
1.51657508881031
>>> math.sqrt(2.9)
1.70293863659264
```

1. In the interactive interpreter try a few of the above examples with new values.

Comparing two numbers

Often times it is necessary to compare two numbers and find out if the first number is less than, equal to, or greater than the second.

The simple function `cmp(x,y)` is not available in Python 3.

Use this idiom instead:

```
1  cmp = (x>y)-(x<y)
```

It returns three different values depending on x and y

- if x<y -1 is returned
- if x>y 1 is returned
- x == y 0 is returned

Sequences

In the next section, we will learn about strings, tuples and lists. These are all examples of python sequences. A string is a sequence of characters `'ACGTGA'`, a tuple `(0.23, 9.74, -8.17, 3.24, 0.16)` and a list `['dog', 'cat', 'bird']` are sequences of any kind of data. We'll see much more detail in a bit.

In Python a type of object gets operations that belong to that type. Sequences have sequence operations so strings can also use sequence operations. Strings also have their own specific operations.

You can ask what the length of any sequence is

```
>>>len('ACGTGA') # length of a string
6
>>>len( (0.23, 9.74, -8.17, 3.24, 0.16) )  # length of a tuple, needs two parentheses ( (TUPLE) )
5
>>>len(['dog', 'cat', 'bird'])  # length of a list ( [LIST] )
3
```

1. What is the length of this sequence?

```
DNA = 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA' ;
```

2. What is the length of this sequence?

```
numbers = [ 3 , 61 , 7 , 27 , 83 , 6 , 0 , 175 , 9 , 28 , 4 ] ;
```

3. How about this sequence?

```
DNA = [ 'GTACCT' , 'TG' , 'ATTTCGTAT' , 'TCTGAGAG' , 'GCT', 'GCT', 'GCTTAGCGGTAGCC' , 'CCTTG','GTTTCCGTG','GCAA','CGGAAAA' ];
```

4. Which of the above lengths are a count of actual characters and which are a count of elements

Strings

- A string is a series of characters starting and ending with single or double quotation marks.
- Strings are an example of a Python sequence. A sequence is defined as a positionally ordered set. This means each element in the set has a position, starting with zero, i.e. 0,1,2,3 and so on until you get to the end of the string.

Quotation Marks

- Single (')
- Double (")
- Triple (''' or """)

Notes about quotes:

- Single and double quotes are equivalent.
- A variable name inside quotes is just the string identifier, not the value stored inside the variable.
- Triple quotes are used before and after a string that spans multiple lines.

Use of quotation examples:

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences. And goes
on and on.
"""
```

Strings and the `print()` function

We saw examples of `print()` earlier. Lets talk about it a bit more. `print()` is a function that takes one or more comma-separated arguments.

Let's use the `print()` function to print a string.

```
>>>print("ATG")
ATG
```

Let's assign a string to a variable and print the variable.

```
>>>dna = 'ATG'
ATG
>>> print(dna)
ATG
```

What happens if we put the variable in quotes?

```
>>>dna = 'ATG'
ATG
>>> print("dna")
dna
```

> *The literal string 'dna' is printed to the screen, not the contents 'ATG'*

Let's see what happens when we give `print()` two literal strings as arguments.

```
>>> print("ATG","GGTCTAC")
ATG GGTCTAC
```

> *We get the two literal strings printed to the screen separated by a space*

What if you do not want your strings separated by a space? Use the concatenation operator to concatenate the two strings before or within the `print()` function.

```
>>> print("ATG"+"GGTCTAC")
ATGGGTCTAC
>>> combined_string = "ATG"+"GGTCTAC"
ATGGGTCTAC
>>> print(combined_string)
ATGGGTCTAC
```

> *We get the two strings printed to the screen without being separated by a space.*
> *You can also use this*

```
>>> print('ATG','GGTCTAC',sep='')
ATGGGTCTAC
```

1. In the interpreter use the `print()` function to print a comma separated list of arguments.
2. Print the same two strings, but instead of separating the arguments with a comma use a '+'.
3. What is the difference between 1 and 2?
4. Try using the `print()` function's `sep=` argument. Print your two strings separated with commas ',' . Then again separated by double dashes '- -'.

Now, lets print a variable and a literal string.

```
>>>dna = 'ATG'
ATG
>>> print(dna,'GGTCTAC')
ATG GGTCTAC
```

> *We get the value of the variable and the literal string printed to the screen separated by a space*

How would we print the two without a space?

```
>>>dna = 'ATG'
ATG
>>> print(dna + 'GGTCTAC')
ATGGGTCTAC
```

Something to think about: Values of variables are variable. Or in other words, they are mutable, changeable.

```
>>>dna = 'ATG'
ATG
>>> print(dna)
ATG
>>>dna = 'TTT'
TTT
>>> print(dna)
TTT
```

> *The new value of the variable 'dna' is printed to the screen when `dna` is an argument for the `print()` function.*

Special/Escape Characters

How would you include a new line, carriage return, or tab in your string?

ESCAPE CHARACTER	DESCRIPTION
\n	New line
\r	Carriage Return
\t	Tab

Let's include some escape characters in our strings and `print()` functions.

```
>>> string_with_newline = 'this sting has a new line\nthis is the second line'
>>> print(string_with_newline)
this sting has a new line
this is the second line
```

> *We printed a new line to the screen*

`print()` adds spaces between arguments and a new line at the end for you. You can change these with `sep=` and `end=`. Here's an example: `print('one line', 'second line' , 'third line', sep='\n', end = '')`

Another way to do this is to express a multi-line string enclosed in triple quotes (""").

```
>>> print("""this string has a new line
... this is the second line""")
this string has a new line
this is the second line
```

Let's print a tab character (\t).

```
>>> line = "value1\tvalue2\tvalue3"
>>> print(line)
value1  value2  value3
```

> *We get the three words separated by tab characters. A common format for data is to separate columns with tabs like this.*

You can add a backslash before any character to force it to be printed as a literal. This is called 'escaping'. This is only really useful for printing literal quotes ' and "

```
>>> print('this is a \'word\'')  # if you want to print a ' inside '...'
this is a 'word'
>>> print("this is a 'word'") # maybe clearer to print a ' inside "..."
this is a 'word'
```

> *In both cases actual single quote character are printed to the screen*

If you want every character in your string to remain exactly as it is, declare your string a raw string literal with 'r' before the first quote. This looks ugly, but it works.

```
>>> line = r"value1\tvalue2\tvalue3"
>>> print(line)
value1\tvalue2\tvalue3
```

> *Our escape characters '\t' remain as we typed them, they are not converted to actual tab characters.*

1. In the intepreter, use the `print()` function to print a string that is stored in a variable (dna="ATGCGGTGGT") and a literal string ("My DNA Fragment is:").
2. What happens if you put quotes around "dna"? Why?

Concatenation

To concatenate strings use the concatenation operator '+'

```
>>> promoter= 'TATAAA'
>>> upstream = 'TAGCTA'
>>> downstream = 'ATCATAAT'
>>> dna = upstream + promoter + downstream
>>> print(dna)
TAGCTATATAAAATCATAAT
```

> *The concatenation operator can be used to combine strings. The newly combined string can be stored in a variable.*

The difference between string + and integer +

What happens if you use `+` with numbers (these are integers or ints)?

```
>>> 4+3
7

```

For strings, `+` concatenates; for integers, `+` adds.

You need to convert the numbers to strings before you can concatenate them

```
>>> str(4) + str(3)
'43'
```

Determine the length of a string

Use the `len()` function to calculate the length of a string. This function takes a sequence as an argument and returns an int

```
>>> print(dna)
TAGCTATATAAAATCATAAT
>>> len(dna)
20
```

> *The length of the string, including spaces, is calculated and returned.*

The value that `len()` returns can be stored in a variable.

```
>>> dna_length = len(dna)
>>> print(dna_length)
20
```

You can mix strings and ints in `print()`, but not in concatenation.

```
>>> print("The lenth of the DNA sequence:" , dna , "is" , dna_length)
The lenth of the DNA sequence: TAGCTATATAAAATCATAAT is 20
```

1. Concatenate two strings and store in a variable then print the new string.

Changing String Case

Changing the case of a string is a bit different than you might first expect. For example, to lowercase a string we need to use a method. A method is a function that is specific to an object. When we assign a string to a variable we are creating an instance of a string object. This object has a series of methods that will work on the data that is stored in the object. The `lower()` function is a string method.

Let's create a new string object.

```
dna = "ATGCTTG"
```

> *Look familiar? It should!!! Creating a string object is what we have been doing all along!!! Jeez!!!*

Now that we have a string object we can use string methods. The way you use a method is to put a '.' between the object and the method name.

```
>>> dna = "ATGCTTG"
>>> dna.lower()
'atgcttg'
```

> *the lower() method returns the contents stored in the 'dna' variable in lowercase.*

The contents of the 'dna' variable have not been changed. Strings are immutable. If you want to keep the lowercased version of the string, store it in a new variable.

```
>>> print(dna)
ATGCTTG
>>> dna_lowercase = dna.lower()
>>> print(dna)
ATGCTTG
>>> print(dna_lowercase)
atgcttg
```

The string method can be nested inside of other functions.

```
>>> dna = "ATGCTTG"
>>> print(dna.lower())
atgcttg
```

> *The contents of 'dna' are lowercased and passed to the `print()` function.*

If you try to use a string method on a object that is not a string you will get an error.

```
>>> nt_count = 6
>>> dna_lc = nt_count.lower()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'lower'
```

> *You get an AttributeError when you use a method on the an incorrect object type. We are told that the int object (an int is returned by `len()`) does not have a function called lower.*

Now let's uppercase a string.

```
>>> dna = 'attgct'
>>> dna.upper()
'ATTGCT'
>>> print(dna)
attgct
```

> *The contents of the variable 'dna' were returned in upper case. The contents of 'dna' were not altered.*

How do you know what methods are available? Use the `dir()` function.

```
>>> dir(dna)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__',
'__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center',
'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Find and Count

`count(str)` returns the number of exact matches of `str` it found (as an int)

```
>>> dna = 'ATGCTGCATT'
>>> dna.count('T')
4
```

> *The number of times 'T' is found is returned. The string stored in 'dna' is not altered.*

1. Count the number of As in a DNA string (dna = 'ATGCTGCATT').
2. Lowercase and print the DNA string.

Find and Replace

`replace(str1,str2)` returns a new string with all matches of `str1` in a string replaced with `str2`.

```
>>> dna = 'ATGCTGCATT'
>>> dna.replace('T','U')
'AUGCUGCAUU'
>>> print(dna)
ATGCTGCATT
>>> rna = dna.replace('T','U')
>>> print(rna)
AUGCUGCAUU
```

> *All occurrences of T are replaced by U. The new string is returned. The original string has not actually been altered. If you want to reuse the new string, store it in a variable.*

Extracting a Substring, or Slicing

Parts of a string can be located based on position and returned. This is because a string is a sequence. Coordinates start at 0. You add the coordinate in square brackets after the string's name.

This string 'ATTAAAGGGCCC' is made up of the following sequence of characters, and positions (starting at zero).

POSITION/INDEX	CHARACTER

0	A
1	T
2	T
3	A
4	A
5	A
6	G
7	G
8	G
9	C
10	C
11	C

Let's return the 4th, 5th, and 6th nucleotides. To do this, we need to count like a computer and start our string at 0 and return the 3rd, 4th, and 5th characters. This will be everything from 3 to 6. Python counts the gaps before each character in the string, starting at 0.

```
>>> dna = 'ATTAAAGGGCCC'
>>> sub_dna = dna[3:6]
>>> print(sub_dna)
AAA
```

> *The characters with indices 3, 4, 5 are returned. Or in other words, every character starting at index 3 and up to but not including, the index of 6 are returned.*

Let's return the first 6 characters.

```
>>> dna = 'ATTAAAGGGCCC'
>>> sub_dna = dna[0:6]
>>> print(sub_dna)
ATTAAA
```

> *Every character starting at index 0 and up to but not including index 6 are returned. This is the same as dna[:6]*

Let's return every character from index 6 to the end of the string.

```
>>> dna = 'ATTAAAGGGCCC'
>>> sub_dna = dna[6:]
>>> print(sub_dna)
GGGCCC
```

> *When the second argument is left blank, every character from index 6 and greater is returned.*

Let's return the last 3 characters.

```
>>> sub_dna = dna[-3:]
>>> print(sub_dna)
CCC
```

> *When the second argument is left blank and the first argument is negative (-X), X characters from the end of the string are returned.*

1. Extract the first 6 nucleotides from this DNA string: ATTAAAGGGCCC and save the substring in a variable.
2. Replace all Ts with U's. Print the new string.

Locate and Report

The positional index of an exact string in a larger string can be found and returned with the string method `find()`. An exact string is given as an argument and the index of its first occurrence is returned. -1 is returned if it is not found.

```
>>> dna = 'ATTAAAGGGCCC'
>>> dna.find('T')
1
>>> dna.find('N')
-1
```

> *The substring 'T' is found for the first time at index 1 in the string 'dna' so 1 is returned. The substring 'N' is not found, so -1 is returned.*

Other String Methods

Since these are methods, be sure to use in this format string.method().

FUNCTION	DESCRIPTION
`s.strip()`	returns a string with the whitespace removed from the start and end
`s.isalpha()`	tests if all the characters of the string are alphabetic characters. Returns True or False.
`s.isdigit()`	tests if all the characters of the string are numeric characters. Returns True or False.
`s.startswith('other_string')`	tests if the string starts with the string provided as an argument. Returns True or False.
`s.endswith('other_string')`	tests if the string ends with the string provided as an argument. Returns True or False.
`s.split('delim')`	splits the string on the given exact delimiter. Returns a list of substrings. If no argument is supplied, the string will be split on whitespace.
`s.join(list)`	opposite of `split()`. The elements of a list will be concatenated together using the string stored in 's' as a delimiter.

1. Try out the `split()` method on this string 'one,two,three,four,five and six'. Split the string on commas ','.

