Programming For Biology 2022

programmingforbiology.org

Instructors
Simon Prochnik
Sofia Robb

Python Lectures

Python 1

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 and 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 the 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 Interpreter

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

`$ python3`

Note: '$' indicates the command line prompt. Recall from Unix 1 that every computer can have a different prompt!

First Python Commands:

```
>>> print("Hello, PFB2022!")
Hello, PFB2022!
```

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

Python Scripts are Text Files

- 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').
- We could call the file `hello.py`

File Contents:

```
print("Hello, PFB2022!")
```

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 `python3` on the command line.

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

```
% python3 hello.py
```

This produces the following result in the Terminal:

```
Hello, PFB2022!
```

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 hello.py`

You can look at the permissions with

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

The first 10 characters you see displayed on the line have special meanings. The first character (`-`) tells you what kind of file `hello.py` is. `-` means a normal file, `d` a directory, `l` a link. The next nine characters come in three sets of three. The first set refers to the your permissions, the second set your group's permissions, and the last set to everyone else. Each three character set shows in order `rwx` for read, write, execute. If someone doesn't have a permission, a `-` is displayed instead of a letter. 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 `!` (exclamation mark or bang) 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 hello.py now looks like this

```
#!/usr/bin/env python3
print("Hello, PFB2022!")
```

Now you can simply type the symbol for the current directory `.` followed by a `/` and the name of the script to run it. Like this

```
% ./hello.py
Hello, PFB2022!
```

Syntax

Python Variable Names

A Python variable name is a name used to identify a variable, function, class, module, or other object. A variable name 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 such as `@`, `$`, and `%` within a variable name. Python is a case sensitive programming language. Thus, `seq_id` and `seq_ID` are two different variable names in Python.

Naming conventions for Python Variable Names

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

Picking good variable names for the objects you name yourself is very important. Don't call your variables things like `items` or `my_list` or `data` or `var`. Except for where you have a very simple piece of code, or you are plotting a graph, don't call your objects `x` or `y` either. All these name examples are not decriptive of what kind of data you will find in the variable or object. Worse is to call a variable that contains gene names as `sequences`. Why is this such a bad idea? Think about what would happen if you filled your car up at a store labelled 'gas station' that sold lemonade. In computer science, names should always accurately describe the object they are attached to. This reduces possibility of bugs in your code, makes it much easier to understand if you come back to it after six months or share your code with someone, and makes it faster to write code that works right. Even though it takes a bit of time and effort to think up a good name for an object, it will prevent so many problems in the future!

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 variable 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      list        hash
```

Lines and Indentation

Python denotes a block of code by lines with the same level of indentation. This keeps lines of code that run together organized. Incorrect line spacing and/or indention will cause an error or can 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 the same amount. For example, using four spaces:

```
#!/usr/bin/env python3
message = '' # make an empty variable
for x in (1,2,3,4,5):
    if x > 4:
        print("Hello")
        message = 'x is big'
    else:
        print(x)
        message = 'x is small'
    print(message)
print('All Done!')
```

Comments

Including comments in your code is 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, PFB2022!") # 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 of 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, which can be changed.

For Example:

```
gene_count = 5
# change the value of gene_count
gene_count = 10
```

> Recall the section above on variable and object names (and variables are objects in Python).

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 = 2.531    # this is a float
message = "Welcome to Python" # this is a string
```

`10`, `2.531`, and `"Welcome to Python"` are singular (scalar) pieces of data, and each is stored in its own variable.

Collections of data can also be stored in special data types, i.e., tuples, lists, sets, and dictionaries. You should always try to store like with like, so each element in the collection 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. Why do you think this might be?

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

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

Index	Value
0	atg
1	aaa
2	agg

> The list index starts at 0

Tuples

- Tuples are similar to lists and contain ordered, *indexed* collections 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

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, and sets of Key/Value pairs are separated by commas
- A colon is written between each key and value. Commas separate key:value pairs.

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

Key	Value
TP53	GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC
BRCA1	GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA

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 module named `sys` 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 until 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!

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
`id(data)`	tells you the unique object id

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

```
>>> data = [2,4,6]
>>> type(data)
<class 'list'>
>>> data = 5
>>> type(data)
<class 'int'>
>>> id(data)
44990666544
```

Link to Python 1 Problem Set

Python 2

Operators

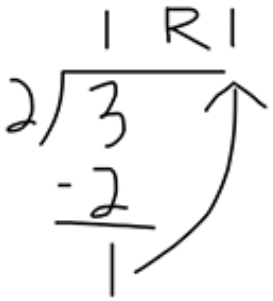
An operator in a programming language is a symbol that tells the compiler or interpreter to perform specific mathematical, relational or logical operation and produce a result. Here we explain the concept of operators.

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

Modulus



Floor examples

```
>>> 3/2
1.5
>>> 3//2
1
>>> -11/3
-3.6666666666666665
>>> -11//3
-4
>>> 11/3
3.6666666666666665
>>> 11//3
3
```

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 + 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

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

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

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
```

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

Note: Find out more about bitwise operators.

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

Use `bool()` to 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
```

Logic: Control Statements

Control Statements are used to direct the flow of your code and create the opportunity for decision making. The foundation of control statements is building 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
```

else

- The `if` portion of the if/else statement behaves 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
```

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
```

Let's change count to 20, which statement block gets executed?

```
count = 20
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:

```
20 is less than 50
```

What happens when count is 50?

```
count = 50
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:

```
50 must be 50
```

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 is a list of functions that take numbers as arguments. These do useful things like rounding.

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 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 imported into our script if we want to use them. To include them, at the top 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
```

Comparing two numbers

Oftentimes, 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:

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

It returns three different values depending on x and y

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

Link to Python 2 Problem Set

Python 3

Sequences

In the next section, we will learn about strings, tuples, and lists. These are all examples of python sequences. 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 different types of data. We'll see 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 (( ))
5
>>>len(['dog', 'cat', 'bird'])  # length of a list
3
```

You can also use string-specific functions on strings, but not on lists and vice versa. We'll learn more about this later on. `rstrip()` is a string method or function. You get an error if you try to use it on a list.

```
>>> 'ACGTGA'.rstrip('A')
'ACGTG'
>>> ['dog', 'cat', 'bird'].rstrip()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'rstrip'
```

What functions go with my object?

How do you find out what functions work with an object? There's a handy function `dir()`. As an example what functions can you call on our string `'ACGTGA'`?

```
>>> dir('ACGTGA')
['__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']
```

`dir()` will return all atributes of an object, among them its functions. Technically, functions belonging to a specific class (object type) are called methods.
You can call `dir()` on any object, most often, you'll use it in the interactive Python shell.

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 quotation marks:

- Single and double quotes are equivalent.
- A variable name inside quotes is just the string identifier, not the value stored inside the variable. `f''` or f-strings are useful for variable interpolation in python

- Triple quotes (single or doubel) 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
```

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 or 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.

`print()` and Common Errors

Let's look at the typical errors you will encounter when you use the `print()` function.

What will happen if you forget to close your quotes?

```
>>> print("GGTCTAC)
  File "<stdin>", line 1
    print("GGTCTAC)
                  ^
SyntaxError: EOL while scanning string literal
```

> We get a 'SyntaxError' if the closing quote is not used

What will happen if you forget to enclose a string you want to print in quotes?

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

> We get a 'NameError' when the literal string is not enclosed in quotes because Python is looking for a variable with the name GGTCTAC

```
>>> print "boo"
  File "<stdin>", line 1
    print "boo"
              ^
SyntaxError: Missing parentheses in call to 'print'
```

In python2, the command was `print`, but this changed to `print()` in python3, so don't forget the parentheses!

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 = '')`

A neater 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.

