```python
# 08-12-22 - Socratica - Sets in Python
# https://youtu.be/sBvaPopWOmQ

# Sets are useful when you are working with data and the order of the elements is irrelevant.
# use .add to add elements to a set.
# Duplicates will not be stored in a set. If you try to store a duplicate, it will ignore it the second time.

examples = set()
examples.add(42)
examples.add(False)
examples.add(3.14159)
examples.add('Thorium')

print(examples)

# Notice that you can add data of different types to the same set.
# Sets are different from lists and tuples in that the order does not matter in a set, and no duplicates allowed.

# Use the length function to find the length of a set:
print(len(examples))

# Use remove function to remove False from the set.
examples.remove(False)
print(examples)

# If you try to remove something not in the list, you get a key error.
# The discard method is a way around this error, and Python says nothing, no alert it is not there.
examples.discard(False)

# To prepopulate a set, do the following, but usually curly braces are used instead of this
# parentheses and bracket combo:

examples_02 = ([28, True, 'Helium', 'lovely', 55.343])
print(len(examples_02))

# As explained at: https://www.edlitera.com/blog/posts/python-parentheses#mcetoc_1fvg1o1m01d
# Sets are collections of mutable, unique, hashable values. When working with sets, you can treat them as
# dictionaries that contain only keys and no values. They are not used as often as dictionaries and are
# usually used as an easy way to remove duplicates from a collection. A set is created by entering values
# instead of pairs inside curly braces.
# NOTE: Curly braces go along with dictionaries, so using them in sets can be confusing.
# However, creating empty sets is not done by using curly braces. If you try to just leave nothing between
# the curly braces, Python will automatically create a dictionary. Therefore, to create an empty set you
```

```
# must invoke set().

# If we use the clear method, it will remove all elements from the set.
examples_02.clear()     # This removes everything from the set.

# EVALUATING THE UNION and INTERSECTION of two sets
# Union = the combination of ALL elements from the two sets, denoted by U.
# Intersection = the elements that are present in both sets, denoted by an upside-down U.

odds = set([1, 3, 5, 7, 9])
evens = set([2, 4, 6, 8, 10])
primes = set([2, 3, 5, 7])
composites = set([4, 6, 8, 9, 10])

print("odds.union(evens) = ", odds.union(evens))
print("odds.intersection(primes) = ", odds.intersection(primes))
print("evens.intersection(odds) = ", odds.intersection(evens))

# You can also use sets and ask questions like "2 in primes", and you will get True or False
# By typing "dir('name of set') you can get a list of all the different methods you can call
# on the class Set.

```

```python
# 08-12-22
# Lists - Socratica - https://www.socratica.com/lesson/lists

# Lists make it easy to work with ordered data, elements that belong in a specific sequence
# Two ways to create a list: use the list constructor examples = list() or just brackets
# examples = []

# A list can be created and populated simultaneously
primes = [2, 3, 5, 7, 11, 13]
primes.append(17)
primes.append(19)

print("This is the primes list: ", primes)
# This prints: This is the primes list:  [2, 3, 5, 7, 11, 13, 17, 19]
# The elements are kept in order.

# You can view single or multiple elements of a list (if you do not wish to view all) by indexing
# into the list using an elements index number, the count of which begins with [0]
primes[4]
print("This is element 4 of the list, which appears 5th: ", primes[4])
# This prints: This is element 4 of the list, which appears 5th:  11
# Indexing will wrap back around when you use negative numbers, thereby making the last number in a list
# both its index number and its negative index number which will always be [-1].
# Going beyond the scope of your list, positive or negative, will give you an index error.

# SLICING: retrieve a range of values from your list
primes[2:5]
print("This is primes[2:5]: ", primes[2:5])