```
>>> string='one,two,three,four,five and six'
>>> string.split(',')
['one', 'two', 'three', 'four', 'five and six']
```

2. Try the `join()` method. Join the list returned from your split with a string of double dashes ' - -'. Think carefully about this. `join()` is a **string method**. Is the list or the double dash a string? Which do you use to the right of the '.join'. Need help? Google it.

String Formatting

Strings can be formated using the `format()` function. Pretty intuitive, but wait til you see the details! For example, if you want to include literal stings and variables in your print statement and do not want to concatenate or use multiple arguments in the `print()` function you can use string formatting.

```
>>> dna = 'TGAACATCTAAAAGATGAAGTTT'
>>> dna_len = len(dna)
>>> gene_name = 'Brac1'
>>> string = "This sequence: {} is {} nucleotides long and is found in {}."
>>> string.format(dna,dna_len,gene_name)
'This sequence: TGAACATCTAAAAGATGAAGTTT is 23 nucleotides long and is found in Brca1.'
>>> print(string) # string.format() does not alter string
This sequence: {} is {} nucleotides long and is found in {}.
>>> new_string = string.format(dna,dna_len,gene_name)
>>> print(new_string)
This sequence: TGAACATCTAAAAGATGAAGTTT is 23 nucleotides long and is found in Brca1.
```

We put together the three variables and literal strings into a single string using the function `format()`. The original string is not altered, a new string is returned that incorporates the arguments. You can save the returned value in a new variable. Each `{}` is a placeholder for the strings that need to be inserted.

Something nice about `format()` is that you can print int and string variable types without converting first.

You can also directly call the format function on a string inside a print function. Here are two examples

```
>>> string = "This sequence: {} is {} nucleotides long and is found in {}."
>>> print(string.format(dna,dna_len,gene_name))
This sequence: TGAACATCTAAAAGATGAAGTTT is 23 nucleotides long and is found in Brca1.
```

1. Try out the `format()` method. Use the following string and the variables from the above example. Type it, don't copy it.

```
>>> string = "This sequence: {} is {} nucleotides long and is found in {}."
>>> print(string.format(dna,dna_len,gene_name))
```

2. Change the order of the the three variables. What gets printed?
3. Create a new string for the three variables to be printed within.

Lists and Tuples

Lists

Lists are data types that store a collection of data.

- Lists are used to store an ordered, *indexed* collection of data.
- Values are separated by commas
- Values are enclosed in square brackets '[]'
- Lists can grow and shrink
- Values are mutable

Tuples

- Tuples are used to store an ordered, *indexed* collection of data.
- Values are separated by commas
- Values are enclosed in square brackets '()'
- Tuples can **NOT** grow or shrink
- Values are immutable

Back to Lists

Accessing Values in Lists

To retrieve a single value in a list use the value's index in this format list[index]. This will return the value at the specified index, starting with 0.

Here is a list:

```
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
```

> *There are 3 values with the indices of 0, 1, 2*

INDEX	VALUE
0	atg
1	aaa
2	agg

Let's access the 0th value, this is the element in the list with index 0. You'll need an index number (`0`) inside square brackets like this `[0]` . This goes after the name of the list (`codons`)

```
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
>>> codons[0]
'atg'
```

The value can be saved for later use by storing in a variable.

```
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
>>> first_codon = codons[0]
>>> print(first_codon)
atg
```

> *Each value can be saved in a new variable to use later.*

The values can be retrieved and used directly.

```
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
>>> print(codons[0])
atg
>>> print(codons[1])
aaa
>>> print(codons[2])
agg
```

> *The 3 values are independently accesses and immediately printed. They are not stored in a variable.*

If you want to access the values starting at the end of the list, use negative indices.

```
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
>>> print(codons[-1])
agg
>>> print(codons[-2])
aaa
```

> *Using a negative index will return the values from the end of the list. For example, -1 is the index of the last value 'agg'. This value also has an index of 2.*

Changing Values in a List

Individual values can be changed using the value's index and the assignment operator.

```
>>> print(codons)
['atg', 'aaa', 'agg']
>>> codons[2] = 'cgc'
>>> print(codons)
['atg', 'aaa', 'cgc']
```

What about trying to assign a value to an index that does not exist?

```
>>> codons[5] = 'aac'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

> *codon[5] does not exist, and when we try to assign a value to this index we get an IndexError. If you want to add new elements to the end of a list use* `codons.append('taa')` *or* `codons.extend(list)` *. See below for more details.*

1. In the interpreter create a list of your favorite things.
2. Use the `print()` function print out the middle element.
3. Now replace the middle element with a different item, your favorite song, or song bird.
4. Use the same print statement from #2 to print your new list.

Extracting a Subset of a List, or Slicing

This works in exactly the same way with lists as it does with strings. This is because both are sequences, or ordered collections of data with positional information. Remember Python counts the divisions between the elements, starting with 0.

INDEX	VALUE
0	atg
1	aaa
2	agg
3	aac
4	cgc
5	acg

```
>>> codons = [ 'atg' , 'aaa' , 'agg' , 'aac' , 'cgc' , 'acg']
>>> print (codons[1:3])
['aaa', 'agg']
>>> print (codons[3:])
['aac', 'cgc', 'acg']
>>> print (codons[:3])
['atg', 'aaa', 'agg']
>>> print (codons[0:3])
['atg', 'aaa', 'agg']
```

> *`codons[1:3]` returns every value starting with the value of codons[1] up to but not including codons[3] `codons[3:]` returns every value starting with the value of codons[3] and every value after. `codons[:3]` returns every value up to but not including codons[3] `codons[0:3]` is the same as `codons[:3]`*

List Operators

OPERATOR	DESCRIPTION	EXAMPLE
`+`	Concatenation	`[10, 20, 30] + [40, 50, 60]` returns `[10, 20, 30, 40, 50, 60]`
`*`	Repetition	`['atg'] * 4` returns `['atg','atg','atg','atg']`
`in`	Membership	`20 in [10, 20, 30]` returns `True`

List Functions

FUNCTIONS	DESCRIPTION	EXAMPLE
`len(list)`	returns the length or the number of values in list	`len([1,2,3])` returns `3`
`max(list)`	returns the value with the highest ASCII value (=latest in ASCII alphabet)	`max(['a','A','z'])` returns `'z'`
`min(list)`	returns the value with the lowest ASCII value (=earliest in ASCII alphabet)	`min(['a','A','z'])` returns `'A'`
`list(seq)`	converts a tuple into a list	`list(('a','A','z'))` returns `['a', 'A', 'z']`
`sorted(list, key=None,`	returns a sorted list based on the key provided	`sorted(['a','A','z'])` returns `['A', 'a', 'z']`

`reverse=False)`		
	`str.lower()` makes all the elements lowercase before sorting	`sorted(['a','A','z'],key=str.lower)` returns `['a', 'A', 'z']`

List Methods

Remember methods are used in the following format list.method().

For these examples use: `nums = [1,2,3]` and `codons = [ 'atg' , 'aaa' , 'agg' ]`

METHOD	DESCRIPTION	EXAMPLE
`list.append(obj)`	appends an object to the end of a list	nums.append(9) ; print(list) ; returns [1,2,3,9]
`list.count(obj)`	counts the occurrences of an object in a list	nums.count(2) returns 1
`list.index(obj)`	returns the lowest index where the given object is found	nums.index(2) returns 1
`list.pop()`	removes and returns the last value in the list. The list is now one element shorter	nums.pop() returns 3
`list.insert(index, obj)`	inserts a value at the given index. Remember to think about the divisions between the elements	list.insert(0,100) ; print(list) returns [100, 1, 2, 3]
`list.extend(new_list)`	appends `new_list` to the end of `list`	list.extend([7, 8]) ; print(list) returns [1, 2, 3, 7,8]
`list.pop(index)`	removes and returns the value of the index argument. The list is now 1 value shorter	nums.pop(0) returns 1
`list.remove(obj)`	finds the lowest index of the given object and removes it from the list. The list is now one element shorter	codons.remove('aaa') ; print(codons) returns ['atg' , 'agg']
`list.reverse()`	reverses the order of the list	nums.reverse() ; print(nums) returns [3,2,1]
	Returns a shallow copy of list	

`list.copy()`	Returns a shallow copy of list. Shallow vs Deep only matters in multidimensional data structures.	
`list.sort([func])`	sorts a list using the provided function. Does not return a list. The list has been changed. Advanced list sort will be covered once writing your own functions has been discussed.	codons.sort() ; print(codons) returns ['aaa', 'agg', 'atg']

Be careful how you make a copy of your list

```
>>> my_list=['a', 'one', 'two']
>>> copy_list=my_list
>>> copy_list.append('1')
>>> print(my_list)
['a', 'one', 'two', '1']
>>> print(copy_list)
['a', 'one', 'two', '1']
```

> *Not what you expected?! Both lists have changed because we only copied a pointer to the original list when we wrote* `copy_list=my_list`.

Let's copy the list using the `copy()` method.

```
>>> my_list=['a', 'one', 'two']
>>> copy_list=my_list.copy()
>>> copy_list.append('1')
>>> print(my_list)
['a', 'one', 'two']
```

> *There we go, we get what we expect this time!*

1. In the interpreter create a list.
2. Add a new element to the end. Read about append().
3. Add a new element to the beginning. Read about insert().
4. Add a new element somewhere other than the beginning or the end.
5. Remove an element from the end. Read about pop().
6. Remove an element from the beginning.
7. Remove an element from somewhere other than the beginning or the end.

Building a List one Value at a Time

Now that you have seen the `append()` function we can go over how to build a list one value at a time.

```
>>> words = []
>>> print(words)
[]
>>> words.append('one')
>>> words.append('two')
>>> print(words)
['one', 'two']
```

> *We start with a an empty list called 'words'. We use `append()` to add the value 'one' then to add the value 'two'. We end up with a list with two values. You can add a whole list to another list with `words.extend(['three','four','five'])`*

Loops

All of the coding that we have gone over so far has been executed line by line. Sometimes there are blocks of code that we want to execute more than once.

Repetitive tasks: There are times when you will want to do the same set of task over and over. In this example we are printing a count, then incrementing, then printing, then incrementing.

```
count = 0
print("count:" , count)
count+=1
print("count:" , count)
count+=1
print("count:" , count)
count+=1
print("count:" , count)
count+=1
print("count:" , count)
count+=1
print("Done")
```

Loops let us simplify this type of activity.

There are two loop types:

1. while loop
2. for loop

While loop

The while loop will continue to execute a block of code as long as the test expression evaluates to `True`.

While Loop Syntax

```
while expression:
 statement1
 statement2
 more_statements
rest_of_code_goes_here
```

> *The condition is the expression. The while loop block of code is the collection of indented statements following the expression.*

Code:

```
#!/usr/bin/env python3

count = 0
while count < 5:
  print("count:" , count)
  count+=1
print("Done")
```

Output:

```
$ python while.py
count: 0
count: 1
count: 2
count: 3
count: 4
Done
```

> *The while condition was true 5 times and the while block of code was executed 5 times.*
>
> - *count is equal to 0 when we begin*
> - *0 is less than 5 so we execute the while block*
> - *count is printed*
> - *count is incremented (count = count + 1)*
> - *count is now equal to 1.*
> - *1 is less than 5 so we execute the while block for the 2nd time.*
> - *this continues until count is 5.*
> - *5 is not less than 5 so we exit the while block*
> - *The first line following the while statement is executed, "Done" is printed*

Infinite Loops

An infinite loop occurs when a while condition is always true. Here is an example of an infinite loop.

```
#!/usr/bin/env python3

count = 0
while count < 5:
  print("count:" , count)
print("Done")
```

Output:

```
$ python infinite.py
count: 0
count: 0
count: 0
count: 0
count: 0
count: 0
count: 0
count: 0
...
...
```

> *What caused the expression to always be `True`? The statement that increments the count is missing, so it will always be smaller than 5. To stop the code from forever printing use ctrl+c.*

1. In your text editor create a script that uses a `while` loop to print out the numbers 1 to 100.
2. CHALLENGE QUESTION: Write a script that uses a `while loop` to calculate the factorial of 1000.

For Loops

A for loop is a loop that executes the for block of code for every member of a sequence, for example the elements of a list or the letters in a string.

For Loop Syntax

```
for iterating_variable in sequence:
  statement(s)
```

An example of a sequence is a list. Let's use a for loop with a list of words.

Code:

```
#!/usr/bin/env python3

words = ['zero','one','two','three','four']
for word in words:
  print(word)
```

> *Notice how I have named my variables, the list is plural and the iterating variable is singular*

Output:

```
python3 list_words.py
zero
one
two
three
four
```

This is next example is using a for loop to iterating over a string. Remember a string is a sequence like a list. Each character has a position. Look back at "Extracting a Substring, or Slicing" in the Strings section to see other ways that strings can be treated like lists.

Code:

```
#!/usr/bin/env python3

dna = 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
for nt in dna:
  print(nt)
```

Output:

```
$ python3 for_string.py
G
T
A
C
C
T
T
...
...
```

> *This is an easy way to access each character in a string. It is especially nice for DNA sequences.*

Another example of iterating over a list of variables, this time numbers.

Code:

```
#!/usr/bin/env python3

numbers = [0,1,2,3,4]
for num in numbers:
  print(num)
```

Output:

```
$ python3 list_numbers.py
0
1
2
3
4
```

Python has a function called `range()` that will return numbers that can be converted to a list.

```
>>> range(5)
range(0, 5)
>>> list(range(5))
[0, 1, 2, 3, 4]
```

The function `range()` can be used in conjunction with a for loop to iterate over a range of numbers. Range also starts at 0 and thinks about the gaps between the numbers.Code:

```
#!/usr/bin/env python3

for num in range(5):
  print(num)
```

Output:

```
$ python list_range.py
0
1
2
3
4
```

> *As you can see this is the same output as using the list `numbers = [0, 1, 2, 3, 4]` And this has the same functionality as a while loop with a condition of `count = 0` ; `count < 5` .*

This is the equivalent while loop

Code:

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

Output:

```
0
1
2
3
4
```

1. Write a script that iterate through and prints each element of this list using a `for` loop: [101,2,15,22,95,33,2,27,72,15,52]
    - Now print out only the values that are even (use modulus operator).