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
```

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. Recall that `dir()` will tell you all the methods that are available for an object. The `lower()` function is a string method.

Let's create a new string object.

```
dna = "ATGCTTG"
```

> Look familiar?

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.

Find and Count

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

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

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

Replace one string with another

`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.

You can get to any part of a string with the following syntax [start : end : step].

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 start counting at 0 and remember that python counts the gaps between each character, starting with zero.

```
index      0   1   2   3   4   5   6   7   8 ...
string       A   T   T   A   A   A   G   G  ...
```

```
>>> 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.

Reverse a string or a list

There is no reverse function, you need to use a slice with step -1 and empty start and end.

For a string, it looks like this

```
>>> dna='GATGAA'
>>> dna[::-1]
'AAGTAG'
```

Other String Methods

Since these are methods, be sure to use in this syntax `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.

split

`split` is a method or a way to break up a string on a set of characters. What is returned is a list of elements with the characters that were used for breaking are removed. We will be going over lists in more detail in the next session. Don't get too worried about this.

Lets look at this string:

```
00000xx000xx000000000000xx0xx00
```

Let's split on 'xx' and get a list of the 0's

What is the 's' in `s.split(delim)` ?

What is the 'delim' in `s.split(delim)` ?

Let's try it:

```
>>> string_to_split='00000xx000xx000000000000xx0xx00'
>>> string_to_split.split('xx')
['00000', '000', '000000000000', '0', '00']
>>> zero_parts = string_to_split.split('xx')
>>> print(zero_parts)
['00000', '000', '000000000000', '0', '00']
```

> We started with a string and now have a list with all the delimeters removed

Here is another example. Let's split on tabs to get a list of numbers in tab separated columns.

```
>>> input_expr = '4.73\t7.91\t3.65'
>>> expression_values = input_expr.split('\t')
>>> expression_values
['4.73', '7.91', '3.65']
```

join

`join` is a method or a way to take a list of elements, of things, and turn them into a string with something put in between each element. List will be covered in the next session in more detail.

Let's join a list of Ns `list_of_Ns = ['NNNNN', 'NNN', 'N', 'NNNNNNNNNNNNNN', 'NN']` on 'xx' to get this string:

```
NNNNNxxNNNxxNxxNNNNNNNNNNNNNNNxxNN
```

What is the 's' in `s.join(list)` ?

What is the 'list' in `s.join(list)` ?

```
>>> list_of_Ns = ['NNNNN', 'NNN', 'N', 'NNNNNNNNNNNNNN', 'NN']
>>> list_of_Ns
['NNNNN', 'NNN', 'N', 'NNNNNNNNNNNNNN', 'NN']
>>>
>>> string_of_elements_with_xx = 'xx'.join(list_of_Ns)
>>> string_of_elements_with_xx
'NNNNNxxNNNxxNxxNNNNNNNNNNNNNNNxxNN'
```

> We started with a list now have all the elements now in one string with the delimiter added in between each element.

Let's take a list of expression values and create a tab delimited string that will open nicely in a spreadsheet with each value in its own column:

```
>>> expression_values = ['4.73', '7.91', '3.65']
>>>expression_values
['4.73', '7.91', '3.65']
>>> expression_value_string = '\t'.join(expression_values)
>>> expression_value_string
'4.73\t7.91\t3.65'
```

> print this to a file and open it in Excel! It is beautiful!!

String Formatting

Strings can be formated using new f-strings `f''`, `f""` and `f'''`

`'''`. That last one is the triple quote multiline string. 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.

```
>>> f'This sequence: {dna} is {dna_len} nucleotides long and is found in
{gene_name}.'
'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 f-strings. 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 variable that needs to be inserted.

Something very nice about f-strings is that you can print int and string variable types without converting first.

You will often put f-strings inside print functions.

```
>>> print(f'This sequence: {dna} is {dna_len} nucleotides long and is found in
{gene_name}.')
This sequence: TGAACATCTAAAAGATGAAGTTT is 23 nucleotides long and is found in
Brca1.
```

There is an older function `format()` that is similar, but not as concise. Here's an example in case you see one in older code:

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

The f-string mini-language

So far, we have just used `{}` to show where to insert the value of a variable in a string. You can add special characters inside the `{}` to change the way the variable is formatted when it's inserted into the string.

Lets right justify some numbers.

```
>>> print( f"{2:>5}" )   # 2 is the number we want to print, the characters
after the colon define
                         # the formatting e.g. > for right justify 5 for min
field width
    2
>>> print( f"{20:>5}" )
   20
>>> print( f"{200:>5}" )
  200
```

How about padding with zeroes? This means the five-character field will be filled as needed with zeroes to the left of any numbers you want to display

```
>>> print( f"{2:05}" )
00002
>>> print( f"{20:05}" )
00020
```

Use a `<` to indicate left-justification.

```
>>> print( f"{2:<5} genes" )
2     genes
>>> print( f"{20:<5} genes" )
20    genes
>>> print( f"{20:<5} genes" )
200   genes
```

Center aligning is done with `^` instead of `>` or `<`. You can also pad with characters other than 0. Here let's try `_` or underscore as in `:_^`. The fill symbol goes before the alignment symbol.

```
>>> print( f"{2:_^10}" )
____2_____
>>> print( f"{20:_^10}" )
____20____
>>> print( f"{200:_^10}" )
___200____
```

Summary of special formatting symbols so far

Here are some of the ALIGNMENT options:

Option	Meaning	
`<`	Forces the field to be left-aligned within the available space (this is the default for most objects).	
`>`	Forces the field to be right-aligned within the available space (this is the default for numbers).	
`=`	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form '+000000120'. This alignment option is only valid for numeric types.	
`^`	Forces the field to be centered within the available space.	

> Here's an example
>
> `{  :    x  <  10   s}`

> fill with `x`
> left justify `<`
> `10` a field of ten characters
> `s` a string

Common Types

type	description
b	convert to binary
d	decimal integer
e	exponent, default precision is 6, uses `e`
E	exponent, uses `E`
f	floating point, default precision 6 (also F)
g	general number, float for values close to 0, exponent for others; also G
s	string, default type (see example above)
x	convert to hexadecimal, also X
%	converts to % by multiplying by 100

What's the point?

So much can be done with the `format()` function. Here is one last example, but not the last functionality of this function. Let round a floating point number to fewer decimal places, starting with a lot. (The default is 6.) Note that the function rounds to the nearest decimal place, but not always exactly the way you expect because of the way computers represent decimals with 1s and 0s.

```
>>> f'{3.141592653589793:f}'
'3.141593'   # note this is converted to a string, useful for printing
             # they are f-strings, after all, so this makes sense
>>> f'{3.141592653589793:.4f}'
'3.1416'
```

F-strings allow you to embed expressions inside string literals, so you can do things like this. Neat.

```
>>> f'sum is {3+4}'
'sum is 7'
>>> f'sum is {3.1234+4.4324:.2f}'
'sum is 7.56'
```

Link to Python 3 Problem Set

Python 4

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

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

Tuples

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

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

Many functions and methods return tuples like `math.modf(x)`. This function returns the fractional and integer parts of `x` in a two-item tuple. Here the is no reason to change this sequence.

```
>>> math.modf(2.6)
(0.6000000000000001, 2.0)
```

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 accessed 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.

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

use the syntax [start : end : step] to slice and dice your python seqeunce

```
>>> 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, reverse=False)`	returns a sorted list based on the key provided	`sorted(['a','A','z'])` returns `['A', 'a', 'z']`
`sorted(list, key=str.lower, 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(nums) ; 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	nums.insert(0,100) ; print(nums) returns [100, 1, 2, 3]
`list.extend(new_list)`	appends `new_list` to the end of `list`	nums.extend([7, 8]) ; print(nums) 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]
`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!