# Slicing includes the value at the starting index but excludes the stopping index, so you have to add 1
# to get all the way to the end with slicing.

# Lists can contain much more than one type: integers, booleans, strings, floats, and even other lists
examples = [128, True, 'love', 1.732, [64, False], "and so on"]

# Lists can also contain duplicate values
dice_rolls = [4, 7, 2, 7, 12, 4, 7]
print("This is your dice roll list complete with the duplicate rolls: ", dice_rolls)
# This will print out: This is your dice roll list complete with the duplicate rolls:  [4, 7, 2, 7, 12, 4, 7]

# COMBINING LISTS, called concatenation, and leaves the original lists unchanged:
numbers = [1, 2, 3]
letters = ['a', 'b', 'c']
print("I shall now combine your lists: ", numbers + letters)
# This will print: I shall now combine your lists:  [1, 2, 3, 'a', 'b', 'c']

# To find out all of the functions you can use with lists, pass a list to the dir() function:
# dir(numbers) prints out a long list of functions as shown below. Typing help(numbers.reverse)
# will explain to  you how to use the function, as it will for all the following functions:
# ['__add__',
```

```
#  '__class__',
#  '__class_getitem__',
#  '__contains__',
#  '__delattr__',
#  '__delitem__',
#  '__dir__',
#  '__doc__',
#  '__eq__',
#  '__format__',
#  '__ge__',
#  '__getattribute__',
#  '__getitem__',
#  '__gt__',
#  '__hash__',
#  '__iadd__',
#  '__imul__',
#  '__init__',
#  '__init_subclass__',
#  '__iter__',
#  '__le__',
#  '__len__',
#  '__lt__',
#  '__mul__',
#  '__ne__',
#  '__new__',
#  '__reduce__',
#  '__reduce_ex__',
#  '__repr__',
#  '__reversed__',
#  '__rmul__',
#  '__setattr__',
#  '__setitem__',
#  '__sizeof__',
#  '__str__',
#  '__subclasshook__',
#  'append',
#  'clear',
#  'copy',
#  'count',
#  'extend',
#  'index',
#  'insert',
#  'pop',
#  'remove',
#  'reverse',
#  'sort']
```

```
# 08-16-22 - PyDoc - Socratica - https://youtu.be/URBSvqib0xw

# Documentation is how engieers desscribe their code in prose.
# PyDoc module is the tool with which you can share your documentation with other
# engineers.

# Metadocumentation = the documentation about the documentation

# To look at the documentation on any given object:
# In the terminal window, type: python -m pydoc name_of_module
# It will tell you how to use every single function in the module.

# You can also use the same method to look up the help info for a class, etc.
# EX: python -m pydoc tuple

# PRINTS:
# class tuple(object)
#  |  tuple(iterable=(), /)
#  |  Built-in immutable sequence.

# Pydoc is identical to the help function except you do not have to import
# a module in order to look at the documentation for things contained in it.

# You can use Pydoc to search all modules for a certain keyword:
# python -m pydoc -k ftp        -k tells it you are going to give it a keyword.

# PRINTS: (Every module that has anything to do with ftp, including 3rd party)
#
# ftplib - An FTP client class and some helper functions.
# numpy.fft.tests.test_helper - Test functions for fftpack.helper module
# pygame 2.1.2 (SDL 2.0.18, Python 3.10.4)
# Hello from the pygame community. https://www.pygame.org/contribute.html
# scipy.fftpack - ==========================================================
# scipy.fftpack.basic - Discrete Fourier Transforms - basic.py
# scipy.fftpack.convolve
# scipy.fftpack.helper
# scipy.fftpack.pseudo_diffs - Differential and pseudo-differential operators.
# scipy.fftpack.realtransforms - Real spectrum transforms (DCT, DST, MDCT)
# scipy.fftpack.setup
# scipy.fftpack.tests
# scipy.fftpack.tests.gen_fftw_ref
# scipy.fftpack.tests.gendata
# scipy.fftpack.tests.test_basic
# scipy.fftpack.tests.test_helper
# scipy.fftpack.tests.test_import - Test possibility of patching fftpack with pyfftw.
# scipy.fftpack.tests.test_pseudo_diffs
# scipy.fftpack.tests.test_real_transforms

# When we search pydoc for info on pydoc:

# pydoc - the Python documentation tool
# pydoc <name> ...
```

```
#     Show text documentation on something.  <name> may be the name of a
#     Python keyword, topic, function, module, or package, or a dotted
#     reference to a class or function within a module or module in a
#     package.  If <name> contains a '/', it is used as the path to a
#     Python source file to document. If name is 'keywords', 'topics',
#     or 'modules', a listing of these things is displayed.
#
# pydoc -k <keyword>
#     Search for a keyword in the synopsis lines of all available modules.
#
# pydoc -n <hostname>
#     Start an HTTP server with the given hostname (default: localhost).
#
# pydoc -p <port>
#     Start an HTTP server on the given port on the local machine.  Port
#     number 0 can be used to get an arbitrary unused port.
#
# pydoc -b
#     Start an HTTP server on an arbitrary unused port and open a web browser
#     to interactively browse documentation.  This option can be used in
#     combination with -n and/or -p.
#
# pydoc -w <name> ...
#     Write out the HTML documentation for a module to a file in the current
#     directory.  If <name> contains a '/', it is treated as a filename; if
#     it names a directory, documentation is written for all the contents.

# Calling pydoc with the -b option will find an available port and open the
# documentation in the browswer for you.

# python -m pydoc -b
# Here you can peruse TONS of modules and all sorts of documentation.
```

```
# 08-13-22 - TUPLES - SOCRATICA - https://youtu.be/NI26dqhs2Rk

# Tuples = the smaller, faster alternative to lists
# The difference between lists and tuples:

# A LIST contains a sequence of data, surrounded by square brackets
# LIST example:
prime_numbers = [2, 3, 5, 7, 11, 13, 17]

# A TUPLE contains a sequence of data surrounded by parentheses
# TUPLE example:
perfect_squares = (1, 4, 9, 16, 25, 36)

# Both can use the len function to display the number of elements:
print("# Primes: ", len(prime_numbers))
print("# Squares = ", len(perfect_squares))

# Both can be iterated over:
for prime in prime_numbers:
    print('Prime: ', prime)
for square in perfect_squares:
    print("Square: ", square)

# DIFFERENCES:
# To see the difference, we will print the methods available for the class LIST
print('List Methods')
print(dir(prime_numbers))
print(80 * '-')
print('Tuple Methods')
print(dir(perfect_squares))

# We get:
# List Methods
# ['_add_', '_class_', '_class_getitem_', '_contains_', '_delattr_', '_delitem_', '_dir_',
# '_doc_', '_eq_', '_format_', '_ge_', '_getattribute_', '_getitem_', '_gt_', '_hash_',
# '_iadd_', '_imul_', '_init_', '_init_subclass_', '_iter_', '_le_', '_len_', '_lt_',
# '_mul_', '_ne_', '_new_', '_reduce_', '_reduce_ex_', '_repr_', '_reversed_', '_rmul_',
# '_setattr_', '_setitem_', '_sizeof_', '_str_', '_subclasshook_', 'append', 'clear', 'copy',
# 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

# Tuple Methods
# ['_add_', '_class_', '_class_getitem_', '_contains_', '_delattr_', '_dir_', '_doc_',
# '_eq_', '_format_', '_ge_', '_getattribute_', '_getitem_', '_getnewargs_', '_gt_', '_hash_',
# '_init_', '_init_subclass_', '_iter_', '_le_', '_len_', '_lt_', '_mul_', '_ne_', '_new_',
# '_reduce_', '_reduce_ex_', '_repr_', '_rmul_', '_setattr_', '_sizeof_', '_str_',
# '_subclasshook_', 'count', 'index']

# Lists have more functions available to them, but they occupy more memory also.
# By importing sys and using the getsizeof function in sys, you can see how many bytes something uses
import sys

```

```python
list_ex = [1, 2, 3, 'a', 'b', 'c', True, 3.14159]
tuple_ex = (1, 2, 3, 'a', 'b', 'c', True, 3.14159)
print("Size of list: ", sys.getsizeof(list_ex))  # 120
print("Size of tuple: ", sys.getsizeof(tuple_ex))  # 104