2. Write a new loop that uses `range()` to print out every number bewteen 1 and 100.
3. Make a list with the following data `['ATGCCCGGCCCGGC','GCGTGCTAGCAATACGATAAACCGG', 'ATATATATCGAT','ATGGGCCC']` .
    - Use a `for` loop to iterate through each element of this list
    - Print out each element
    - Print out the length and the sequence i.e., "4\tATGC\n"

Loop Control

Loops control statements allow for altering the normal flow of execution.

CONTROL STATEMENT	DESCRIPTION
`break`	A loop is terminated when a break statement is executed. All the lines of code after the break, but within the loop block are not executed. No more iteration of the loop are preformed
`continue`	A single iteration of a loop is terminated when a continue statement is executed. The next iteration will proceed normally.

Loop Control: Break

When a `break` is encountered no more code within the loop with be executed. The loop is done!

Code:

```
#!/usr/bin/env python3

count = 0
while count < 5:
  print("count:" , count)
  count+=1
  if count == 3:
    break
print("Done")
```

Output:

```
$ python break.py
count: 0
count: 1
count: 2
Done
```

> *when the count is equal to 3, the execution of the while loop is terminated, even though the initial condition (count < 5) is still True.*

Loop Control: Continue

When a `continue` is encounter the current iteration of the loop is done, nothing below the `continue` will be executed, but the next loop will proceed normally.

Code:

```
#!/usr/bin/env python3

count = 0
while count < 5:
  print("count:" , count)
  count+=1
  if count == 3:
    continue
  print("line after our continue")
print("Done")
```

Output:

```
$ python continue.py
count: 0
line after our continue
count: 1
line after our continue
count: 2
count: 3
line after our continue
count: 4
line after our continue
Done
```

> *When the count is equal to 3 the continue is executed. This causes all the lines within the loop block to be skipped. "line after our continue" is not printed when count is equal to 3. The next loop is executed normally.*

Iterators

An iterable is any data type that is can be iterated over, or can be used in iteration. An iterable can be made into an iterator with the `iter()` function. This means you can use the `next()` function.

```
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
>>> codons_iterator=iter(codons)
>>> next(codons_iterator)
'atg'
>>> next(codons_iterator)
'aaa'
>>> next(codons_iterator)
'agg'
>>> next(codons_iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

> *An iterator allows you to get the next element in the iterator until there are no more elements. If you want to go through each element again, you will need to redefine the iterator.*

Example of using an iterator in a for loop:

```
codons = [ 'atg' , 'aaa' , 'agg' ]
>>> codons_it = iter(codons)
>>> for codon in codons_it :
...   print( codon )
...
atg
aaa
agg
```

> *This is nice if you have a large large large list that you don't want to keep in memory. An iterator allows you to go through each element but not keep the entire list in memory. Without iterators the entire list is in memory.*

List Comprehension

List comprehension is a way to make a list without typing out each element. There are many many ways to use list comprehension to generate lists. Some are quite complex, yet useful.

Here is an simple example:

```
>>> dna_list = ['TAGC', 'ACGTATGC', 'ATG', 'ACGGCTAG']
>>> lengths = [len(dna) for dna in dna_list]
>>> lengths
[4, 8, 3, 8]
```

This is how you could do the same with a for loop:

```
>>> lengths = []
>>> dna_list = ['TAGC', 'ACGTATGC', 'ATG', 'ACGGCTAG']
>>> for dna in dna_list:
...   lengths.append(len(dna))
...
>>> lengths
[4, 8, 3, 8]
```

Using conditions:

This will only return the length of an element that starts with 'A':

```
>>> dna_list = ['TAGC', 'ACGTATGC', 'ATG', 'ACGGCTAG']
>>> lengths = [len(dna) for dna in dna_list if dna.startswith('A')]
>>> lengths
[8, 3, 8]
```

> *This generates the following list: [8, 3, 8]*

Here is an example of using mathematical operators to generate a list:

```
>>> two_power_list = [2 ** x for x in range(10)]
>>> two_power_list
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

> *This creates a list of the of the product of [2^0 , 2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7, 2^8, 2^9]*

Dictionaries

Dictionaries are another iterable, like a string and list. Unlike strings and lists, dictionaries are not a sequence, or in other words, they are unordered and the position is not important.

Dictionaries are a collection of key/value pairs. In Python, each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: `{}`

Each key in a dictionary is unique, while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Data that is appropriate for dictionaries are two pieces of information that naturally go together, like gene name and sequence.

KEY	VALUE
TP53	GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC
BRCA1	GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA

Creating a Dictionary

```
genes = { 'TP53' : 'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC' , 'BRCA1' : 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA' }
```

Breaking up the key/value pairs over multiple lines make them easier to read.

```
genes = {
        'TP53' : 'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC' ,
        'BRCA1' : 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
        }
```

Accessing Values in Dictionaries

To retrieve a single value in a dictionary use the value's key in this format `dict[key]`. This will return the value at the specified key.

```
>>> genes = { 'TP53' : 'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC' , 'BRCA1' : 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA' }
>>>
>>> genes['TP53']
GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC
```

> *The sequence of the gene TP53 is stored as a value of the key 'TP53'. We can assess the sequence by using the key in this format dict[key]*

The value can be accessed and passed directly to a function or stored in a variable.

```
>>> print(genes['TP53'])
GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC
>>>
>>> seq = genes['TP53'];
>>> print(seq)
GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC
```

1. Create a dictionary of your favorite color, book, song, and organism. Use the these as the keys: color, book, song, organism.
2. Print out your favorite book.

```
print(fav_dict['book'])
```

3. Print out your favorite book but use a variable in the key.

```
fav_thing = 'book'
print(fav_dict[fav_thing])
```

4. Print out your favorite organism using the literal 'organism' as the key and then with using the variable fav_thing.

```
fav_thing = 'organism'
```

Changing Values in a Dictionary

Individual values can be changed by using the key and the assignment operator.

```
>>> genes = { 'TP53' : 'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC' , 'BRCA1' : 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA' }
>>>
>>> print(genes)
{'BRCA1': 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA', 'TP53': 'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC'}
>>>
>>> genes['TP53'] = 'atg'
>>>
>>> print(genes)
{'BRCA1': 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA', 'TP53': 'atg'}
```

> *The contents of the dictionary have changed.*

Other assignment operators can also be used to change a value of a dictionary key.

```
>>> genes = { 'TP53' : 'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC' , 'BRCA1' : 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA' }
>>>
>>> genes['TP53'] += 'TAGAGCCACCGTCCAGGGAGCAGGTAGCTGCTGGGCTCCGGGGACACTTTGCGTTCGGGCTGGGAGCGTG'
>>>
>>> print(genes)
{'BRCA1': 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA', 'TP53': 'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTCTAGAGCCACCGTCCAGGGAGCAGGTAGCTGCTGGGCTCCGGGGACACTTTGCGTTCGGGCTGGGAGCGTG'}
```

> *Here we have used the '+=' concatenation assignment operator. This is equivalent to genes['TP53'] = genes['TP53'] + 'TAGAGCCACCGTCCAGGGAGCAGGTAGCTGCTGGGCTCCGGGGACACTTTGCGTTCGGGCTGGGAGCGTG'.*

1. Change the value of your favorite organism.

Accessing Each Dictionary Key/Value

Since a dictionary is a sequence we can iterate through its contents.

A for loop can be used to retrieve each key of a dictionary one a time:

```
>>> for gene in genes:
...   print(gene)
...
TP53
BRCA1
```

Once you have the key you can retrieve the value:

```
>>> for gene in genes:
...   seq = genes[gene]
...   print(gene, seq[0:10])
...
TP53 GATGGGATTG
BRCA1 GTACCTTGAT
```

1. Use your text editor to create a script. Create a dictionary of genes (can be fake genes with fake sequences).

2. Use a `for` loop to print each gene name.
3. Add the sequence to your print statement. (name\tsequence)
4. Replace the sequence with the length in your print statement (name\tlength)
5. Add the number of As to your print statement (name\tlength\tACount)
6. Calculate and add the number of Ts to your print statment (name\tlength\tACount\tTCount)
7. Calculate and add the number of Gs to your print statment (name\tlength\tACount\tTCount\tGCount)
8. Calculate and add the number of Cs to your print statment (name\tlength\tACount\tTCount\tGCount\tCCount)
9. Calculate and add the percent GC to your print statement (name\tlength\tACount\tTCount\tGCount\tCCount\tGC%)

Building a Dictionary one Key/Value at a Time

Building a dictionary one key/value at a time is akin to what we just saw when we change a key's value.Normally you won't do this. We'll talk about ways to build a dictionary from a file in a later lecture.

```
1  >>> genes = {}
2  >>> print(genes)
3  {}
4  >>> genes['Brca1'] =
   'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTT
   GGTTTCCGTGGCAACGGAAAA'
5  >>> genes['TP53'] =
   'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAA
   AGTTTTGAGCTTCTCAAAAGTC'
6  >>> print(genes)
7  {'Brca1':
   'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTT
   GGTTTCCGTGGCAACGGAAAA', 'TP53':
   'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAA
   AGTTTTGAGCTTCTCAAAAGTC'}
```

> *We start by creating an empty dictionary. Then we add each key/value pair using the same syntax as when we change a value. dict[key] = new_value*

Checking That Dictionary Keys Exist

Python generates an error (NameError) if you try to access a key that does not exist.

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

Dictionary Operators

OPERATOR	DESCRIPTION
`in`	`key in dict` returns True if the key exists in the dictionary
`not in`	`key not in dict` returns True if the key does not exist in the dictionary

Because Python generates a NameError if you try to use a key that doesn't exist in the dictionary, you need to check whether a key exists before trying to use it.

The best way to check whether a key exists is to use `in`

```
>>> gene = 'TP53'
>>> if gene in genes:
...     print('found')
...
found
>>>
>>> if gene in genes:
...     print(genes[gene])
...
GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC
>>>
```

1. In a text editor create a script that loads these sequences into a dictionary, with the gene name as the key and the sequence as the value.

```
>WHTH4091 Wheat histone H4 TH091 gene
AGCACCCTCCCACCTCATCCCACCCTTCTGATCTCAATCC
AACGTCGCATCTCCACCGTCTCGCGGATCG
ACCCAGCGAAGTCCCTCCCGCCCCCAAAGTCCCCCAAATC
TTGCAGTTCCCTCCTAAATCCTCCCCATAT
AAACCAACCCCCCGCCCTCAGATCCCTAATCCCATCGCAA
GCATCAGACTCCCTCCAAAGCAGGCAGCAG
CTCCTCTTCTTCCTAATCACACTATCTCGGAGAGGAGCGG
CCATGTCTGGGCGCGACAAGGGCGGCAAGG
GGCTGGGCAAGGGCGGCGCCAAGCGGCACCGGAAGGTCC
TCCGCGACAACATCCAGGGCATCACCAAGCC

>X83548.1 H.sapiens gene for histone H4
TCTAGAGATGGCGCCATTTGATTCCAGCAGCCACAAAGCA
CTAGAACAATCGATGCTAAGAGGTGACAGG
AAAAACAGGCTGCAAAGACCCAGACAATGGAATGCAGCG
GTGGTCAGCCTAAAACACTGTAGAAGGGCAA
GATGAGCTGAGTAATTTTTAACTGGGCATCATTTTTAGAA
ACTGGAGTTTAAGTACCCCCTTTTCCATTT

```

2. Check to see if WHTH4091, Brca1, and H4 are in your dictionary. Use the `in` operator.

Sorting Dictionary Keys

If you want to print the contents of a dictionary, you probably want to sort the keys then iterate over the keys with a for loop. Why do you want to sort the keys?

```
for gene_key in sorted(genes):
  print(gene_key, '=>' , genes[gene_key])
```

> *This will print the same order of keys every time you run your script.*

1. Sort your dictionary from the last exercise.
2. Now sort by the sequences (values). Need help? Google it.

Dictionary Functions

FUNCTION	DESCRIPTION
`len(dict)`	returns the total number of key/value pairs
`str(dict)`	returns a string representation of the dictionary
`type(variable)`	Returns the type or class of the variable passed to the function. If the variable is dictionary, then it would return a dictionary type.

These functions work on several other data types too!

Dictionary Methods

METHOD	DESCRIPTION
`dict.clear()`	Removes all elements of dictionary dict
`dict.copy()`	Returns a shallow copy of dictionary dict. Shallow vs. deep copying only matters in multidimensional data structures.
`dict.fromkeys(seq,value)`	Create a new dictionary with keys from seq (Python sequence type) and values set to value.
`dict.items()`	Returns a list of (key, value) tuple pairs
`dict.keys()`	Returns list of keys
`dict.get(key, default =`	get value from dict[key], use default if not not

None)	present
dict.setdefault(key, default = None)	Similar to get(), but will set dict[key] = default if key is not already in dict
dict.update(dict2)	Adds dictionary dict2's key-values pairs to dict
dict.values()	Returns list of dictionary dict's values

1. Return a list of gene names from your dictionary.
2. Join your list on ', ' and print. Need Help? Google it. Read about `join() on TutorialsPoint`. Remember to think about where to place your string and your sequence (list).

Sets

A set is another Python data type. It is essentially a dictionary with keys but no values.

- A set is unordered
- A set is a collection of data with no duplicate elements.
- Common uses include looking for differences and eliminating duplicates in data sets.

Curly braces `{}` or the `set()` function can be used to create sets.