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. Loops let us do this.

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:
  # these statements get executed every time the code enters the loop
  statement1
  statement2
  more_statements
# code below here gets executed after the while loop exits
rest_of_code_goes_here
more_code
```

> 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:            # this is normally a bug!!
  print("count:" , count)   # forgot to increment count in the loop!!
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 printing forever use ctrl+c. Behavior like this is almost always due to a bug in the code.

A better way to write an infinite loop is with `True`. You'll need to include something like `if ...: break`

```
#!/usr/bin/env python3
count=0
while True:
  print("count:",count)
  # you probably want to add if...: break
  # so you can get out of the infinite loop
print('Finished the loop')
```

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 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
```

Loop Control

Loop 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

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

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]

Link to Python 4 Problem Set

Python 5

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 access 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
```

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':
'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTCTAGAGCCAC
CGTCCAGGGAGCAGGTAGCTGCTGGGCTCCGGGGACACTTTGCGTTCGGGCTGGGAGCGTG'}
```

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

Accessing Each Dictionary Key/Value

Since a dictionary is an iterable object, 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
```

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.

```
>>> genes = {}
>>> print(genes)
{}
>>> genes['Brca1'] =
'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
>>> genes['TP53'] =
'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC'
>>> print(genes)
{'Brca1':
'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA',
'TP53':
'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC'}
```

> 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
>>>
```

Building a Dictionary one Key/Value at a Time using a loop

Now we have all the tools to build a dictionary one key/value using a for loop. This is how you will be building dictionaries more often in real life.

Here we are going to count and store nucleotide counts:

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

# create a new empty dictionary
nt_count={}

# loop example from loops lecture
dna = 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
for nt in dna:

  # is this nt in our dictionary?
  if nt in nt_count:
```

```
    # if it is, lets increment our count
    previous_count = nt_count[nt]
    new_count = previous_count + 1
    nt_count[nt] = new_count
  else:
    # if not, lets add this nt to our dictionary and make count = 1
    nt_count[nt] = 1;

print(nt_count)
```

> {'G': 20, 'T': 21, 'A': 13, 'C': 16}

What is another way we could increment our count?

```
nt_count={}

dna = 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
for nt in dna:
  if nt in nt_count:
    nt_count[nt] += 1
  else:
    nt_count[nt] = 1;

print(nt_count)
```

> remember that `count=count+1` is the same as `count+=1`

Sorting Dictionary Keys

If you want to print the contents of a dictionary, you should 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): # python allows you to use this shortcut in a for
loop
                               # you don't have to write genes.keys() in a for
loop
                               # to iterate over the keys
  print(gene_key, '=>' , genes[gene_key])
```

> This will print keys in the same order every time you run your script. Dictionaries are unordered, so without sorting, you'll get a different order every time you run the script, which could be confusing.

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.pop(key)`	Removes the key:value pair and returns the value
`dict.keys()`	Returns list of keys
`dict.get(key, default = None)`	get value from dict[key], use default if not 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

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

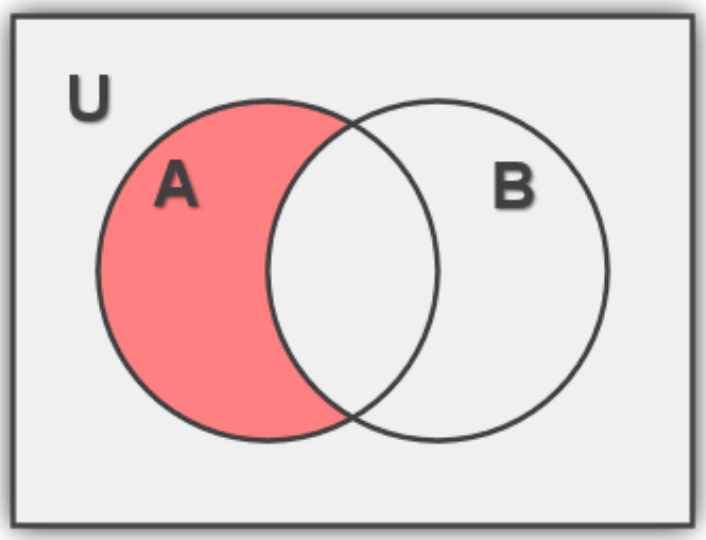
```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
{'a', 'r', 'b', 'c', 'd'}
```

> 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.

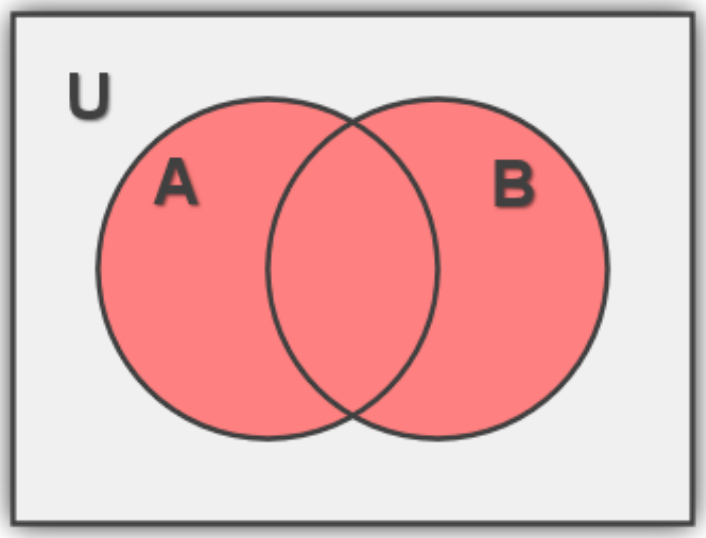



```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a - b
{'r', 'd', 'b'}
```

> 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.

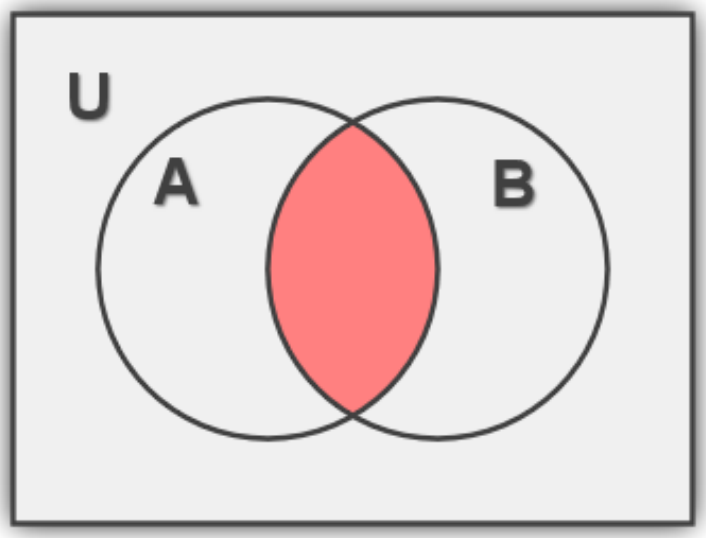



```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a | b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
```

> 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.