# MORE DIFFERENCES
# Lists - you can add, remove, and change data
# Tuples - cannot be changed, immutable, allowing Python to optimize
# timeit module has function also called timeit. The first argument is a statement containing a command
# we would like to execute.
# Below, I have created a list of 5 integers and am going to run 1 million times.
import timeit

list_test = timeit.timeit(stmt="[1, 2, 3, 4, 5]", number=1000000)
tuple_test = timeit.timeit(stmt="(1, 2, 3, 4, 5)", number=1000000)

print("Time of list: ", list_test)  # Time of list:  0.05036733404267579
print("Time of tuple    : ", tuple_test)  # Time of tuple    :  0.010904083028435707

# WORKING WITH TUPLES:
# Tuples use parentheses, and you can make an empty one with empty parentheses.
empty_tuple = ()
test1 = ('a')
test2 = ('a', 'b')
test3 = ('a', 'b', 'c')
print(empty_tuple)  # ()
print(test1)        # a     <- test1 came back a string. Put a comma at the end to make a tuple with 1 element
print(test2)        # ('a', 'b')
print(test3)        # ('a', 'b', 'c')

empty_tuple = ()
test1 = ('a',)
test2 = ('a', 'b')
test3 = ('a', 'b', 'c')
print(empty_tuple)  # ()
print(test1)        # ('a',)    <- Now it is a tuple
print(test2)        # ('a', 'b')
print(test3)        # ('a', 'b', 'c')

# Alternative Construction of Tuples:
# You can leave out parentheses all together
test4 = 1,
test5 = 1, 2
test6 = 1, 2, 3

print(test4)        # (1,)
print(test5)        # (1, 2)
print(test6)        # (1, 2, 3)
print(type(test4))        # <class 'tuple'>
print(type(test5))        # <class 'tuple'>
```

```python
print(type(test6))        # <class 'tuple'>


# Tuples with one element:
# The reason for the above situation of the tuple with one element is because of "tuple assignment".
# Imagine working with a data set of people that contains 3 things about each:
# their age, country, and whether or not they know Python, taken from a survey and stored in a tuple.

# age, country, knows_python
survey = (27, "Vietnam", True)

# These can be accessed the same way list elements can:
age = survey[0]
country = survey[1]
knows_python = survey[2]

# Printing values to make sure this method is successful
print('Age: ', age)
print('Country: ', country)
print('Knows Python? ', knows_python)

# Prints:
# Age:  27
# Country:  Vietnam
# Knows Python?  True

# Now, add a second person to the survey:
survey2 = (21, 'Switzerland', False)

# TUPLE ASSIGNMENT: You can assign all elements to different variables in a tuple in a single line.
# Python unpacks all the variables and assigns them for you.
age, country, knows_python = survey2

print('Age: ', age)
print('Country: ', country)
print('Knows Python? ', knows_python)

# Tuple Assignment explains the need for the trailing comma when creating a tuple with just one element.
# According to the rules of tuple assignment, without the comma, Python will unpack and assign the variables
# rather than create a new, single-element tuple.

country = ("Australia")     # <- Unpacks and assigns Australia as the country for a survey person
print(country)

# VS

```

```python
country = ("Australia",)    # <- Creates the tuple country with the single element Australia and tells
print(country)              #    Python not to unpack it as a variable.

# Make sure the number of variables matches the number of elements in the tuple, or you get a ValueError.

a, b, c = (1, 2, 3, 4)  # <- Not enough variables to hold all of the values of the tuple.
x, y, z = (1, 2)        # ValueError: too many values to unpack (expected 3)

```

```
# 08-15-22 - SORTING in PYTHON - SOCRATICA - https://youtu.be/QtwhlHP_tqc

# SORTING ALPHABETICALLY:
# Alkaline Earth metals, currently sorted by atomic number:
earth_metals = ['Beryllium', 'Magnesium', 'Calcium', 'Strontium', 'Barium', 'Radium']

# By default, the sort method assumes you want the information sorted alphabetically ascending.
# So to sort this list alphabetically, we only need:
earth_metals.sort()
# And then print them.
print("Earth metals sorted alphabetically, ascending: ", earth_metals)
# OR
print(sorted(earth_metals))