> *Note: to create an empty set you have to use `set()`, not `{}` the latter creates an empty dictionary.*

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)
{'orange', 'banana', 'pear', 'apple'}
```

> *Look, duplicates have been removed*

Test to see if an value is in the set

```
>>> 'orange' in basket
True
>>> 'crabgrass' in basket
False
```

> *The in operator works the same with sets as it does with lists and dictionaries*

Union, intersection, difference and symmetric difference can be done with sets

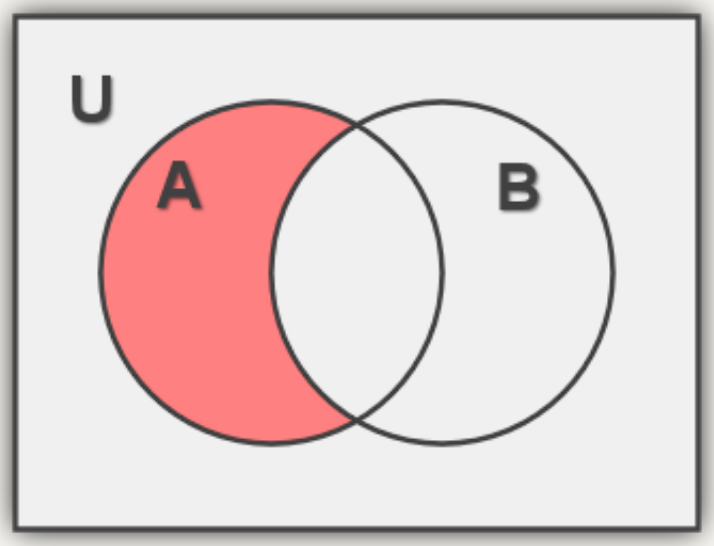
```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
{'a', 'r', 'b', 'c', 'd'}
```

> *Sets contain unique elements, therefore, even if duplicate elements are provided they will be removed.*

Set Operators

Difference

The difference between two sets are the elements that are unique to the set to the left of the `-` operator, with duplicates removed.

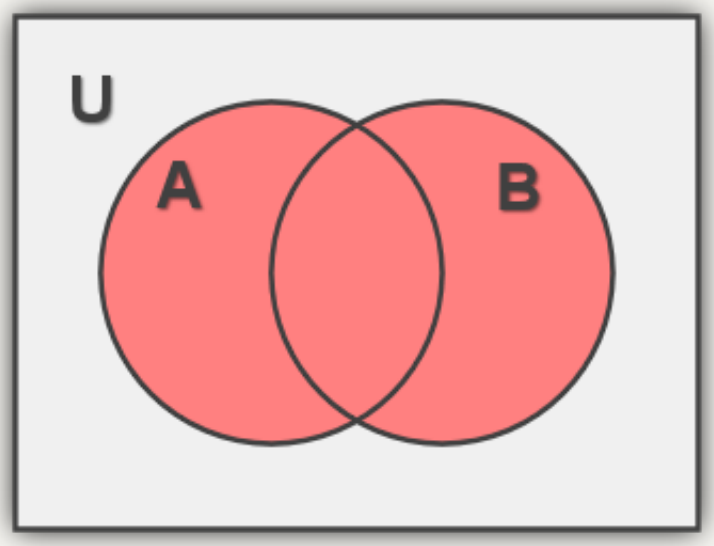


```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a - b
{'r', 'd', 'b'}
```

> *This results the letters that are in a but not in b*

Union

The union between two sets is a sequence of the all the elements of the first and second sets combined, with duplicates removed.

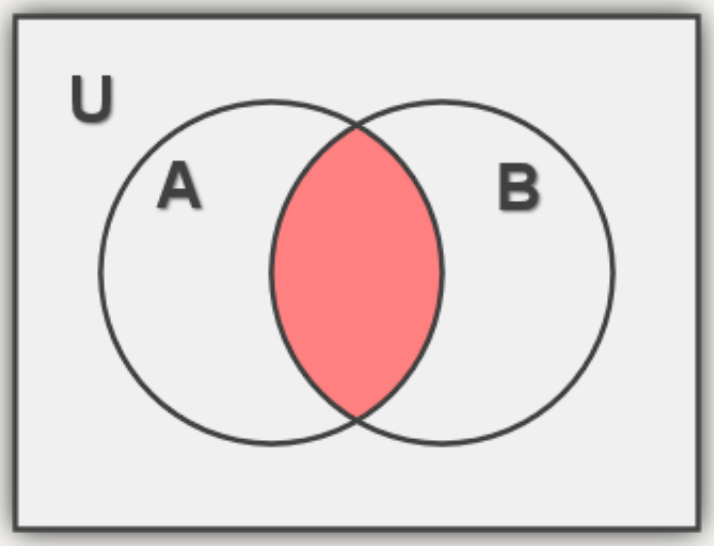



```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a | b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
```

> *This returns letters that are in a or b both*

Intersection

The intersection between two sets is a sequence of the elements which are in both sets, with duplicates removed.

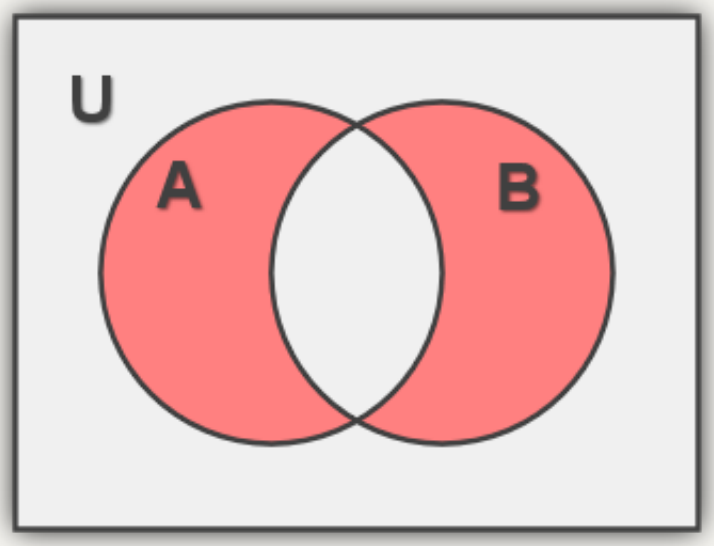



```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a & b
{'a', 'c'}
```

> *This returns letters that are in both a and b*

Symmetric Difference

The symmetric difference is the elements that are only in the first set plus the elements that are only in the second set, with duplicates removed.



```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a ^ b
{'r', 'd', 'b', 'm', 'z', 'l'}
```

> *This returns the letters that are in a or b but not in both (also known as exclusive or)*

Set Functions

FUNCTION	DESCRIPTION
all()	returns True if all elements of the set are true (or if the set is empty).
any()	returns True if any element of the set is true. If the set is empty, return False.
enumerate()	returns an enumerate object. It contains the index and value of all the items of set as a pair.
len()	returns the number of items in the set.
max()	returns the largest item in the set.
min()	returns the smallest item in the set.
sorted()	returns a new sorted list from elements in the set (does not alter the original set).
sum()	returns the sum of all elements in the set.

Set Methods

METHOD	DESCRIPTION
set.add(new)	adds a new element
set.clear()	remove all elements
set.copy()	returns a shallow copy of a set
set.difference(set2)	returns the difference of set and set2
set.difference_update(set2)	removes all elements of another set from this set
set.discard(element)	removes an element from set if it is found in set. (Do nothing if the element is not in set)

	element is not in set)
`set.intersection(sets)`	return the intersection of set and the other provided sets
`set.intersection_update(sets)`	updates set with the intersection of set and the other provided sets
`set.isdisjoint(set2)`	returns True if set and set2 have no intersection
`set.issubset(set2)`	returns True if set2 contains set
`set.issuperset(set2)`	returns True if set contains set2
`set.pop()`	removes and returns an arbitrary element of set.
`set.remove(element)`	removes element from a set.
`set.symmetric_difference(set2)`	returns the symmetric difference of set and set2
`set.symmetric_difference_update(set2)`	updates set with the symmetric difference of set and set2
`set.union(sets)`	returns the union of set and the other provided sets
`set.update(set2)`	update set with the union of set and set2

I/O and Files

I/O stands for input/output. The in and out refer to getting data into and out of your script. It might be a little surprising at first, but writing to the screen, reading from the keyboard, reading from a file, and writing to a file are all examples of I/O.

Writing to the Screen

You should be well versed in writing to the screen. We have been using the `print()` function to do this.

```
>>> print ("Hello, EVOP2018!")
Hello, EVOP2018!
```

> *Remember this example from one of our first lessons?*

Reading input from the keyboard

This is something new. There is a function which prints a message to the screen and waits for input from the keyboard. This input can be stored in a variable. It always starts as a string. Convert to an int or float if you want a number.

```
>>> user_input = input("Type Something Now: ")
Type Something Now: Hi
>>> print(user_input)
Hi
>>> type(user_input)
<class 'str'>
```

1. In a text editor, create a script that asks the user for their name.
2. Uppercase all the letters and print out the name. Need help? Google it

Reading from a File

Most of the data we will be dealing with will be contained in files.

The first thing to do with a file is open it. We can do this with the `open()` function. The `open()` function takes the file name and access mode as arguments and returns a file object.

The most common access modes are read (r) and write (w).

Open a File

```
>>> file_object = open("seq.nt","r")
```

> *'file_object' is a name of a variable. This can be anything, but make it a helpful name that describes what kind of file you are opening.*

Reading the contents of a file

Now that we have opened a file and created a file object we can do things with it, like read it. Lets read all the contents at once.

Let's go to the command line and `cat` the contents of the file to see what's in it first

```
1  $ cat seq.nt
2  ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTT
   GGTAGCAAACACTTCCAAAAG
3  ACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAAT
   GTTTATTGTTGTAGCTCTGG
4  $
```

Note the new lines. Now, lets print the contents to the screen with Python. We will use `read()` to read the entire contents of the file into a variable.

```
1  >>> file = open("seq.nt","r")
2  >>> contents = file.read()
3  seq.nt.fa
4  >>> print(contents)  # note newline characters are part of the file!
5  ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTT
   GGTAGCAAACACTTCCAAAAG
6  ACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAAT
   GTTTATTGTTGTAGCTCTGG
7
8  >>> file.close()
```

> *The complete contents can be retrieved with the `read()` method. Notice the newlines are maintained when `contents` is printed to the screen. `print()` adds another new line when it is finished printing. It is good practice to close your file. Use the `close()` method.*

Here's another way to read data in from a file. A `for` loop can be used to iterate through the file one line at a time.

```
##!/usr/bin/env python3

file = open("seq.nt","r")
for line in file: # Python magic: reads in a line from file
  print(line)
```

Output:

```
$ python3 file_for.py
ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTT
GGTAGCAAACACTTCCAAAAG

ACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAAT
GTTTATTGTTGTAGCTCTGG

```

> *Notice the blank line at after each line we print.* `print()` *adds a newline and we have a newline at the end of each line in our file. Use* `rstrip()` *method to remove the newline from each line.*

Let's use `rstrip()` method to remove the newline from our file input.

```
$ cat file_for_rstrip.py
##!/usr/bin/env python3

file_object = open("seq.nt","r")
for line in file_object:
  line = line.rstrip()
  print(line)
```

> `rstrip()` *without any parameters returns a string with whitespace removed from the end.*

Output:

```
$ python3 file_for_rstrip.py
ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTT
GGTAGCAAACACTTCCAAAAG
ACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAAT
GTTTATTGTTGTAGCTCTGG
```

> *Where do the newlines in the above output come from?*

Opening a file with `with open() as fh:`

Many people add this because it closes the file for you automatically. Good programming practice. Your code will clean up as it runs. For more advanced coding, `with ... as ...` saves limited resources like filehandles and database connections. For now, we just need to know that the `with ... as ...:` does the same as `fh = open(...) ... fh.close()`. So here's what the adapted code looks like

```
1  ##!/usr/bin/env python3
2
3  with open("seq.nt","r") as file_object: #cleans up after exiting with
   block
4    for line in file_object:
5      line = line.rstrip()
6      print(line)
7  ##file gets closed for you here.
```

Writing to a File

Writing to a file is nothing more than opening a file for writing then using the `write()` method.

The `write()` method is like the `print()` function. The biggest difference is that it writes to your file object instead of the screen. Unlike `print()` it does not add a newline by default. `write()` takes a single string argument.

Let's write a few lines to a file named "writing.txt".

```
1  ##!/usr/bin/env python3
2
3  fo = open("writing.txt" , "w")
4  fo.write("One line.\n")
5  fo.write("2nd line.\n")
6  fo.write("3rd line" + " has extra text\n")
7  some_var = 5
8  fo.write("4th line has " + str(some_var) + " words\n")
9  fo.close()
10 print("Wrote to file 'writing.txt'") # it's nice to tell the user you wrote
   a file
```

Output:

```
$ python3 file_write.py
Wrote to file 'writing.txt'
$ cat writing.txt
One line.
2nd line.
3rd line has extra text
4th line has 5 words
```

Now, let's get crazy! Lets read from one file a line at a time. Do something to each line and write the results to a new file.

```
##!/usr/bin/env python3

seq_read  = open("seq.nt","r")
seq_write = open("nt.counts.txt","w")

total_nts = 0
for line in seq_read:
  line = line.rstrip()
  nt_count = len(line)
  total_nts += nt_count
  seq_write.write(str(nt_count) + "\n")

seq_write.write("Total: " + str(total_nts) +"\n")

seq_read.close()
seq_write.close()
print("Wrote 'nt.counts.txt'")
```

Output:

```
$ python file_read_write.py
$ cat nt.counts.txt
71
71
Total: 142
```

> *The file we are reading from is named, "seq.nt.fa"*
> *The file we are writing to is named, "nt.counts.txt"*
> *We read each line, calculate the length of each line and print the length*
> *We also create a variable to keep track of the total nt count*
> *At the end, we print out the total count of nts*
> *Finally we close each of the files*

Building a Dictionary from a File

This is a very common task. It will use a loop, file I/O and a dictionary.

Assume we have a file called "sequence_data.txt" that contains tab-delimited gene names and sequences that looks something like this

```
1  TP53
   GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAA
   AGTTTTGAGCTTCTCAAAAGTC
2  BRCA1
   GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTT
   GGTTTCCGTGGCAACGGAAAA
```

How can we read this whole file in to a dictionary?