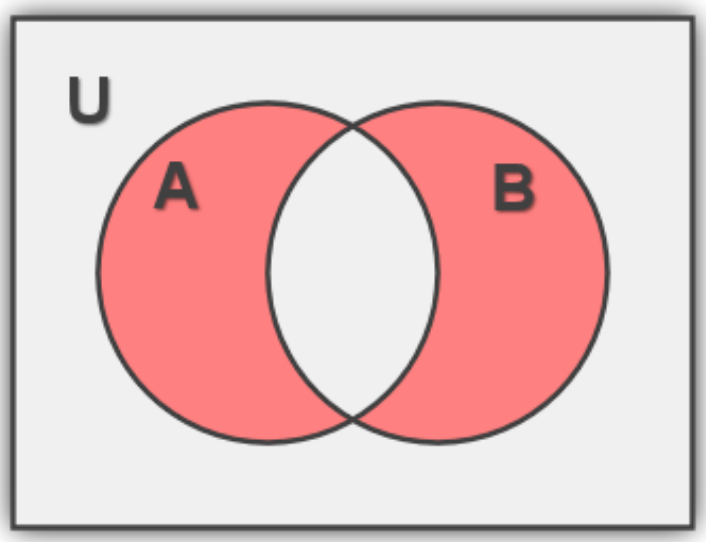


```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a & b
{'a', 'c'}
```

> 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)
`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

Build a dictionary of NT counts using a set and loops

Let us put a twist on our nt count script. Let's use a set to find all the unique nts, then use the string `count()` method to count the nucleotide instead if incrementing the count as we did earlier.

Code:

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

# create a new empty dictionary
nt_count = {}

# get a set of unique characters in our DNA string

dna = 'GTACCNTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
unique = set(dna)

print('unique nt: ', unique) ## {'C', 'A', 'G', 'T', 'N'}

# iterate through each unique nucleotide
for nt in unique:
  # count the number of this unique nt in dna
  count = dna.count(nt)

  # add our count to our dict
  nt_count[nt] = count


print('nt count:', nt_count)
```

Output:

```
unique nt:  {'N', 'C', 'T', 'G', 'A'}
nt count: {'G': 20, 'T': 21, 'A': 13, 'C': 16, 'N': 1}
```

> We have the count for all NTs even ones we might not expect.

[Link to Python 5 Problem Set]

Python 6

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, PFB2022!")
Hello, PFB2022!
```

> 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. When you are done entering text, press the enter key to end the input. A newline character is not included in the input.

```
>>> user_input = input("Type Something Now: ")
Type Something Now: Hi
>>> print(user_input)
Hi
>>> in_str = input("Enter a number: ")
>>> type(in_str)
<class 'str'>
>>> num = int(in_str)
>>> num
445
```

Reading from a File

Mostly you will read data from 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

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

> `seq_file_obj` 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 and to distinguish it from the filename.

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 from the file. Let's read all the contents at once.

Before we do that, let's go to the command line and `cat` the contents of the file to see what's in it first.

```
$ cat seq.nt.txt
ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAG
ACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG
$
```

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.

```
>>> seq_file_obj = open("seq.nt.txt","r")
>>> contents = seq_file_obj.read()
>>> print(contents)  # note newline characters are part of the file!
ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAG
ACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG

>>> seq_file_obj.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

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

Output:

```
$ python3 file_for.py
ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAG

ACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG

```

> 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

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

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

Output:

```
$ python3 file_for_rstrip.py
ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAG
ACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG
```

> 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

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

with open("seq.nt.txt","r") as seq_file_obj: #cleans up after exiting
                                             # the 'with' block
  for line in seq_file_obj:
    line = line.rstrip()
    print(line)
#file gets closed for you here.
```

Writing to a File

Writing to a file just required 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".

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

fo = open("writing.txt" , "w")  # note that we are writing so the mode is "w"
fo.write("One line.\n")
fo.write("2nd line.\n")
fo.write("3rd line" + " has extra text\n")
some_var = 5
fo.write("4th line has " + str(some_var) + " words\n")  # the write() method
does not convert ints for you
fo.close()
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

total_nts = 0
# open two file objects, one for reading, one for writing
with open("seq.nt.txt","r") as seq_read, open("nt.counts.txt","w") as seq_write:
  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")

print("Wrote 'nt.counts.txt'")
```

Output:

```
$ python3 file_read_write.py
$ cat nt.counts.txt
71
71
Total: 142
```

> The file we are reading from is named, "seq.nt.txt"
> 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

```
TP53    GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC
BRCA1   GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA
```

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

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

genes = {}
with open("sequence_data.txt","r") as seq_read:
  for line in seq_read:
    line = line.rstrip()
    gene_id,seq = line.split() #split on whitespace
    genes[gene_id] = seq
print(genes)
```

Output:

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

Link to Python 6 Problem Set

Python 7

Regular Expressions

Regular Expressions is a language for pattern matching. Many different computer languages incorporate regular expressions, as do 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 =
'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACC
GGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACCGGTTTCCAAA
GACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG'
>>> 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 =
'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACC
GGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACCGGTTTCCAAA
GACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG'
>>> 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 =
'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACC
GGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACCGGTTTCCAAA
GACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG'
>>> 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 =
'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACC
GGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACCGGTTTCCAAA
GACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG'
>>> 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.

Let' Try It

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, 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, which are symbols 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]`
`[^]`	a carat within a bracketed list of characters indicates anything but the characters that follows

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:

```
g..t
```

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

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

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

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

> matches 867-5309, and 5867-5309 but not 8-67-5309.

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

> matches 867-5309 and 867-53091 but not 5867-5309.

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

> only matche 3 digits followed by a dash followed by 4 digits, not extra characters anywhere are allowed

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).

Let' Try It

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.

```
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"

Let' Try It

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.

```
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.

```
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.

Let' Try It

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 =
'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACC
GGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTT
TCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG'
>>> 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="ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATT
CACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCC
GGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACAAAATACGTTT
TGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACA
GTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCT
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG"
>>> 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

```
>>>
dna="ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATT
CACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCC
GGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACAAAATACGTTT
TGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACA
GTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCT
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG"
>>> for (upstream, downstream) in re.findall( r"(.{50})TATTAT(.{25})"  , dna ):
...   print("upstream:" , upstream)
...   print("downstream:" , downstream)
...
upstream: TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
downstream: CCGGTTTCCAAAGACAGTCTTCTAA
upstream: TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
downstream: CCGGTTTCCAAAGACAGTCTTCTAA
```

1. This code executes the `findall()` function once
2. The subpatterns are returned in a tuple
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.

```
>>>
dna="ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATT
CACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCC
GGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACAAAATACGTTT
TGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACA
GTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCT
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG"
>>> for match in re.finditer(r"(.{50})TATTAT(.{25})"  , dna):
...   print("upstream:" , match.group(1))
...   print("downstream:" , match.group(2))
...
upstream: TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
downstream: CCGGTTTCCAAAGACAGTCTTCTAA
upstream: TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
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="ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATT
CACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCC
GGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACAAAATACGTTT
TGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACA
GTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCT
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG"
```

```
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)
  dn_start = found.start(2) + 1
  dn_end   = found.end(2)

  print( whole , up , up_start, up_end , down , dn_start , dn_end , sep="\t" )
```

> 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
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCTTCTA
A TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA  98  148
CCGGTTTCCAAAGACAGTCTTCTAA 154 179
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCTTCTA
A 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 is 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.

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 are both 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 new string that is returned 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

Let' Try It

1. How would you use regular expressions to find all occurrences of 'ATG' and replace with '-M-' in this sequence 'GCAGAGGTGATGGACTCCGTAATGGCCAAATGACACGT'?

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)`

Helpful Regex tools

There are a lot of online tools for actually seeing what is happening in your regular expression. Search for `Python Regular Expression Tester`

- regex101

- pyregex
- pythex

Link to Python 7 Problem Set

Python 8

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 the full length of one sequence:

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

> Use the outermost index to access each sequence

Retrieve the nucleotide at a particular position in a sequence.

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

> Use the outermost 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 4th column for each sequence.

Lists of dictionaries

You can nest dictionaries in lists as well:

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

To retrieve just one gene's nucleotide composition

```
>>> genes['gene1']['nt_comp']
{'C': 2, 'G': 1, 'A': 1, 'T': 2}
```

Alter one gene's nucleotide count with `=` assignment operator:

```
>>> genes['gene1']['nt_comp']
{'C': 2, 'G': 1, 'A': 1, 'T': 2}
>>>
>>> genes['gene1']['nt_comp']['T']=6
>>> genes['gene1']['nt_comp']
{'C': 2, 'G': 1, 'A': 1, 'T': 6}
```

Alter one gene's nucleotide count with `+=` assignment operator:

```
>>> genes['gene1']['nt_comp']
{'C': 2, 'G': 1, 'A': 1, 'T': 6}
>>>
>>> genes['gene1']['nt_comp']['A']+=1
>>>
>>> genes['gene1']['nt_comp']
{'C': 2, 'G': 1, 'A': 2, 'T': 6}
>>>
>>>
```

To retrieve the A composition of every gene use a for loop.