# Trying print(earth_metals.sort()) prints None, because it changed the original list rather than
# returning anything.
# Prints:
# Earth metals sorted alphabetically, ascending:  ['Barium', 'Beryllium',
#                                                  'Calcium', 'Magnesium', 'Radium', 'Strontium']
# ['Barium', 'Beryllium', 'Calcium', 'Magnesium', 'Radium', 'Strontium']

# To put them in reverse order alphabetically:
earth_metals.sort(reverse = True)
print(earth_metals) # Prints: ['Strontium', 'Radium', 'Magnesium', 'Calcium', 'Beryllium', 'Barium']

# Now with a tuple, rather than a list:
# earth_metals_tuple = ('Beryllium', 'Magnesium', 'Calcium', 'Strontium', 'Barium', 'Radium')
# earth_metals_tuple.sort()
# print(earth_metals_tuple)
# Prints an error: AttributeError: 'tuple' object has no attribute 'sort'
# Tuples are immutable objects, and they cannot be changed. Sorting changes things.
# Sorting actually changes the object itself rather than making another that is sorted.

# help(list.sort)
# Help on method_descriptor:
# sort(self, /, *, key=None, reverse=False)     <- By default, reverse is set to False
#     Sort the list in ascending order and return None.     <- So it will sort ascending.
#     The sort is in-place (i.e. the list itself is modified) and stable (i.e. the
#     order of two equal elements is maintained).   <- In-place means Python does not create a 2nd list.
#     If a key function is given, apply it once to each list item and sort them,
#     ascending or descending, according to their function values.
#     The reverse flag can be set to sort in descending order.

# The key argument (first) for sort is a sorting function, which will be used to determine
# what values to sort by.

# The following list is the planets in the solar system, their radius, density, and average
# distance from the Sun in astronomical units, 1 = avg distance of Earth from Sun.

planets = [('Mercury', 2440, 5.43, 0.395),
           ('Venus', 6052, 5.24, 0.723),
```

```python
          ('Earth', 6378, 5.52, 1.000),
          ('Mars', 3396, 3.93, 1.530),
           ('Jupiter', 71492, 1.33, 5.210),
           ('Saturn', 60268, 0.69, 9.551),
           ('Uranus', 25559, 1.27, 19.213),
           ('Neptune', 24764, 1.64, 30.070)]

# Currently, the planets are sorted by their distance from the sun.
# We want to sort by their size / radii highest to lowest instead.
# We need to create a function to sort by, in this case, one that returns the second
# value in the tuple:

size = lambda planet: planet[1] # <- This will choose the second element in the tuple, index[1]
planets.sort(key=size, reverse = True) # <- passing in the function to sort by and reverse
                                   # to sort planets from largest to smallest.

print(planets)
# Prints: [('Jupiter', 71492, 1.33, 5.21), ('Saturn', 60268, 0.69, 9.551),
#       ('Uranus', 25559, 1.27, 19.213), ('Neptune', 24764, 1.64, 30.07),
#       ('Earth', 6378, 5.52, 1.0), ('Venus', 6052, 5.24, 0.723),
#       ('Mars', 3396, 3.93, 1.53), ('Mercury', 2440, 5.43, 0.395)]

# Now to sort by density:
density = lambda planet: planet[2]
planets.sort(key=density) # <- Going to print by default (ascending), not reverse

print(planets)
# Prints: [('Saturn', 60268, 0.69, 9.551), ('Uranus', 25559, 1.27, 19.213),
#        ('Jupiter', 71492, 1.33, 5.21), ('Neptune', 24764, 1.64, 30.07),
#        ('Mars', 3396, 3.93, 1.53), ('Venus', 6052, 5.24, 0.723),
#        ('Mercury', 2440, 5.43, 0.395), ('Earth', 6378, 5.52, 1.0)]


# What if you want to create a sorted copy of a list instead? Or sort a tuple?
# For this, we can use the SORTED method:

help(sorted)