```
1  ##!/usr/bin/env python3


2
3  seq_read  = open("sequence_data.txt","r")
4  genes = {}
5  for line in seq_read:
6      line = line.rstrip()
7      id,seq = line.split() #split on whitespace

8      genes[id] = seq
9  seq_read.close()
10 print(genes)
```

Output:

```
{'TP53':
'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAA
AGTTTTGAGCTTCTCAAAAGTC', 'BRCA1':
'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTT
GGTTTCCGTGGCAACGGAAAA'}
```

1. Let's do it! Use sequence_data.txt to read in the gene information and create a dictionary.

Regular Expressions

Regular Expressions is a language for pattern matching. Many different computer languages incorporate regular expressions as well as some unix commands like grep and sed. So far we have seen a few functions for finding exact matches in strings, but this is not always sufficient.

Functions that utilize regular expressions allow for non-exact pattern matching.

These specialized functions are not included in the core of Python. We need to import them by typing `import re` at the top of your script

```
#!/usr/bin/env python3

import re
```

First we will go over a few examples then go into the mechanics in more detail.

Let's start simple and find an exact match for the EcoRI restriction site in a string.

```
>>> dna =
'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTT
GGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACAGTCT
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
CCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAAT
GTTTATTGTTGTAGCTCTGG'
>>> if re.search(r"GAATTC",dna):
...   print("Found an EcoRI site!")
...
Found an EcoRI site!
>>>
```

> *Since we can search for control characters like a tab (\t) it is good to get in the habit of using the raw string function `r` when defining patterns.*

> *Here we used the `search()` function with two arguments, 1) our pattern and 2) the string we want to search.*

Let's find out what is returned by the `search()` function.

```
>>> dna =
'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTT
GGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACAGTCT
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
CCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAAT
GTTTATTGTTGTAGCTCTGG'
>>> found=re.search(r"GAATTC",dna)
>>> print(found)
<_sre.SRE_Match object; span=(70, 76), match='GAATTC'>
```

> *Information about the first match is returned*

How about a non-exact match. Let's search for a methylation site that has to match the following criteria:

- G or A
- followed by C
- followed by one of anything or nothing
- followed by a G

This could match any of these: GCAG GCTG GCGG GCCG GCG ACAG ACTG ACGG ACCG ACG

We could test for each of these, or use regular expressions. This is exactly what regular expressions can do for us.

```
>>> dna = 'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG'
>>> found=re.search(r"[GA]C.?G",dna)
>>> print(found)
<_sre.SRE_Match object; span=(7, 10), match='ACG'>
```

> *Here you can see in the returned information that ACG starts at string postion 7 (nt 8).*
>
> *The first position following the end of the match is at string postion 10 (nt 11).*

What about other potential matches in our DNA string? We can use `findall()` function to find all matches.

```
>>> dna = 'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG'
>>> found=re.findall(r"[GA]C.?G",dna)
>>> print(found)
['ACG', 'GCTG', 'ACTG', 'ACCG', 'ACAG', 'ACCG', 'ACAG']
```

> *`findall()` returns a list of all the pieces of the string that match the regex.*

A quick count of all the matching sites can be done by counting the length of the returned list.

```
>>> len (re.findall(r"[GA]C.?G",dna))
7
```

> *There are 7 methylation sites.*

> *Here we have another example of nesting.*
>
> *We call the `findall()` function, searching for all the matches of a methylation site.*
>
> *This function returns a list, the list is past to the `len()` function, which in turn returns the number of elements in the list.*

1. If you want to find just the first occurrence of a pattern, what method do you use?
2. If you want to find all the occurrences of a pattern, what method do you use?
3. What operator have we seen that will report if an exact match is in a sequence (string, list, dictionary, etc)?
4. What string method have we seen that will count the number of occurrences of an exact match in a string?

Let's talk a bit more about all the new characters we see in the pattern.

The pattern in made up of atoms. Each atom represents **ONE** character.

Individual Characters

ATOM	DESCRIPTION
a-z, A-Z, 0-9 and some punctuation	These are ordinary characters that match themselves
"."	The dot, or period. This matches any single character except for the newline.

Character Classes

A group of characters that are allowed to be matched one time. There are a few predefined classes, symbol that means a series of characters.

ATOM	DESCRIPTION
`[ ]`	A bracketed list of characters, like `[GA]`. This indicates a single character can match any character in the bracketed list.
`\d`	Digits. Also can be written `[0-9]`
`\D`	Not digits. Also can be written `[^0-9]`
`\w`	Word character. Also can be written `[A-Za-z0-9_]` Note underscore is part of this class
`\W`	Not a word character, or `[^A-Za-z0-9_]`
`\s`	White space character. Also can be written `[\r\t\n]`. Note the space character after the first `[`
`\S`	Not whitespace. Also `[^ \r\\t\n]`

Anchors

A pattern can be anchored to a region in the string:

ATOM	DESCRIPTION
`^`	Matches the beginning of the string
`$`	Matches the end of the string
`\b`	Matches a word boundary between `\w` and `\W`

Examples:

```
1  g..t
```

> *matches "gaat", "goat", and "gotta get a goat" (twice)*

```
1  g[gatc][gatc]t
```

> *matches "gaat", "gttt", "gatt", and "gotta get an agatt" (once)*

```
\d\d\d-\d\d\d\d`
```

> *matches 376-8380, and 5128-8181but not 055-98-2818.*

```
^\d\d\d-\d\d\d\d
```

> *matches 376-8380 and 376-83801but not 5128-8181.*

```
^\d\d\d-\d\d\d\d$
```

> *only matches telephone numbers (without area code)*

```
\bcat
```

> *matches "cat", "catsup" and "more catsup please" but not "scat".*

```
\bcat\b
```

> *only text containing the word "cat".*

Quantifiers

Quantifiers quantify how many atoms are to be found. By default an atom matches only once. This behaviour can be modified following an atom with a quantifier.

QUANTIFIER	DESCRIPTION
`?`	atom matches zero or exactly once
`*`	atom matches zero or more times
`+`	atom matches one or more times
`{3}`	atom matches exactly 3 times
`{2,4}`	atom matches between 2 and 4 times, inclusive
`{4,}`	atom matches at least 4 times

Examples:

```
goa?t
```

> *matches "goat" and "got". Also any text that contains these words.*

```
g.+t
```

> *matches "goat", "goot", and "grant", among others.*

```
g.*t
```

> *matches "gt", "goat", "goot", and "grant", among others.*

```
^\d{3}-\d{4}$
```

> *matches US telephone numbers (no extra text allowed).*

1. What would be a pattern to recognize an email address?
2. What would be a pattern to recognize the ID portion of a sequence record in a FASTA file?

Variables and Patterns

Variables can be used to store patterns.

```
>>> pattern = r"[GA]C.?G"
>>> len (re.findall(pattern,dna))
7
```

> *In this example we stored our methylation pattern in the variable named 'pattern' and used it as the first argument to* `findall` *.*

Either Or

A pipe '|' can be used to indicated that either the pattern before or after the '|' can match. Enclose the two options in parenthesis.

```
1  big bad (wolf|sheep)
```

> *This pattern must match a string that contains:*
>
> - *"big" followed by a space followed by*
> - *"bad" followed by*
> - *a space followed by*
> - *either "wolf" or "sheep"*
>
> *This would match:*
>
> - *"big bad wolf"*
> - *"big bad sheep"*

1. What would a pattern to match 'ATG' followed by a C or a T look like?

Subpatterns

Subpatterns, or parts of the pattern enclosed in parenthesis can be extracted and stored for later use.

```
1  Who's afraid of the big bad w(.+)f
```

> *This pattern has only one subpattern (.+)*

You can combine parenthesis and quantifiers to quantify entire subpatterns.

```
1  Who's afraid of the big (bad )?wolf\?
```

> *This matches:*
>
> - *"Who's afraid of the big bad wolf?"*
> - *.As well as "Who's afraid of the big wolf?".*

> *The 'bad ' is optional, it can be present 0 or 1 times in our string.*
>
> *This also shows how to literally match special characters. Use a '\' in to escape them.*

1. What pattern could you use to capture the ID in a sequence record of a FASTA file in a subpattern.

Example FASTA sequence record.

```
>ID Optional Descrption
SEQUENCE
SEQUENCE
SEQUENCE
```

Using Subpatterns Inside the Regular Expression Match

This is helpful when you want to find a subpattern and then match the contents again. They can be used within the function call and used after the function call.

Subpatterns within the function call

Once a subpattern matches, you can refer to it within the same regular expression. The first subpattern becomes \1, the second \2, the third \3, and so on.

```
Who's afraid of the big bad w(.)\1f
```

> *This would match:*
>
> - *"Who's afraid of the big bad woof"*
> - *"Who's afraid of the big bad weef"*
> - *"Who's afraid of the big bad waaf"*
>
> *But Not:*
>
> - *"Who's afraid of the big bad wolf"*
> - *"Who's afraid of the big bad wife"*

In a similar vein,

```
\b(\w+)s love \1 food\b
```

> *This pattern will match*
>
> - *"dogs love dog food"*
> - *But not "dogs love monkey food".*
>
> *We were able to use the subpattern within the regular expression by using* `\1`
>
> *If there were more subpatterns they would be* `\2` , `\3` , `\4` , *etc*

Using Subpatterns Outside the Regular Expression

Subpatterns can be retrieved after the `search()` function call, or outside the regular expression, by using the `group()` method. This is a method and it belongs to the object that is returned by the `search()` function.

The subpatterns are retrieved by a number. This will be the same number that could be used within the regular expression, i.e.,

- `\1` within the subpattern can be used outside with `search_found_obj.group(1)`
- `\2` within the subpattern can be used outside with `search_found_obj.group(2)`
- `\3` within the subpattern can be used outside with `search_found_obj.group(3)`
- and so on

Example:

```
>>> dna = 'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG'
>>> found=re.search( r"(.{50})TATTAT(.{25})" , dna )
>>> upstream = found.group(1))
>>> print(upstream)
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
>>> downstream = found.group(2))
>> print(downstream)
CCGGTTTCCAAAGACAGTCTTCTAA
```

> 1. *This pattern will recognize a consensus transcription start site (TATTAT)*
> 2. *And store the 50 base pairs upstream of the site*
> 3. *And the 25 base pairs downstream of the site*

If you want to find the upstream and downstream sequence of ALL 'TATTAT' sites, use the `findall()` function.

```
>>> dna="ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG"
>>> found = re.findall( r"(.{50})TATTAT(.{25})" , dna )
>>> print(found)
[('TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA', 'CCGGTTTCCAAAGACAGTCTTCTAA'), ('TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA', 'CCGGTTTCCAAAGACAGTCTTCTAA')]
```

> *The subpatterns are stored in tuples within a list. More about this type of data structure later.*

Another option for retrieving the upstream and downstream subpatterns is to put the `findall()` in a for loop

```
1  >>>
dna="ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAA
ACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACA
GTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTC
TGGATATTATCCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAAT
AAGTAAAATGTTTATTGTTGTAGCTCTGGACAAAATACGTTTTGTAAAT
GTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAA
GGAATTCACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATA
AGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGAC
AGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCT
CTGG"
2  >>> for (upstream, downstream) in re.findall( r"(.{50})TATTAT(.{25})"  ,
dna ):
3  ...  print("upstream:" , upstream)
4  ...  print("downstream:" , downstream)
5  ...
6  upstream:
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
7  downstream: CCGGTTTCCAAAGACAGTCTTCTAA
8  upstream:
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
9  downstream: CCGGTTTCCAAAGACAGTCTTCTAA
```

> 1. *This code executes the `findall()` function once*
> 2. *The subpatterns are returned*
> 3. *The subpatterns are stored in the variables upstream and downstream*
> 4. *The for block of code is executed*
> 5. *The `findall()` searches again*
> 6. *A match is found*
> 7. *New subpatterns are returned and stored in the variables upstream and downstream*
> 8. *The for block of code gets executed again*
> 9. *The `findall()` searches again, but no match is found*
> 10. *The for loop ends*

Another way to get this done is with an iterator, use the `finditer()` function in a for loop. This allows you to not store all the matches in memory. `finditer()` also allows you to retrieve the postion of the match.

```
1  >>>
dna="ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAA
ACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACA
GTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTC
TGGATATTATCCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAAT
AAGTAAAATGTTTATTGTTGTAGCTCTGGACAAAATACGTTTTGTAAAT
GTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAA
GGAATTCACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATA
AGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGAC
AGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCT
CTGG"
2  >>> for match in re.finditer(r"(.{50})TATTAT(.{25})"  , dna):
3  ...  print("upstream:" , match.group(1))
4  ...  print("downstream:" , match.group(2))
5  ...
6  upstream:
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
7  downstream: CCGGTTTCCAAAGACAGTCTTCTAA
8  upstream:
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
9  downstream: CCGGTTTCCAAAGACAGTCTTCTAA
```

1. *This code executes `finditer()` function once.*
2. *The match object is returned. A match object will have all the information about the match.*
3. *In the for block we call the `group()` method on the first match object returned*
4. *We print out the first and second subpattern using the `group()` method*
5. *The `finditer()` function is executed a second time and a match is found*
6. *The second match object is returned*
7. *The second subpatterns are retrieved from the match object using the `group()` method*
8. *The `finditer()` function is executed again, but no matches found, so the loop ends*

Get position of the subpattern with `finditer()`

The match object contains information about the match that can be retrieved with match methods like `start()` and `end()`

```
#!/usr/bin/env python3

import re

dna="ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAAT
AAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAG
ACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTA
GCTCTGGATATTATCCGGTTTCCAAAGACAGTCTTCTAATTCCTCATT
AGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACAAAATACGTTT
TGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACAC
TTCCAAAAGGAATTCACCGGTTTCCAAAGACAGTCTTCTAATTCCTCA
TTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGT
TTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTA
TTGTTGTAGCTCTGG"

for found in re.finditer(r"(.{50})TATTAT(.{25})"  , dna):
  whole    = found.group(0)
  up       = found.group(1)
  down     = found.group(2)
  up_start = found.start(1) + 1   # need to convert from 0 to 1
notation
  up_end   = found.end(1)   + 1
  dn_start = found.start(2) + 1
  dn_end   = found.end(2)   + 1

  output = [ whole , up , str(up_start), str(up_end) , down ,
str(dn_start) , str(dn_end)  ]

  print( "\t".join(output) )
```

> *we can use these match object methods `group()` , `start()` , `end()` to get the string, start position, and end position of each subpattern.*

```
$ python3 re.finditer.pos.py

TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
TATTATCCGGTTTCCAAAGACAGTCTTCTAA
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
 98  148 CCGGTTTCCAAAGACAGTCTTCTAA  154 179

TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
TATTATCCGGTTTCCAAAGACAGTCTTCTAA
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
 320 370 CCGGTTTCCAAAGACAGTCTTCTAA  376 401
```

FYI: `match()` function is another regular expression function that looks for patterns. It is similar to search but it only looks at the beginning of the string for the pattern while `search()` looks in the entire string. Usually `finditer()` , `search()` and `findall()` will be more useful.

Subpatterns and Greediness

By default, regular expressions are "greedy". They try to match as much as they can. Use the quantifier '?' to make the match not greedy. The not greedy match is called 'lazy'

```
>>> str = 'The fox ate my box of doughnuts'
>>> found = re.search(r"(f.+x)",str)
>>> print(found.group(1))
fox ate my box
```

> *The pattern f.+x does not match what you might expect, it matches past 'fox' all the way out to 'fox ate my box'. The '.+' id greedy As many characters as possible are found that are between the 'f' and the 'x'.*

Let's make this match lazy by using '?'