```
>>> for gene in sorted(genes):
...     A_comp = genes[gene]['nt_comp']['A']
...     print(gene+":","As=", A_comp)
...
gene1: As= 2
gene2: As= 3
```

Building Complex Datastructures

Below is an example of building a list with a mixed collection of value types. Remember that all elements inside a list or dictionary should be the same type. In other words, the values in a list should all be lists or dictonaries or scalar values. This allows you to loop over the data structure.

The dictionary which is a list value has a key that has a dictionary as a value.

```
[{'gene1' : {'sequence' : [1, 2, 3], [4, 5, 6], [7,8,9]]
```

Just spaced differently:

```
[
   [1, 2, 3],
   [4, 5, 6],
   {
       'key': 'value',
       'key2':
            {
                'something_new': 'Yay'
            }
    }
 ]
```

Building this data structure in the interpreter:

```
>>> new_data = []
>>> new_data
[]
>>> new_data.append([1,2,3])
>>> new_data
[[1, 2, 3]]
>>> new_data[0]
[1, 2, 3]
>>> new_data.append([4,5,6])
>>> new_data
[[1, 2, 3], [4, 5, 6]]
>>> new_data[1]
[4, 5, 6]
>>> new_data[1][2]
6
>>> new_data.append({})
```

```
>>> new_data
[[1, 2, 3], [4, 5, 6], {}]
>>> new_data[2]['key']='value'
>>> new_data
[[1, 2, 3], [4, 5, 6], {'key': 'value'}]
>>> new_data[2]['key2']={}
>>> new_data
[[1, 2, 3], [4, 5, 6], {'key2': {}, 'key': 'value'}]
>>> new_data[2]['key2']['something_new']='Yay'
>>> new_data
[[1, 2, 3], [4, 5, 6], {'key2': {'something_new': 'Yay'}, 'key': 'value'}]
>>>
```

Same example in a script file: Building Complex Datastructures

Course T-shirt Organization and Counting

We have a spreadsheet of everyone's style, size, color. We want to know how many of each unique combination of style-size-color we need to order

shirts.txt

```
mens  small heather seafoam
womens  medium  Heather Purple
womens  medium  berry
mens  medium  heather coral silk
womens  Small Kiwi
Mens  large Graphite Heather
mens  large sport grey
mens  small Carolina Blue
```

We want something like this:

```
womens  small antique heliconia 2
womens  xs  heather orange  1
womens  medium  kiwi  2
womens  medium  royal heather 1
```

shirts.py

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

shirts = {}
```

```
with open("shirts.txt","r") as file_object:
  for line in file_object:
    line = line.rstrip()
    [style, size, color] = line.split("\t")
    style = style.lower()
    size = size.lower()
    color = color.lower()
    if style not in shirts:
        shirts[style] = {}
    if size not in shirts[style]:
        shirts[style][size] = {}
    if color not in shirts[style][size]:
        shirts[style][size][color] = 0

    shirts[style][size][color] += 1

for style in shirts:
  for size in shirts[style]:
    for color in shirts[style][size]:
      count = shirts[style][size][color]
      print(style,size,color,count,sep="\t")
```

Output:

```
sro$ python3 shirts.py
mens  small heather maroon  1
mens  small royal blue  1
mens  small olive 1
mens  large graphite heather  1
womens  medium  heather purple  3
womens  medium  berry 2
womens  medium  royal heather 1
womens  medium  kiwi  2
...
```

This is what the data structure we just built looks likes

```
{
  'mens':
    {
      'small':
```

```
        {
           'heather seafoam': 1,
           'carolina blue': 1,
           'cornsilk': 1,
           'dark heather': 1,
           'heather maroon': 1,
           'royal blue': 1,
           'olive': 1
        },
      'large':
        {
          'graphite heather': 1,
          'sport grey': 1,
          'heather purple': 1,
          'heather coral silk': 1,
          'heather irish': 1,
          'heather royal': 1,
          'carolina blue': 1
        },
      'medium':
        {
          'heather coral silk': 1,
          'heather royal': 2,
          'heather galapagos blue': 1,
          'heather forest': 1,
          'gold': 1,
          'heather military green': 1,
          'dark heather': 1,
          'carolina blue': 1,
          'iris': 1
        },
      'xs':
        {
          'white': 1
        },
      'xl':
        {
          'heather cardinal': 1,
          'indigo blue': 1
        }
          },
  'womens':
    {
```

```
      'medium':
        {
          'heather purple': 3,
          'berry': 2,
          'royal heather': 1,
          'kiwi': 2,
          'carolina blue': 1
        },
      'small':
        {
          'kiwi': 1,
          'berry': 1,
          'antique heliconia': 2
        },
      'large':
        {
          'kiwi': 1
        },
      'xs':
        {
          'heather orange': 1
        }
    },
  'child':
    {
      '4t':
        {
          'green': 2
        },
      '3t':
        {
          'pink': 1
        },
      '2t':
        {
          'orange': 1
        },
      '6t':
        {
          'pink': 1
        }
    }
}
```

There are also specific data table and frame handling libraries like Pandas.

Here is a intro to data structures in Panda.

Here is a very nice interactive tutorial

Link to Python 8 Problem Set

Python 9

Exceptions

There are a few different types of errors when coding. Syntax errors, logic errors, and exceptions. You have probably encountered all three. Syntax and logic errors are issues you need to deal with while coding. An exception is a special type of error that can be informative and used to write code to respond to this type of error. This is especially relavent when dealing with user input. What if they don't give you any, or it is the wrong kind of input. We want our code to be able to detect these types of errors and respond accordingly.

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

import sys
file = sys.argv[1]

print("User provided file:" , file)
```

> This code takes user provided input and prints it

Run it.

```
$ python scripts/exceptions.py test.txt
User provided file: test.txt
```

What happens if the user does not provide any input and we try to print it?

```
$ python scripts/exceptions.py
Traceback (most recent call last):
  File "scripts/exceptions.py", line 4, in <module>
    file = sys.argv[1]
IndexError: list index out of range
```

> We get an **IndexError** exception, which is raised when an index is not found in a sequence.

We have already seen quite a few exceptions throughout the lecture notes, here are some:

- ValueError: math domain error
- AttributeError: 'list' object has no attribute 'rstrip'
- SyntaxError: EOL while scanning string literal
- NameError: name 'GGTCTAC' is not defined
- SyntaxError: Missing parentheses in call to 'print'
- AttributeError: 'int' object has no attribute 'lower'
- IndexError: list assignment index out of range
- NameError: name 'HDAC' is not defined

Link to Python Documentation of built in types of exceptions

We can use the exception to our advantage to help the people who are running the script. We can use a try/except condition like an if/else block to look for exceptions and to execute specific code if we **do not have** an exception and do something different if we **do have** an exception.

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

file = ''
try:
  file = sys.argv[1]
  print("User provided file:" , file)
except:
  print("Please provide a file name")
```

> We need to "try" to get a user provided argument. If we are successful then we can print it out. If we try and fail, we execute the code in the except portion of our try/except and print that we need a file name.