```
>>> found = re.search(r"(f.+?x)",str)
>>> print(found.group(1))
fox
```

> *The match is now lazy and will only match 'fox'*

Practical Example: Codons

Extracting codons from a string of DNA can be accomplished by using a subpattern in a `findall()` function. Remember the `findall()` function will return a list of the matches.

```
>>> dna = 'GTTGCCTGAAATGGCGGAACCTTGAA'
>>> codons = re.findall(r"(.{3})",dna)
>>> print(codons)
['GTT', 'GCC', 'TGA', 'AAT', 'GGC', 'GGA', 'ACC', 'TTG']
```

Or you can use a for loop to do something to each match.

```
>>> for codon in re.findall(r"(.{3})",dna):
...  print(codon)
...
GTT
GCC
TGA
AAT
GGC
GGA
ACC
TTG
>>>
```

> *`finditer()` would also work in this for loop. Each codon can be accessed by using the `group()` method.*

1. Create a script in your text editor.
2. Open sequence_data.txt
3. loop through each line of the file
4. store the gene name in a variable
5. store the sequence in a variable
6. create a list of codons
7. count the number of codons in each sequence
8. print "gene name\tcodon count"

Truth and Regular Expression Matches

The `search()` , `match()` , `findall()` , and `finditer()` can be used in conditional tests. If a match is not found an empty list or 'None' is returned. These both are False.

```
>>> found=re.search( r"(.{50})TATTATZ(.{25})"  , dna )
>>> if found:
...    print("found it")
... else:
...    print("not found")
...
not found
>>> print(found)
None
```

> *None is False so the else block is executed and "not found" is printed*

Nest it!

```
>>>
>>> if re.search( r"(.{50})TATTATZ(.{25})"  , dna ):
...    print("found it")
... else:
...    print("not found")
...
not found
>>> print(found)
None
```

Using Regular expressions in substitutions

Earlier we went over how to find an **exact pattern** and replace it using the `replace()` method. To find a pattern, or inexact match, and make a replacement the regular expression `sub()` function is used. This function takes the pattern, the replacement, the string to be searched, the number of times to do the replacement, and flags.

```
>>> str = "Who's afraid of the big bad wolf?"
>>> re.sub(r'w.+f' , 'goat', str)
"Who's afraid of the big bad goat?"
>>> print(str)
Who's afraid of the big bad wolf?
```

> *The `sub()` function returns "Who's afraid of the big bad goat?" The value of variable str has not been altered The new string can be stored in a new variable for later use.*

Let's save the returned new string in a variable

```
>>> str = "He had a wife."
>>> new_str = re.sub(r'w.+f' , 'goat', str)
>>> print(new_str)
He had a goate.
>>> print(str)
He had a wife.
```

> *The characters between 'w' and 'f' have been replaced with 'goat'. The new string is saved in new_str*

Using subpatterns in the replacement

Sometimes you want to find a pattern and use it in the replacement.

```
>>> str = "Who's afraid of the big bad wolf?"
>>> new_str = re.sub(r"(\w+) (\w+) wolf" , r"\2 \1 wolf" , str)
>>> print(new_str)
Who's afraid of the bad big wolf?
```

> *We found two words before 'wolf' and swapped the order.\2 refers to the second subpattern\1 refers to the first subpattern*

1. Create a script in your text editor.

2. Open sequence_data.txt
3. loop through each line of the file
4. store the gene name in a variable
5. store the sequence in a variable
6. Create a regular expressions to find all occurrences of 'ATG' and replace with '-M-' in each sequence
7. print "gene name\t reformatted sequence"

Regular Expression Option Modifiers

MODIFIER	DESCRIPTION
re.I re.IGNORECASE	Performs case-insensitive matching.
re.M re.MULTILINE	Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
re.S re.DOTALL	Makes a period (dot) match any character, including a newline.
re.U	Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
re.X VERBOSE	This flag allows you to write regular expressions that look nicer and are more readable by allowing you to visually separate logical sections of the pattern and add comments. Whitespace within the pattern is ignored, except when in a character class or when preceded by an unescaped backslash. When a line contains a # that is not in a character class and is not preceded by an unescaped backslash, all characters from the leftmost such # through the end of the line are ignored.

```
>>> dna = "atgcgtaatggc"
>>> re.search(r"ATG",dna)
>>>
>>> re.search(r"ATG",dna , re.I)
<_sre.SRE_Match object; span=(0, 3), match='atg'>
>>>
```

> *We can make our search case insensitive by using the `re.I` or `re.IGNORECASE` flag.*

You can use more than one flag by concatenating them with `|`.
`re.search(r"ATG",dna , re.I|re.M)`

Basic FASTA Parser

Now that we have gone over these points we can build our first FASTA parser

1. getting arguments from the command line
2. opening a file and reading every line
3. Building a dictionary one key/value pair at a time
4. regular expressions to match a pattern

Spoiler Alert! Don't click this link yet: Basic FASTA Parser.

1. Try creating your on our own FASTA parser using your text editor.
    1. get a FASTA file name from the command line
    2. create an empty dictionary
    3. open the file
    4. read the file line by line
    5. check to see if the line is the seqid or sequence
    6. add sequence lines to your dictionary using the id as the key
2. **Remember to test your code often!!** The more lines you write without running and testing are more lines to debug.
3. Need help? Check out my basic FASTA parser.

Functions

Functions consist of lines of code that do something useful and that you want to run more than once. You also give that function a name so you can refer to it in your code. This avoids copying and pasting the code to many places in your script and makes your code easier to read.

Let's see some examples.

Python has built-in functions

```
>>> print('Hello world!')
Hello world!
>>> len('AGGCT')
5
```

You can also define your own functions with `def` Let's write a function that calculates the GC content. Let's define this as the fraction of nucleotides in a DNA sequence that are G or C. It can vary from 0 to 1.

First we can look at the code that makes the calculation, then we can convert those lines of code into a function.

Code to find GC content:

```
dna = 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCT'
c_count = dna.count('C')  # count is a string method
g_count = dna.count('G')
dna_len = len(dna) # len is a function
gc_content = (c_count + g_count) / dna_len # fraction from 0 to 1
print(gc_content)
```

Defining a Function that calculates GC Content

We use `def` do define our own function. It is followed by the name of the function (`gc_content`) and parameters it will take in parentheses. A colon is the last character on the `def` line. The parameter variables will be available for your code inside the function to use.

```
def gc_content(dna):  # give our function a name and parameter 'dna'
  c_count = dna.count('C')
  g_count = dna.count('G')
  dna_len = len(dna)
  gc_content = (c_count + g_count) / dna_len
  return gc_content # return the value to the code that called this function
```

> *Here is a custom function that you can use like a built in Python function*

Using your function to calculate GC content

This is just like any other python function. You write the name of the function with any variables you want to pass to the function in parentheses. In the example below the contents of `dna_string` get passed into `gc_content()`. Inside the function this data is passed to the variable `dna`.

```
dna_string = "GTACCTTGATTTCGTATTCTGAGAGGCTGCT"
print(gc_content(dna_string))
```

This code will print 0.45161290322580644 to the screen. You can save this value in a variable to use later in your code like this

```
dna_gc = gc_content('GTACCTTGATTTCGTATTCTGAGAGGCTGCT')
```

As you can see we can write a nice clear line of python to call this function and because the function has a name that describes what it does it's easy to understand how the code works. Don't define your functions like this `def my_function(a):`!

How could you convert the GC fraction to % GC. Use format()

```
dna_string = "GTACCTTGATTTCGTATTCTGAGAGGCTGCT"
dna_gc = gc_content(dna_string)
pc_gc = '{:.2%}'.format(dna_gc)
print('This sequence is' , pc_gc , 'GC')
```

Here's the output

```
This sequence is 45.16% GC
```

The details

1. You define a function with `def`. You need to define a function before you can call it.
2. The function must have a name. This name should clearly describe what the function does. Here is our example `gc_content`
3. You can pass variables to functions but you don't have to. In the definition line, you place variables your function needs inside parentheses like this `(dna)`. This variable only exists inside the function.
4. The first line of the function must end with a `:` so the complete function definition line looks like this `def gc_content(dna):`
5. The next lines of code, the function body, needs to be indented. This code comprises what the function does.
6. You can return a value as the last line of the function, but this is not required. This line `return gc_content` at the end of our function definition passes the value of gc_content back to the code that called the function in your main script.

Naming Arguments

You can name your argument variables anything you want, but they should describe the data they contain. The name needs to be consistent within your function. You could change `dna` to `seqeunce` like this

```
1  def gc_content(sequence):   # give our function a name and parameter
   'sequence'
2    c_count = sequence.count('C')
3    g_count = sequence.count('G')
4    dna_len = len(sequence)
5    gc_content = (c_count + g_count) / dna_len
6    return gc_content # return the value of gc_content to the code that
   called this function
```

Keyword Arguments

Arguments can be named and these names can be used when the function is called. This name is called a 'keyword'

```
>>> dna_string = "GTACCTTGATTTCGTATTCTGAGAGGCTGCT"
>>> print(gc_content(dna_string))
0.45161290322580644
>>> print(gc_content(dna=dna_string)
0.45161290322580644

```

> *The keyword must be the same as the defined function argument. If a function has multiple arguments, using the keyword allows for calling the function with the arguments in any order.*

Default Values for Arguments

As defined above, our function is expecting an argument (`dna`) in the definition. You get an error if you call the function without any parameters.

```
>>> gc_content()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: gc_content() missing 1 required positional argument: 'dna'

```

You can define default values for arguments when you define your function.

```
def gc_content(dna='A'):   # give our function a name and parameter 'dna'
    c_count = dna.count('C')
    g_count = dna.count('G')
    dna_len = len(dna)
    gc_content = (c_count + g_count) / dna_len
    return gc_content # return the value to the code that called this function

```

> *If you call the function with no arguments, the default will be used. In this case a default is pretty useless, and the function will return '0' if called without providing a DNA sequence.*

1. Create a script that contains a gc_content function.
2. Open sequence_data.txt
3. loop through each line of the file
4. store the gene name in a variable
5. store the sequence in a variable
6. calculate the gc content using your function
7. print "gene name\tgc content"

Lambda expressions

Lambda expressions can be used to define simple (one-line) anonymous functions. There are some uses for lambda which we won't go into here. We are showing it to you because sometimes you will come across it.

Here is a one line custom function, like the functions we have already talked about:

```
1  def get_first_codon(dna):
2   return dna[0:3]
3
4  print(get_first_codon('ATGTTT'))
```

> *This will print* `ATG`

Here is the same function written as a lambda

```
1  get_first_codon = lambda dna : dna[0:3]
2  print(get_first_codon('ATGTTT'))
```

> *This also prints* `ATG`*. lambdas can only contain one line and there is no* `return` *statement.*

List comprehensions can often be used instead of lambdas and may be easier to read. You can read more about `lambda`, particularly in relation to `map` which will perform an operation on a list, but generally a `for` loop is easier to read.

Scope

Global Variables

Almost all python variables are global. This means they are available everywhere in your code. The most important exception is variables thare are defined in functions which only exist inside their function. This is called 'local'. Remember that python blocks are defined as code at the same level of indentation.

```
##!/usr/bin/env python3
print('Before if block')
x = 100
print('x=',x)
if True:  # this if condition will always be True
  # we want to make sure the block gets executed
  # so we can show you what happens
  print('Inside if block')
  y = 10
  x = 30
  print("x=", x)
  print("y=", y)

print('After if block')
print("x=", x)
print("y=", y)


```

Let's Run it:

```
$ python3 scripts/scope.py
Before if block
x= 100
Inside if block
x= 30
y= 10
After if block
x= 30
y= 10

```

Inside a function, global variables are visible, but it's better to pass variables to a function as arguments

```
def show_n():
 print(n)
n = 5
show_n()
```

The output is this `5` as you would expect, but this is better programming practice. Why? We'll see a little later.

```
def show_n(n):
 print(n)
n = 5
show_n(n)
```

Local Variables

Variables inside functions are local and therefore can only been accessed from within the function block. This applies to arguments as well as variables defined inside a function.

```
##!/usr/bin/end python3

def set_local_x_to_five(x):
 print('Inside def')
 x = 5 # local to set_local_x_to_five()
 y=5
 print("x =",x)
 print("y = ",y)

print('After def')
x = 100 # global x
y = 100 # global
print('x=',x)
print('y=',y)

set_local_x_to_five(500)
print('After function call')
print('x=',x)
print('y=',y)

```

> *Here we have added a function `set_local_x_to_five` with an argument named 'x'. This variable exists only within the function where is replaces any variable with the same name outside the `def`. Inside the `def` we also initialize a variable `y` that also replaces any global `y` within the `def`*

Let's run it:

```
1  $ python3 scope_w_function.py
2  After def
3  x= 100
4  y= 100
5  Inside def
6  x = 5
7  y =  5
8  After function call
9  x= 100
10 y= 100
11
12
13
```

> *There is a global variable, `x` = 100, but when the function is called, it makes a new local variable, also called `x` with value = 5. This variable disappears after the function finishes and we go back to using the global variable `x` = 100. Same for `y`*

Global

You can make a local variable global with the statement `global`. Now a variable you use in a function is the same variable as in the rest of the code. It is best not to define any variables as global until you know you need to because you might modify the contents of a variable without meaning to.

Here is an example use of `global`.

```
1  ##!/usr/bin/env python3
2
3  def set_global_variable():
4    global greeting  # make greeting global
5    greeting = "I say hello"
```

```
6
7
8  greeting = 'Good morning'
9  print('Before function call')
10 print('greeting =',greeting)
11
12 ##make call to function
13 set_global_variable()
14 print('After function call')
15 print('greeting =',greeting)
16
```

Let's look at the output

```
1  $ python3 scripts/scope_global.py
2  Before function call
3  greeting = Good morning
4  After function call
5  greeting = I say hello
6
```

> *Note that the function has changed the value of the global variable. You might not want to do this.*

Data Structures

Sometimes a *simple* list or dictionary just doesn't do what you want. Sometimes you need to organize data in a more *complex* way. You can nest any data type inside any other type. This lets you build multidimensional data tables easily.

List of lists

List of lists, often called a matrix are important for organizing and accessing data

Here's a way to make a 3 x 3 table of values.

```
>>> M = [[1,2,3], [4,5,6],[7,8,9]]
>>> M[1] # second row (starts with index 0)
[4,5,6]
>>>M[1][2] # second row, third element
6
```

Here's a way to store sequence alignment data:

Four sequences aligned:

```
AT-TG
AATAG
T-TTG
AA-TA
```

The alignment in a list of lists.

```
aln = [['A', 'T', '-', 'T', 'G'],
['A', 'A', 'T', 'A', 'G'],
['T', '-', 'T', 'T', 'G'],
['A', 'A', '-', 'T', 'A']]
```

Get an the full length of one sequence:

```
>>> seq = aln[2]
>>> seq
['T', '-', 'T', 'T', 'G']
```

> *Use the outer most index to access each sequence*

Retrieve the nucleotide at a particular position in a sequence.

```
>>> nt = aln[2][3]
>>> nt
'T'
```

> *Use the outer most index to access the sequence of interest and the inner most index to access the position*

Get every nucleotide in a single column:

```
>>> col = [seq[3] for seq in aln]
>>> col
['T', 'A', 'T', 'T']
```

> *Retrieve each sequence from the aln list then the 3rd column for each sequence.*

Lists of dictionaries

You can nest dictionaries in lists as well:

```
>>> records = [
... {'name' : 'actgctagt', 'accession' : 'ABC123', 'genetic_code' : 1},
... {'name' : 'ttaggttta', 'accession' : 'XYZ456', 'genetic_code' : 1},
... {'name' : 'cgcgatcgt', 'accession' : 'HIJ789', 'genetic_code' : 5}
... ]
>>> records[0]['name']
'actgctagt'
>>> records[0]['accession']
'ABC123'
>>> records[0]['genetic_code']
1
```

> *Here you can retrieve the accession of one record at a time by using a combination of the outer index and the key 'accession'*

Dictionaries of lists

And, if you haven't guessed, you can nest lists in dictionaries

Here is a dictionary of kmers. The key is the kmer and its values is a list of postions

```
>>> kmers = {'ggaa': [4, 10], 'aatt': [0, 6, 12], 'gaat': [5, 11], 'tgga':
... [3, 9], 'attg': [1, 7, 13], 'ttgg': [2, 8]}
>>> kmers
{'tgga': [3, 9], 'ttgg': [2, 8], 'aatt': [0, 6, 12], 'attg': [1, 7, 13], 'ggaa': [4, 10],
'gaat': [5, 11]}
>>>
>>> kmers['ggaa']
[4, 10]
>>> len(kmers['ggaa'])
2
```

> *Here we can get a list of the positions of a kmer by using the kmer as the key. We can also do things to the returned list, like determining its length. The length will be the total count of this kmers.*

You can also use the `get()` method to retrieve records.

```
>>> kmers['ggaa']
[4, 10]
>>> kmers.get('ggaa')
[4, 10]
```

> *These two statements returns the same results, but if the key does not exist you will get nothing and not an error.*

Dictionaries of dictionaries

Dictionaries of dictionaries is my favorite!! You can do so many useful things with this data structure. Here we are storing a gene name and some different types of information about that gene, such as its, sequence, length, description, nucleotide composition and length.

```
>>> genes = {
... 'gene1' : {
...     'seq' : "TATGCC",
...    'desc' : 'something',
...     'len' : 6,
... 'nt_comp' : {
...             'A' : 1,
...             'T' : 2,
...             'G' : 1,
```

```
...           'C' : 2,
...           }
...   },
...
... 'gene2' : {
...     'seq' : "CAAATG",
...    'desc' : 'something',
...     'len' : 6,
... 'nt_comp' : {
...           'A' : 3,
...           'T' : 1,
...           'G' : 1,
...           'C' : 1,
...           }
...     }
... }
>>> genes
{'gene1': {'nt_comp': {'C': 2, 'G': 1, 'A': 1, 'T': 2}, 'desc': 'something',
'len': 6, 'seq': 'TATGCC'}, 'gene2': {'nt_comp': {'C': 1, 'G': 1, 'A': 3, 'T': 1},
'desc': 'something', 'len': 6, 'seq': 'CAAATG'}}
>>> genes['gene2']['nt_comp']
{'C': 1, 'G': 1, 'A': 3, 'T': 1}
```

> *Here we store a gene name as the outermost key, with a second level of keys for qualities of the gene, like sequence, length, nucleotide composition. We can retrieve a quality by using the gene name and quality in conjunction.*

Pipelines

Often you will want to run a series of programs, with the results of one required by the next. We can write a pipeline to do this. This is commonly written in bash as shell script, as a make file, or in a scripting language like Python on Perl.

subprocess Module

The subprocess module had a function called `run()` that can be used to

- execute commands
- get the exit status of the command

        - 0 => Good
        - anything else => Bad

- get STDOUT and STDERR of the command
    - STDOUT => regular output
    - STDERR => any warning messages or error messages

Let's run a command that we know will not have any errors:

```
>>> import subprocess
>>>
>>> cmd = subprocess.run(args=["ls" ,"-l" , "seq.nt" ] , stdout = subprocess.PIPE, stderr=subprocess.PIPE)
>>>
>>> cmd.stdout.decode('utf-8')
'-rw-r--r--  1 smr  SGC\\Domain Users  142 Feb 28 13:22 seq.nt\n'
>>>
>>> cmd.stderr.decode('utf-8')
''
>>>
>>> cmd.returncode
0
```

> *We have content in stdout*
>
> *We have nothing in sterr*
>
> *Our exit code is 0*

Let's see what happens when we introduce a problem. 'seq.aa' does not exist.

```
>>> cmd = subprocess.run(args=["ls" ,"-l" , "seq.nt" , "seq.aa"] , stdout = subprocess.PIPE, stderr=subprocess.PIPE
>>> cmd.stdout.decode('utf-8')
'-rw-r--r--  1 smr  SGC\\Domain Users  142 Feb 28 13:22 seq.nt\n'
>>> cmd.stderr.decode('utf-8')
'ls: seq.aa: No such file or directory\n'
>>> cmd.returncode
1
```

> *We have content in stdout*
>
> *We have an error message in sterr*
>
> *Our exit code is 1*

shell=True

If you use the option shell=True, you can use a complete command, not broken into components like above

```
>>> cmd = subprocess.run("echo atc | wc -c" , shell=True, stdout = subprocess.PIPE, stderr=subprocess.PIPE)
>>> cmd.stdout.decode('utf-8')
'       4\n'
>>> cmd.returncode
0
```

Use Error Code to Control Pipeline

If the error code is good (0) then we can proceed, if the error code is bad (!=0) stop.

```
#!/usr/bin/env python3
import subprocess
import sys

blastcmd = "blastx -query test.query -db ~/dbs/uniprot_sprot.fasta -outfmt 7 -out test.blastout.tab -evalue 1e-5"
countcmd = 'grep \'hits found\' test.blastout.tab  | perl -ne \'m/(\d+)/; $count=$1; print $count,"\n"\''

blastcmd_run = subprocess.run(blastcmd, shell=True , stdout = subprocess.PIPE, stderr=subprocess.PIPE)
if blastcmd_run.returncode == 0:
  # run count cmd if blast exit code is 0
```

```
  countcmd_run = subprocess.run(countcmd,  shell=True , stdout =
subprocess.PIPE, stderr=subprocess.PIPE)
else:
  sys.exit("BLAST had issues " + blastcmd_run.stderr.decode('utf-8'))


if countcmd_run.returncode == 0:
 if int(countcmd_run.stdout.decode('utf-8')) > 0:
  # parse results
  print("We will put parsing code here")
 else:
  sys.exit("no hits")
else:
 sys.exit("count had issues " + countcmd_run.stderr.decode('utf-8'))
```

Biopython

What is biopython?

Biopython is a collection of python modules that contain code for manipulating biological data. Many handle sequence data and common analysis and processing of the data including reading and writing all common file formats. Biopython will also run blast for you and parse the output into objects inside your script. This requires just a few lines of code.

Installing Biopython

This is very straightforward once you have anaconda or minconda installed. I use miniconda because it's smaller. We are going to use `sudo` because this will give us permission to install in the 'correct' directory python is expecting to find the modules. Other users will be able to use it too. Using `sudo` can cause problems, but it's ok here. You will need the administrator password for the machine. If you don't have this, ask the person who does administration on your machine.

```
% sudo conda install biopython
WARNING: Improper use of the sudo command could lead to data
loss
or the deletion of important system files. Please double-check your
```

```
typing when using sudo. Type "man sudo" for more information.

To proceed, enter your password, or type Ctrl-C to abort.

Password:
Fetching package metadata ...........
Solving package specifications: .

Package plan for installation in environment /anaconda3:

The following NEW packages will be INSTALLED:

    biopython: 1.69-np113py36_0

The following packages will be UPDATED:

    conda:     4.3.29-py36hbf39572_0 anaconda --> 4.3.30-
py36h173c244_0

The following packages will be SUPERSEDED by a higher-priority
channel:

    conda-env: 2.6.0-h36134e3_0      anaconda --> 2.6.0-h36134e3_0

Proceed ([y]/n)?

conda-env-2.6. 100%
|###################################################| Time:
0:00:00   4.02 MB/s
biopython-1.69 100%
|###################################################| Time:
0:00:00  21.02 MB/s
conda-4.3.30-p 100%
|###################################################| Time:
0:00:00  43.61 MB/s

```

See if the install worked

```
python3
>>> import Bio
>>> print(Bio.__version__)
1.69
```

If we get no errors, biopython is installed correctly.

Biopython documentation

Biopython wiki page

http://biopython.org/

Getting started

http://biopython.org/wiki/Category%3AWiki_Documentation

Biopython tutorial

ttp://biopython.org/DIST/docs/tutorial/Tutorial.html#chapter:Bio.SeqIO

Complete tree of Biopython Classes

http://biopython.org/DIST/docs/api/Bio-module.html

Working with DNA and protein sequences

This is the core of biopython. And uses the Seq object. Seq is part of Bio. This is denoted Bio.Seq

Visit biopython.org to read about Sequence objects

```
#!/usr/bin/env python3
import Bio.Seq
seqobj = Bio.Seq.Seq('ATGCGATCGAGC')
# convert to string with str(seqobj)
seq_str = str(seqobj)
print('{} has {} nucleotides'.format( seq_str , len(seq_str)))
```

produces

```
ATGCGATCGAGC has 12 nucleotides
```

From ... import ...

Another way to import modules is with `from ... import ...` . This saves typing the Class name every time. Bio.Seq is the class name. Bio is the superclass. Seq is a subclass inside Bio. It's written Bio.Seq. Seq has several different subclasses, of which one is called Seq. So we have Bio.Seq.Seq. To make the creation simpler, we call Seq() after we import with `from ... import ...` like this

```
#!/usr/bin/env python3
from Bio.Seq import Seq
seqobj=Seq('ATGCGATCGAGC')
seq_str=str(seqobj)
protein = seqobj.translate()
prot_str = str(protein)
print('{} translates to {}'.format(seq_str,prot_str))
```

produces

```
ATGCGATCGAGC
```

Bio.Alphabets

Visit biopython.org to read about Sequences and Alphabets

A Seq object likes to know what alphabet it uses A,C,G,T for DNAAlpabet etc. Not essential for most uses, but prevents you trying to translate a protein sequence!

Specific Alphabets

For DNA

```
>>> seqobj
Seq('ATG', Alphabet())
>>> from Bio.Alphabet import DNAAlphabet
>>> seqobj=Seq('ATG',DNAAlphabet())
>>> seqobj
Seq('ATG', DNAAlphabet())
>>> seqobj.translate()
Seq('M', ExtendedIUPACProtein())
```

For proteins

```
>>> seqobj = Seq('MGT')
>>> seqobj.translate()
Seq('X', ExtendedIUPACProtein())
```

> *'X' Thats not right! Wait! Why did python let us translate MGT, its not DNA?*

```
>>> from Bio.Alphabet import ProteinAlphabet
>>> seqobj = Seq('MGT', ProteinAlphabet())
>>> seqobj.translate()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/smr/anaconda3/envs/py3.6/lib/python3.5/site-packages/Bio/Seq.py", line 1059, in translate
    raise ValueError("Proteins cannot be translated!")
ValueError: Proteins cannot be translated!
```

> *Thats better.*

Extracting a subsequence

You can use a range [0:3] to get the first codon

Visit biopython.org to read about Slicing a sequence

```
>>> seqobj=Seq('ATGCGATCGAGC')
>>> seqobj[0:3]
Seq('ATG', Alphabet())
```

Let's use Regular expressions in conjunction with BioPython to get every codon

```
>>> seqobj=Seq('ATGCGATCGAGC')
>>> import re
>>> for codon in re.findall(r"(.{3})",str(seqobj)):
...   print(codon)
...
ATG
CGA
TCG
AGC
>>>
```

Read a FASTA file

We were learning how to read a fasta file line by line. SeqIO.parse() is the main method for reading from almost any file format. We'll need a fasta file. We can use Python_05.fasta which looks like this

```
>seq1
AAGAGCAGCTCGCGCTAATGTGATAGATGGCGGTAAAGTAAATGTCCTATGGGCCACCAATTATGGTGTATGAGTGAATCTCTGGTCCGAGATTCACTGAGTAACTGCTGTACACAGTAGTAACACGTGGAGATCCCATAAGCTTCACGTGTGGTCCAATAAAACACTCCGTTGGTCAAC
>seq2
GCCACAGAGCCTAGGACCCCAACCTAACCTAACCTAACCTAACCTACAGTTTGATCTTAACCATGAGGCTGAGAAGCGATGTCCTGACCGGCCTGT
CCTAACCGCCCTGACCTAACCGGCTTGACCTAACCGCCCTGACCTAACCAGGCTAACCTAACCAAACCGTGAAAAAAGGAATCT
>seq3
ATGAAAGTTACATAAAGACTATTCGATGCATAAATAGTTCAGTTTTGAAAACTTACATTTTGTTAAAGTCAGGTACTTGTGTATAATATCAACTAAAT
>seq4
ATGCTAACCAAAGTTTCAGTTCGGACGTGTCGATGAGCGACGCTCAAAAAGGAAACAACATGCCAAATAGAAACGATCAATTCGGCGATGGAAATC
AGAACAACGATCAGTTTGGAAATCAAAATAGAAATAACGGGAACGATCAGTTTAATAACATGATGCAGAATAAAGGGAATAATCAATTTAATCCAG
GTAATCAGAACAGAGGT
```

Get help on the parse() method with

```
>>> from Bio import SeqIO
>>> help(SeqIO.parse)

Help on function parse in module Bio.SeqIO:

parse(handle, format, alphabet=None)
    Turns a sequence file into an iterator returning SeqRecords.