Let's run it WITH user input

```
$ python3 scripts/exceptions_try.py test.txt
User provided file: test.txt
```

> It runs as expected

Let's run it WITHOUT user input

```
$ python scripts/exceptions_try.py
Please provide a file name
```

> Yeah, the user is informed that they need to provide a file name to the script

What if the user provides input but it is not a valid file or the path is incorrect? Or if you want to check to see if the user provided input as well as if it can open the input.

We can add multiple exception tests, like if/elif block. Each except statement can specify what kind of exception it is waiting to recieve. If that kind of exception occures, that block of code will be executed.

```
import sys

file = ''
try:
  file = sys.argv[1]
  print("User provided file name:" , file)
  FASTA = open(file, "r")
  for line in FASTA:
    line = line.rstrip()
    print(line)
except IndexError:
  print("Please provide a file name")
except IOError:
  print("Can't find file:" , file)
```

> Here we test for an IndexError: Raised when an index is not found in a sequence.
> The IndexError occurs when we try to access a list element that does not exists.
> And we test for a IOError: Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
> The IOError happens when we try to access a file that does not exist.

Let's run it with a file that does not exist.

```
$ python scripts/exceptions_try_files.py test.txt
User provided file name: test.txt
Can't find file: test.txt
```

> This informs the user that they did provide input but that the file listed can not be found.

Let's run it with no input

```
$ python scripts/exceptions_try_files.py
Please provide a file name
```

> This informs the user that they need to provide a file.

try/except/else/finally

Lets summarize what we have covered and add on `else` and `finally`.

```
try:
  # try block is executed until an exception is raised
except _ExceptionType_:
  # if there is an exception of "ExceptionType" this block will be executed
  # there can be more than one except block, just like an elif
except:
  # if there are any exceptions that are not of "ExceptionType" this except
block will be executed
else:
  # the else block is executed after the try block has been completed, which
means there were no exceptions raised
finally:
  # the finally block is executed if exceptions are or are not raised (no matter
what happens)
```

Getting more information about an exception

Some exceptions can be thrown for multiple reasons, for example, ErrorIO will occur if the file does not exist as well as if you don't have permissions to read it. We can get more information by viewing the contents of our Exception Object. Yes, an exception is an object too! The system errors get stored in the exception object. To access the object use `as` and supply a variable name, like 'ex'

```
file = ''
try:
  file = sys.argv[1]
  print("User provided file name:" , file)
  FASTA = open(file, "r")
  for line in FASTA:
    line = line.rstrip()
    print(line)
except IndexError:
  print("Please provide a file name")
except IOError as ex:
  print("Can't find file:" , file , ': ' , ex.strerror  )
```

> Here we added `except IOError as ex` and now we can get the 'strerror' message from ex.

Run it.

```
$ python scripts/exceptions_try_files_as.py  test.txt
User provided file name: test.txt
Can't find file: test.txt :  No such file or directory
```

> Now we know that this file name or path is not valid

Raising an Exception

We can call or raise exceptions too!! This is accomplished by using a `raise` statement.

1. First, create a new Exception Object, i.e., `ValueError()`
2. Use the Exception Object in a Raise statment `raise ValueError('your message')`

Let's raise an exception if the file name does not end in 'fa'

```
import sys

file = ''
try:
  file = sys.argv[1]
  print("User provided file name:" , file)
  if not file.endswith('.fa'):
    raise ValueError("Not a FASTA file")
  FASTA = open(file, "r")
  for line in FASTA:
    print(line)
```

```
except IndexError:
  print("Please provide a file name")
except IOError as ex:
  print("Can't find file:" , file , ': ' , ex.strerror  )
```

> Here we raise a known exception, 'ValueError', if the file does not end with (uses `endswith()` method).

Let's run it.

```
$ python scripts/exceptions_try_files_raise.py test.txt
User provided file name: test.txt
Traceback (most recent call last):
  File "scripts/exceptions_try_files_raise.py", line 10, in <module>
    raise ValueError("Not a FASTA file")
ValueError: Not a FASTA file
```

> Our exception get's raised, now lets do something with it.

```
import sys

file = ''
try:
  file = sys.argv[1]
  print("User provided file name:" , file)
  if not file.endswith('.fa'):
    raise ValueError("Not a FASTA file")
  FASTA = open(file, "r")
  for line in FASTA:
    print(line)
except IndexError:
  print("Please provide a file name")
except ValueError:
  print("File needs to be a FASTA file and end with .fa")
except IOError as ex:
  print("Can't find file:" , file , ': ' , ex.strerror  )
```

> Here we created an exception to catch any ValueError

Let's Run it.

```
$ python scripts/exceptions_try_files_raise_value.py test.txt
User provided file name: test.txt
File needs to be a FASTA file and end with .fa
```

> We get a great error message now.

But what if there is another ValueError, how can we tell if has anything to do with the FASTA file extension or not? Answer: the message will be different.

Creating Custom Exceptions

We can create our own custom exception. We will need to create a new class of exception. Below is the syntax to do this.

```
import sys

class NotFASTAError(Exception):
  pass


file = ''
try:
  file = sys.argv[1]
  print("User provided file name:" , file)
  if not file.endswith('.fa'):
    raise NotFASTAError("Not a FASTA file")
  FASTA = open(file, "r")
  for line in FASTA:
    print(line)
except IndexError:
  print("Please provide a file name")
except NotFASTAError:
  print("File needs to be a FASTA file and end with .fa")
except IOError as ex:
  print("Can't find file:" , file , ': ' , ex.strerror  )
```

> Here we created a new class of exception called 'NotFASTAError'. Then we raised this new exception.

Let's Run it.

```
$ python scripts/exceptions_try_files_raise_try.py test.txt
User provided file name: test.txt
File needs to be a FASTA file and end with .fa
```

> Our new class of exception, NotFASTAError, works just like the built in exceptions.

Link to Python 9 Problem Set

Python 10

Functions

Functions consist of several lines of code that do something useful and that you want to run more than once. There are built-in functions in python. You can also write your own. You also give your function a name so you can refer to it in your code. This avoids copying and pasting the same 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 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 give your functions names 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, need 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 the name should describe the data contained. The name needs to be consistent within your 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='N'):   # 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.

Lambda expressions

Lambda expressions can be used to define simple (one-line) 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:

```
def get_first_codon(dna):
  return dna[0:3]

print(get_first_codon('ATGTTT'))
```

> This will print `ATG`

Here is the same function written as a lambda

```
get_first_codon = lambda dna : dna[0:3]
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

Almost all python variables are global. This means they are available everywhere in your code. 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')
  x = 30
  y = 10
  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
```

The most important exception to variables being global is that variables that are defined in **functions** are **local** i.e. they only exist inside their function. 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 the example below 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   # also local
  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:

```
$ python3 scope_w_function.py
After def
x= 100
y= 100
Inside def
x = 5
y =  5
After function call
x= 100
y= 100
```

> 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`.

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

def set_global_variable():
  global greeting  # make greeting global
  greeting = "I say hello"


greeting = 'Good morning'
print('Before function call')
print('greeting =',greeting)

#make call to function
set_global_variable()
print('After function call')
print('greeting =',greeting)
```

Let's look at the output

```
$ python3 scripts/scope_global.py
Before function call
greeting = Good morning
After function call
greeting = I say hello
```

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

By creating new local variables inside function definitions, python stops variables with the same name from over-writing each other by mistake.

Modules

Python comes with some core functions and methods. There are many useful modules that you will want to use. `import` is the statement for telling your script you want to use code in a module. As we've already seen with regular expresions, you can bring in code that handles regular expressions with `import re`

Getting information about modules with `pydoc`

How do you find out information about a module? Python has help pages built into the command line, like `man` we met earlier in the unix lecture. Online information may be more up to date. Search at https://docs.python.org/3.6/. But if you don't have internet access, you can always use `pydoc`. To find out about the `re` module, type `pydoc re` on the command line. The last line in the output tells you where the python module is actually installed.