        - handle   - handle to the file, or the filename as a string
          (note older versions of Biopython only took a handle).
        - format   - lower case string describing the file format.
        - alphabet - optional Alphabet object, useful when the sequence type
          cannot be automatically inferred from the file itself
          (e.g. format="fasta" or "tab")

```

```
16      Typical usage, opening a file to read in, and looping over the
        record(s):
17
18
19
```

You can also visit biopythorg.org to get help on Parsing or Reading Sequences

Lets try it out!

```
1   >>> from Bio import SeqIO
2   >>> filename = "files/seq.nt.fa"
3   >>> for record in SeqIO.parse(filename, "fasta"):
4   ...   print('ID {}'.format(record.id))
5   ...   print('len {}'.format(len(record)))
6   ...   print('alphabet {}'.format(record.seq.alphabet))
7   ...
8   ID seq1
9   len 180
10  alphabet SingleLetterAlphabet()
11  ID seq2
12  len 180
13  alphabet SingleLetterAlphabet()
14  ID seq3
15  len 98
16  alphabet SingleLetterAlphabet()
17  ID seq4
18  len 209
19  alphabet SingleLetterAlphabet()
```

1. In your text editor write a FASTA parser using BioPython and SeqIO.
2. Print out the ID\ttranlstion. Use `translate()` to translate the DNA into protein

Here's a script to read fasta records and print out some information

```
#!/usr/bin/env python3
# assumes we are in the pfb2017 directory
from Bio import SeqIO
for seq_record in SeqIO.parse("files/seq.nt.fa", "fasta"):  # give
filename and format
  print('ID',seq_record.id)
  print('Sequence',str(seq_record.seq))
  print('Length',len(seq_record))

```

Prints this output

```
ID seq1
Sequence
AAGAGCAGCTCGCGCTAATGTGATAGATGGCGGTAAAGTAAATGTCCT
ATGGGCCACCAATTATGGTGTATGAGTGAATCTCTGGTCCGAGATTCA
CTGAGTAACTGCTGTACACAGTAGTAACACGTGGAGATCCCATAAGCT
TCACGTGTGGTCCAATAAAACACTCCGTTGGTCAAC
Length 180
ID seq2
Sequence
GCCACAGAGCCTAGGACCCCAACCTAACCTAACCTAACCTAACCTACA
GTTTGATCTTAACCATGAGGCTGAGAAGCGATGTCCTGACCGGCCTG
TCCTAACCGCCCTGACCTAACCGGCTTGACCTAACCGCCCTGACCTAA
CCAGGCTAACCTAACCAAACCGTGAAAAAAGGAATCT
Length 180
ID seq3
Sequence
ATGAAAGTTACATAAAGACTATTCGATGCATAAATAGTTCAGTTTTGA
AAACTTACATTTTGTTAAAGTCAGGTACTTGTGTATAATATCAACTAAA
T
Length 98
ID seq4
Sequence
ATGCTAACCAAAGTTTCAGTTCGGACGTGTCGATGAGCGACGCTCAA
AAAGGAAACAACATGCCAAATAGAAACGATCAATTCGGCGATGGAAA
TCAGAACAACGATCAGTTTGGAAATCAAAATAGAAATAACGGGAACG
ATCAGTTTAATAACATGATGCAGAATAAAGGGAATAATCAATTTAATC
CAGGTAATCAGAACAGAGGT
Length 209

```

Convert fasta file to python dictionary in one line

There are three ways of doing this that use up more memory if you want more flexibility. `Bio.SeqIO.to_dict()` is the most flexible but also reads the entire fasta file into memory as a python dictionary so might take a lot of time and memory.

```
>>> id_dict = SeqIO.to_dict(SeqIO.parse('files/seq.nt.fa', 'fasta'))
>>> id_dict
{'seq1':
SeqRecord(seq=Seq('AAGAGCAGCTCGCGCTAATGTGATAGATGGCGGTA
AAGTAAATGTCCTATGGGC...AAC', SingleLetterAlphabet()), id='seq1',
name='seq1', description='seq1', dbxrefs=[]), 'seq2':
SeqRecord(seq=Seq('GCCACAGAGCCTAGGACCCCAACCTAACCTAACCT
AACCTAACCTACAGTTTGA...TCT', SingleLetterAlphabet()), id='seq2',
name='seq2', description='seq2', dbxrefs=[]), 'seq3':
SeqRecord(seq=Seq('ATGAAAGTTACATAAAGACTATTCGATGCATAAAT
AGTTCAGTTTTGAAAACTT...AAT', SingleLetterAlphabet()), id='seq3',
name='seq3', description='seq3', dbxrefs=[]), 'seq4':
SeqRecord(seq=Seq('ATGCTAACCAAAGTTTCAGTTCGGACGTGTCGATG
AGCGACGCTCAAAAAGGAA...GGT', SingleLetterAlphabet()), id='seq4',
name='seq4', description='seq4', dbxrefs=[])}

```

Let's retrieve some info from our new dictionary

```
>>> id_dict['seq4']
SeqRecord(seq=Seq('ATGCTAACCAAAGTTTCAGTTCGGACGTGTCGATG
AGCGACGCTCAAAAAGGAA...GGT', SingleLetterAlphabet()), id='seq4',
name='seq4', description='seq4', dbxrefs=[])
>>> id_dict['seq4'].seq
Seq('ATGCTAACCAAAGTTTCAGTTCGGACGTGTCGATGAGCGACGCTC
AAAAAGGAA...GGT', SingleLetterAlphabet())
>>> str(id_dict['seq4'].seq)
'ATGCTAACCAAAGTTTCAGTTCGGACGTGTCGATGAGCGACGCTCAAA
AAGGAAACAACATGCCAAATAGAAACGATCAATTCGGCGATGGAAATC
AGAACAACGATCAGTTTGGAAATCAAAATAGAAATAACGGGAACGATC
AGTTTAATAACATGATGCAGAATAAAGGGAATAATCAATTTAATCCAGG
TAATCAGAACAGAGGT'
>>>
```

> *need to use this format to get the string of the sequence:*
> `str(id_dict['seq4'].seq)`

1. In your text editor create a new FASTA parser that uses `SeqIO.parse` to create a dictionary of all the sequences in your FASTA file.
2. Find all the codons in the first frame.
3. Print out the codons of each sequence in FASTA format.

```
>my_seq
ATG TTC ATC
```

4. CHALLENGE QUESTION: Print out the codons of each sequence in all 6 frames.

Seq methods

Visit biopython.org to read how Sequences act like strings

```
seqobj.count("A")  # counts how many As are in sequence
seqobj.find("ATG") # find coordinate of ATG (-1 for not found)
```

SeqRecord objects

SeqIO.Parse generates Bio.SeqRecord.SeqRecord objects. These are annotated Bio.Seq.Seq objects.

Main attributes:

- id - Identifier such as a locus tag (string)

- seq - The sequence itself (Seq object or similar)

Access these with `sr.id` and `sr.seq` . `str(sr.seq)` gets the actual sequence string.

Additional attributes:

- name - Sequence name, e.g. gene name (string)
- description - Additional text (string)
- dbxrefs - List of database cross references (list of strings)
- features - Any (sub)features defined (list of SeqFeature objects)
- annotations - Further information about the whole sequence (dictionary). Most entries are strings, or lists of strings.
- letter_annotations - Per letter/symbol annotation (restricted dictionary). This holds Python sequences (lists, strings or tuples) whose length matches that of the sequence. A typical use would be to hold a list of integers representing sequencing quality scores, or a string representing the secondary structure.

SeqRecord objects have .format() to convert to a string in various formats

```
>>> seq.format('fasta')
'>seq1\nAAGAGCAGCTCGCGCTAATGTGATAGATGGCGGTAAAGTAAATGTCCTATGGGCCACCAA\nTTATGGTGTATGAGTGAATCTCTGGTCCGAGATTCACTGAGTAACTGCTGTACACAGTAG\nTAACACGTGGAGATCCCATAAGCTTCACGTGTGGTCCAATAAAACACTCCGTTGGTCAAC\n'

```

Retrieving annotations from GenBank file

To read sequences from a genbank file instead, not much changes.

```
#!/usr/bin/env python3
from Bio import SeqIO
for seq_record in SeqIO.parse("files/sequence.gb", "genbank"):
  print('ID',seq_record.id)
  print('Sequence',str(seq_record.seq)[0:60],'...')
  print('Length',len(seq_record))
```

```
ID NM_204156.1
Sequence
GGCCCCGGCCGGTGGGGCGGGTTGCGTTGCGCTGCGCGGCGGTAGGG
TCTGCGGCCGTGG ...
Length 3193
```

File Format Conversions

Many are straightforward, others are a little more complicated because the alphabet can't be determined from the data. It's usually easier to go from richer formats to simpler ones.

```
#!/usr/bin/env python3
from Bio import SeqIO
fasta_records = SeqIO.parse("files/seq.nt.fa", "fasta")
tab_records = SeqIO.write(records , 'files/seqs.tab' , 'tab')
```

Produces

```
% more seqs.tab
seq1
AAGAGCAGCTCGCGCTAATGTGATAGATGGCGGTAAAGTAAATGTCCTA
TGGGCCACCAATTATGGTGTATGAGTGAATCTCTGGTCCGAGATTCACT
GAGTAACTGCTGTACACAGTAGTAACACGTGGAGATCCCATAAGCTTCA
CGTGTGGTCCAATAAAACACTCCGTTGGTCAAC
seq2
GCCACAGAGCCTAGGACCCCAACCTAACCTAACCTAACCTAACCTACAG
TTTGATCTTAACCATGAGGCTGAGAAGCGATGTCCTGACCGGCCTGTCC
TAACCGCCCTGACCTAACCGGCTTGACCTAACCGCCCTGACCTAACCAG
GCTAACCTAACCAAACCGTGAAAAAAGGAATCT
seq3
ATGAAAGTTACATAAAGACTATTCGATGCATAAATAGTTCAGTTTTGAA
AACTTACATTTTGTTAAAGTCAGGTACTTGTGTATAATATCAACTAAAT
seq4
ATGCTAACCAAAGTTTCAGTTCGGACGTGTCGATGAGCGACGCTCAAA
AAGGAAACAACATGCCAAATAGAAACGATCAATTCGGCGATGGAAATC
AGAACAACGATCAGTTTGGAAATCAAAATAGAAATAACGGGAACGATC
AGTTTAATAACATGATGCAGAATAAAGGGAATAATCAATTTAATCCAGG
TAATCAGAACAGAGGT
```

Even easier is the convert() method. Let's try fastq to fasta.

```
#!/usr/bin/env python3
from Bio import SeqIO
fasta_records = SeqIO.convert('files/sample.fastq', 'fastq',
'files/sample.converted.fa', 'fasta')
```

Hmm, was that easy or what??!??!!?

Parsing BLAST output

For simple BLAST parsing, ask for output format in tab-separated columns (`-outfmt 6` or `-outfmt 7`) Both these formats are customizable! See next section.

If you want to parse the full output of blast with biopython, it's best to work with XML formatted BLAST output `-outfmt 5` . It breaks the parsing method less easily. Code is stable for working with NCBI blast.

You can get biopython to run the blast for you too. See `Bio.NCBIWWW`

To parse the output, you'll write something like this

```
>>> from Bio.Blast import NCBIXML
>>> result_handle = open("files/test.blastout.xml")
>>> blast_records = NCBIXML.parse(result_handle)
>>> for blast_record in blast_records:
>>>   for alignment in blast_record.alignments:
>>>     for hsp in alignment.hsps:
>>>       if hsp.expect < 1e-10:
>>>         print('id', alignment.title)
>>>         print('E = ' , hsp.expect)

```

1. Run BLAST with output in XML format.
    1. Query = query.fa
    2. Program = blastx
    3. Database = Swissprot/UniProt
        1. Already downloaded, path:
        2. Don't do this now, but FYI: This DB can be downloaded from ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/knowledgebase/complete/uniprot_sprot.fasta.gz

2. Parse the results and print the following "queryName\thitName\te-value\thitDesc" if the evalue is better than 1e-10.

There are many other uses for Biopython

- reading multiple sequence alignments
- searching on remote biological sequence databases
- working with protein structure (requires numpy to be installed)
- biochemical pathways (KEGG)

- drawing pictures of genome and sequence features
- population genetics

Why use biopython

Massive time saver once you know your way around the classes.

Reuse someone else's code. Very quick parsing of many common file formats.

Clean code.