```
% pydoc re
Help on module re:

NAME
    re - Support for regular expressions (RE).

MODULE REFERENCE
    https://docs.python.org/3.6/library/re

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    This module provides regular expression matching operations similar to
    those found in Perl.  It supports both 8-bit and Unicode strings; both
    the pattern and the strings being processed can contain null bytes and
    characters outside the US ASCII range.

    Regular expressions can contain both special and ordinary characters.
    Most ordinary characters, like "A", "a", or "0", are the simplest
    regular expressions; they simply match themselves.  You can
    concatenate ordinary characters, so last matches the string 'last'.
...
FILE
```

```
    /anaconda3/lib/python3.6/glob.py
```

Here are some of the most common and useful modules, along with their methods and objects. It's a lightning tour.

os.path

`os.path` has common utilities for working file paths (filenames and directories). A path is either a relative or absolute list of directories (often ending with a filename) that tells you where to find a file or directory.

function	description
os.path.basename(path)	what's the last element of the path? Note `/home/tmp/` returns `''`, rather than `tmp`
os.path.dirname(path)	what's the directory the file is in?
os.path.exists(path)	does the path exist?
os.path.getsize(path)	returns path (file) size in bytes or error
os.path.isfile(path)	does the path point to a file?
os.path.isdir(path)	does the path point to a directory?
os.path.splitext(path)	splits before and after the file extension (e.g. '.txt')

os.system

Replaced by subprocess.

subprocess

This is the current module for running command lines from python scripts

```
import subprocess
subprocess.run(["ls","-l"])  # same as running ls -l on the command line
```

more complex than `os.system()`. You need to specify where input and output go. Let's look at this in some more detail.

Capturing output from a shell pipeline

Let's say we want to find all the files that have user amanda (or in the filename)

`ls -l | grep amanda`

becomes this 'shortcut' which will capture the output of the two unix commands in the variable `output`

```
import subprocess
output = subprocess.check_output('ls -l | grep amanda', shell = True)
```

This is better than alternatives with `subprocess.run()`. This is equivalent to the unix backtick quoted string.

`output` contains a bytes object (more or less a string of ASCII character encodings)

```
b'-rw-r--r--  1 amanda  staff       161952 Oct  2 18:03 test.subreads.fa\n-rw-r-
-r--  1 amanda  staff          126 Oct  2 13:23 test.txt\n'
```

You can covert by decoding the bytes object into a string

```
>>>output.decode('utf-8')
'-rw-r--r--  1 amanda  staff       161952 Oct  2 18:03 test.subreads.fa\n-rw-r--
r--  1 amanda  staff          126 Oct  2 13:23 test.txt\n'
```

Capturing output the long way (for a single command)

Let's assume that `ls -l` generates some output something like this

```
total 112
-rw-r--r--  1 amanda  staff           69 Jun 14 17:41 data.cfg
-rw-r--r--  1 amanda  staff       161952 Oct  2 18:03 test.subreads.fa
-rw-r--r--  1 amanda  staff          126 Oct  2 13:23 test.txt
```

How do we run `ls -l` in Python and capture the output (stdout)?

```
import subprocess
rtn = subprocess.run(['ls','-l'], stdout=subprocess.PIPE )  # specify you want
to capture STDOUT
bytes = rtn.stdout
stdout = bytes.decode('utf-8')
# something like
lines = stdout.splitlines()
```

`lines` now contains elements from every line of the `ls -l` output, including the header line, which is not a file

```
>>> lines[0]
'total 112'
>>> lines[1]
'-rw-r--r--  1 amanda  staff     69 Jun 14 17:41 data.cfg'
```

Check the exit status of a command

To run a command and check the exit status (really to check the exit status was ok or zero), use

```
oops = subprocess.check_call(['ls', '-l'])
# or, simpler...
oops = subprocess.check_call('ls -l', shell=True)
```

Run a command with that redirects stdout to a file using python subprocess

You can't write `ls -l > listing.txt` to redirect stdout in the subprocess method, so use this instead

```
 tmp_file = 'listing.txt'
 with open(tmp_file,'w') as ofh:
     oops = subprocess.check_call(['ls', '-l'], stdout=ofh )
```

sys

A couple of useful variables for beginners. Many more advanced system parameters and settings that we are not covering here.

function	description
sys.argv	list of command line parameters
sys.path	where Python should look for modules

re

See notes on regular expressions

collections

Better lists etc.

`from collections import deque`

copy

`copy.copy()`

and

`copy.deepcopy()`

Link to more info for more on deep vs shallow copying

math

function	description
math.exp()	e**x
math.log2()	log base 2
math.log10()	log base 10
math.sqrt()	square root
math.sin()	sine
math.pi(), math.e()	constants
etc	

see also numpy

random

Random numbers generated by computers are not truly random, so python calls these pseudo-random.

example	description
random.seed(1)	set starting seed for random sequence to 1 to enable reproducibility
random.randrange(9)	integer between 0 and 8
random.randint(1,5)	integer between 1 and 5
random.random()	float between 0 and 1
random.uniform(1,2)	float between 1 and 2
random.choice(my_genes)	return a random element of the sequence

To get a random index from an element of `list` use `i=random.randrange(len(list))`

statistics

Typical statistical quantities

example	description
statistics.mean([1,2,3,4,5])	mean or average
statistics.median([2,3,4,5])	median = 3.5
statistics.stdev([1,2,3,4,5])	standard deviation of sample (square root of sample variance)
statistics.pstdev([1,2,3,4,5])q	estimate of population standard deviation

glob

Does unix-like wildcard file path expansion.

```
>>> import glob
>>> glob.glob('pdfs/*.pdf')
['pdfs/python1.pdf', 'pdfs/python2.pdf', 'pdfs/python3.pdf', 'pdfs/python4.pdf',
'pdfs/python6.pdf', 'pdfs/python8.pdf', 'pdfs/unix1.pdf', 'pdfs/unix2.pdf']
>>> fasta_files = glob.glob('sequences/*.fa')
>>>
```

argparse

Great (if quite complicated) tool for parsing command line arguments and automatically generating help messages for scripts (very handy!). Here's a simple script that explains a little of what it does.

```
#!/usr/bin/env python3
import argparse
parser = argparse.ArgumentParser(description="A test program that reads in some
number of lines from an input file. The output can be screen or an output file")
# we want the first argument to be the filename
parser.add_argument("file", help="path to input fasta filename")
# second argument will be line number
# default type is string, need to specify if expecting an int
parser.add_argument("lines", type=int, help ="how many lines to print")
# optional outfile argument specified with -o or --out
parser.add_argument("-o","--outfile", help = "optional: supply output filename,
otherwise write to screen", dest = 'out')
args = parser.parse_args()
```

```
# arguments appear in args
filename = args.file
lines = args.lines
if args.out:
  print("writing output to", args.out)
```

With this module, -h help comes for free. --outfile type arguments are optional unless you write 'required=True' like this

```
parser.add_argument('-f', "-fasta", required=True, help='Output fasta filename',
dest='outfile')
```

Many more modules that do many things

time, HTML, XML, email, CGI, sockets, audio, GUIs with Tk, debugging, testing, unix utils

Also, non-core: BioPython for bioinformatics, Numpy for mathematics, statistics, pandas for data, scikitlearn for machine learning.

Link to Python 10 Problem Set

Python 11

Classes

The advantages of writing classes and writing functions are very similar.

When we write functions we group core Python functions and methods to create a unique collection statements that occur in a specific order.

These new functions make our code easier to read and to write, especially if you will use the function many times.

A conceptual difference between a function and a class is that a function usually does one thing, while a class will do many related things to help solve a problem.

What is a class really, what does it do? A class doesn't really do anything except for setting a list of rules for creating a new custom object. Every time you use the class you are creating an instance of a type of object.

You have been using classes to create objects

You have already been using classes to create objects. Here we are using the `open` function to create two instances of a file object. One instance holds information about a FASTA file while the other holds information about a GFF file.

```
fa_input = open("somedata.fa")
gff_input = open("somedata.gff")
```

attributes and methods

Classes create objects, these objects will have attributes and methods associated with them.

methods

Methods are functions which belong to objects of a particular class.

attributes

Attributes are variables that are associated with an object of a particular class.

Creating a Class

Defining a class is straightforward.

The first step is to decide what attributes and what methods it will have.

Create a DNARecord Class.

When we create a class, we are really setting up a series of rules that a DNARecord object must follow.

DNARecord Rules:

1. DNARecord must have a sequence [attribute]

2. DNARecord must have a name [attribute]
3. DNARecord must have an organism [attribute]
4. DNARecord will be able to calculate AT content [method]
5. DNARecord will be able to calculate the reverse complement [method]

Here is the first, but not final draft of our class. We will go through each section of this code below:

```
class DNARecord(object):
  # define class attributes
  sequence = 'ACGTAGCTGACGATC'
  gene_name = 'ABC1'
  species_name = 'Drosophila melanogaster'

  # define methods
  def reverse_complement(self):
    replacement1 = self.sequence.replace('A', 't')
    replacement2 = replacement1.replace('T', 'a')
    replacement3 = replacement2.replace('C', 'g')
    replacement4 = replacement3.replace('G', 'c')
    reverse_comp = replacement4[::-1]
    return reverse_comp.upper()

  def get_AT(self):
    length = len(self.sequence)
    a_count = self.sequence.count('A')
    t_count = self.sequence.count('T')
    at_content = (a_count + t_count) / length
    return at_content

## create a new DNARecord Object
dna_rec_obj = DNARecord()

## Use New DNARecord object
print('Created a record for ' + dna_rec_obj.gene_name + ' from ' +
dna_rec_obj.species_name)
print('AT is ' + str(dna_rec_obj.get_AT()))
print('complement is ' + dna_rec_obj.reverse_complement())
```

Now let's go through each section:

We start with the keyword `class`, followed by the name of our class `DNARecord` with the name of the base class in parentheses `object`.

```
class DNARecord(object):
```

Then we define class attributes. These are variables with data that belongs to the class, and therefore to any object that is created using this class

```
  # define class attributes
  sequence = 'ACGTAGCTGACGATC'
  gene_name = 'ABC1'
  species_name = 'Drosophila melanogaster'
```

Next, we define our class methods:

```
  # define methods
  def reverse_complement(self):
    replacement1 = self.sequence.replace('A', 't')
    replacement2 = replacement1.replace('T', 'a')
    replacement3 = replacement2.replace('C', 'g')
    replacement4 = replacement3.replace('G', 'c')
    reverse_comp = replacement4[::-1]
    return reverse_comp.upper()

  def get_AT(self):
    length = len(self.sequence)
    a_count = self.sequence.count('A')
    t_count = self.sequence.count('T')
    at_content = (a_count + t_count) / length
    return at_content
```

The methods are using an argument called `self`, i.e., `length = len(self.sequence)`. This is a special variable that you use inside a class. With it you can access all the data that is contained inside the object when it is created.

Use `self.attribute` format to retrieve the value of variables created within the class. Here we use `self.sequence` to retrieve the information stored in our attribute named `sequence`.

```
replacement1 = self.sequence.replace('A', 't')
```

Creating a DNARecord Object

The above class is a set of rules that need to be followed when creating a new DNARecord object. Now let's create a new DNARecord object:

```
  dna_rec_obj = DNARecord()
```

`dna_rec_obj` is our new DNARecord object that was creating using the rules we put into place in the class definition.

Retrieving attribute values

Now that a new DNARecord object has been created, and assigned to the variable `dna_rec_obj`, we can access its attributes using the following format, `object.attribute_name`

To get the gene name of the object we created, we simply write `dna_rec_obj.gene_name`.

This is possible because within our class definition we create a `gene_name` variable.

Let's try it:

```
>>> dna_rec_obj.gene_name
'ABC1'
>>> dna_rec_obj.sequence
'ACGTAGCTGACGATC'
```

Using class methods

To call a method associated with our new object, we use a similar format `object.method_name`.

So to call the `get_AT()` method, we would use `dna_rec_obj.get_AT()`. This should look familiar, you have done used class methods over and over again: `some_string.count('A')`

Let's try it with our `dna_rec_obj`:

```
>>> dna_rec_obj.sequence
'ACGTAGCTGACGATC'
>>> dna_rec_obj.get_AT()
0.4666666666666667
```

Now let's use the `reverse_complement()` method

```
>>> dna_rec_obj.sequence
'ACGTAGCTGACGATC'
>>> dna_rec_obj.reverse_complement()
GATCGTCAGCTACGT
```

Wow!! Getting the reverse complement in one line is pretty nice!

Getting data into a new instance of our class

Great!!!

We can now create a DNARecord object and retrieve the object attributes and use the cool methods we created.

But..... It always contains the same gene_name, sequence, and species information 😟

Let's make our class more generic, or in other words, make it so that a user can provide a new gene name, gene sequence, and source organism everytime a DNARecord object is created.

__init__

To do this we need to add an `__init__` function to our Object Rules, or Class.

The `init` function will automatically get called when you create an object.

It contains specific instructions for creating a new DNARecord Object.

It specifies how many pieces of data we want to collect from the creator of a DNARecord object to use within a DNARecord object.

Below our **__init__** instructions indicate that we want to create object attributes called `sequence`, `gene_name`, and `species_name` and to set them with the values provided as arguments when the object was created.

Here is our new class definition and new object creation when using the **__init__** function:

```
#!/usr/bin/env python3
class DNARecord(object):

  # define class attributes
  def __init__(self, sequence, gene_name, species_name): ## note that '__init__'
is wrapped with two underscores
    #sequence = 'ACGTAGCTGACGATC'
    #gene_name = 'ABC1'
    #species_name = 'Drosophila melanogaster'
    self.sequence = sequence
    self.gene_name = gene_name
    self.species_name = species_name

  # define methods
  def reverse_complement(self):
    replacement1 = self.sequence.replace('A', 't')
    replacement2 = replacement1.replace('T', 'a')
    replacement3 = replacement2.replace('C', 'g')
    replacement4 = replacement3.replace('G', 'c')
    reverse_comp = replacement4[::-1]
    return reverse_comp.upper()

  def get_AT(self):
    length = len(self.sequence)
    a_count = self.sequence.count('A')
    t_count = self.sequence.count('T')
    at_content = (a_count + t_count) / length
    return at_content

## Create new DNARecord Objects with user defined data
dna_rec_obj_1 = DNARecord('ACTGATCGTTACGTACGAGT', 'ABC1', 'Drosophila
melanogaster')
dna_rec_obj_2 = DNARecord('ATATATTATTATATTATA', 'COX1', 'Homo sapiens')

for d in [ dna_rec_obj_1, dna_rec_obj_2 ]:
  print('name:' , d.gene_name , ' ' , 'seq:' , d.sequence)
```

Output:

```
$ python3 dnaRecord_init.py
name: ABC1   seq: ACTGATCGTTACGTACGAGT
name: COX1   seq: ATATATTATTATATTATA
```

Now you can create as many DNASequence Objects as you like, each can contain information about a different sequence.

Link to Python 11 Problem Set