# Help on built-in function sorted in module builtins:
# sorted(iterable, /, *, key=None, reverse=False)
#     Return a new list containing all items from the iterable in ascending order.
#     A custom key function can be supplied to customize the sort order, and the
#     reverse flag can be set to request the result in descending order.

# When calling SORTED, the first argument is a list or any iterable. Then a key
# or function to sort by, then a specification for reverse or  not.

earth_metals_02 = ['Beryllium', 'Magnesium', 'Calcium', 'Strontium', 'Barium', 'Radium']
sorted_earth_metals_02 = sorted(earth_metals_02)

print(sorted_earth_metals_02)
print(earth_metals_02)
# Prints: ['Barium', 'Beryllium', 'Calcium', 'Magnesium', 'Radium', 'Strontium']
```

```python
# Printed the metals in alphabetical order.
# But left the original list in its original atomic order:
# ['Beryllium', 'Magnesium', 'Calcium', 'Strontium', 'Barium', 'Radium']

# Tuple of first positive integers in random order
data = (7, 2, 5, 6, 1, 3, 9, 10, 4, 8)
# Tuples are immutable, so they do not have a sort method, since they cannot be changed.
# However, if you pass them to the sorted function:
print(sorted(data))
# Prints: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# Input was a tuple, but the output is a list. And the original tuple remains unaltered.

# SORTED can also sort strings character by character, capital letters coming first:
print(sorted("Alphabetical"))
# Prints: ['A', 'a', 'a', 'b', 'c', 'e', 'h', 'i', 'l', 'l', 'p', 't']
```

```python
# 08-12-22 - Socratica Python Videos - Notes
# https://youtu.be/NE97ylAnrz4

import math

# Functions enable you to use information a repeatable number of times without repeating yourself.

# Inside the parentheses, you write the inputs for the function, the arguments
# Pass tells Python to skip this code and move on.
# If you type the function without its parentheses, Python will tell you where in memory the function is stored
# rather than trying to run the function.

def f():
    pass

# Returns are optional

def ping():
    return 'ping!'

# Write a function that will return the value of the volume of a sphere when given the radius, based on the equation
# for calculating the volume of a sphere: V = 4/3(pi)(r^3) - Must import math module in order to use pi
# We use floats to get an accurate 4/3.

def volume_sphere(r):
    '''Returns the volume of a sphere when given its radius'''
    volume = (4.0/3.0) * math.pi * (r**3)
    return volume
volume_sphere(4)
print(volume)

# Because we give an argument when creating the function, r is a required argument when calling the function as well.

# Write a function that takes two arguments and computes the area of a triangle, a = 1/2(base x height)

def area_triangle(b, h):
    """Returns the area of a triangle when given the base and height measurements."""
    volume = 0.5 * b * h
    return volume

area_triangle(3, 6)
print(volume)

# KEYWORD ARGUMENTS:
# Write a function that converts a person's height from American units to centiments, given that
# 1 inch = 2.54cm and 1 foot = 12 inches.
```

```python
# The function will take two keyword arguments, feet and inches. We assign each a default value of 0.
# This is why Python also refers to keyword arguments as default arguments.

def standard_metric(feet = 0, inches = 0):
    """ Converts a person's height from standard American feet and inches to centimeters."""
    inches_to_centimeters = inches * 2.54
    feet_to_centimeters = feet * 12 * 2.54
    conversion = inches_to_centimeters + feet_to_centimeters
    return conversion
standard_metric(feet = 5, inches = 7)
print(conversion)

# TYPES of ARGUMENTS: Keyword (has = sign and a default value) and Required
# When you write a function and use BOTH types of arguments together, the keyword arguments must come last.
# Example:

def g(y, x = 0):
    return x+y

# You must provide the required argument y, but do not have to provide an x. If you do not provide an x, the
# default value assigned in the definition of the function will be used.
# To provide a value for a keyword argument, you must speficy it by its name:
# Required arguments are not given a name and are determined by their position.

g(5, x = 4)
```

```python
# 08-17-22: Socratica - Iterators - https://youtu.be/WR7mO_jYN9g

# Looping, every programmers favorite activity.
# Iterables and the itertools module:

# In Python, if you can loop over something in a for-loop, it is called an iterable.
# Iterables include any sequence that is ordered: lists, tuples, strings, and bytes

# LIST:
list_thing = ['CX32', 'GSOF', 'Emily', 'Franz', 'Rex']
for element in list_thing:
    print(element)

# TUPLE:
for element in ('Jose', 'Boh', 'Rusti'):
    print(element)

#STRING:
for letter in 'Socratica':
    print(letter)

# BYTES (ASCII codes for each letter):
for byte in b'Binary':
    print(byte)

# Non-iterables: digits of an integer, and an iterable must be constructed for that.
# Instead, you can iterate over the characters in a string version of a number.
# The following will convert each number of the integer into a character in a list.
c = 299792498
digits = [int(d) for d in str(c)]
# Now, we can loop over the digits:
for digit in digits:
    print(digit)

# What makes an object iterable?
# Iterables are containers that have two special methods that make them iterable:
# _iter_() and _next_().
# container._iter_() - returns an iterator object
# container._next_() - returns the next item from the collection
# Repeated calls to _next_() will go through items one item at a time until there
# is nothing left to iterate over, at which point a StopIterator Exception is raised.

# We will iterate a for-loop by calling the _iter_() and _next_() methods ourselves:
usernames = ('Rainer', 'Alfons', 'Flatsheep')
looper1 = usernames._iter_()      # <- This creates our iterator
print(type(looper1))    # PRINTS: <class 'tuple_iterator'>

print(looper1._next_())   # PRINTS: Rainer
print(looper1._next_())   # PRINTS: Alfons
print(looper1._next_())   # PRINTS: Flatsheep

# Another call would give us an error due to the StopIteration
```

```
# You can also use the iter and next functions without the underscores:

looper2 = iter(usernames)   # <- This creates our iterator
print(next(looper2))    # PRINTS: Rainer
print(next(looper2))    # PRINTS: Alfons
print(next(looper2))    # PRINTS: Flatsheep

# Again, another call gives us the StopIteration message.
# Now for a for-loop using these functions:
users = ['laust', 'LeoMoon', 'JennaSys', 'dgletts']
# As a conventional for-loop
for user in users:
    print(user)

# As a long-winded, typed-out for-loop
looper3 = iter(users)   # <- This creates our iterator
while True:             # <- Creates an infinite loop, only stopping
    try:                # when exception happens.
        user = next(looper3)
        print(user)
    except StopIteration:
        break
# The 7 lines above explain the mechanics of iterations with iterables.

# Now, we will create a class with iteration built in (a stock portfolio):

class Portfolio:            # <- constructor creates a dictionary to hold
    def _init_(self):       # number of shares in each asset.
        self.holdings = {}  # <- Key = ticker, value = number of shares.

    # buy method will increase the holdings in ticker by the specified
    # number of shares.
    def buy(self, ticker, shares):
        # If this is the first time purchasing this asset, we will use a
        # default value of zero shares.
        self.holdings[ticker] = self.holdings.get(ticker, 0) + shares

    # Next, we make a sell method for selling shares.
    # Buy and sell could be done in one method, postive integer for buy
    # and negative integer for sell.
    def sell(self, ticker, shares):
        self.holdings[ticker] = self.holdings.get(ticker, 0) - shares

    # Now, we want to be able to iterate over the holdings in a portfolio
    def _iter_(self):
        # Here, we only need to supply an iterator, and since our holdings
        # are in a dictionary, they are already iterable, so all we need
        # to do is return the iteration of that iterable.
        return iter(self.holdings.items())
        # The items() method returns a view object. The view object contains
        # the key-value pairs of the dictionary, as tuples in a list.
```

```python

# Now we can create a portfolio and invest in some imaginary companies:
p = Portfolio()     # <- instantiates a Portfolio object named 'p'
p.buy('ALPHA', 15)
p.buy('BETA', 9)
p.buy('GAMMA', 23)
p.buy('GAMMA', 20)

# Loop over portfolio and display holdings:
for ticker, shares in p:
    print(ticker, shares)

# PRINTS:
# ALPHA 15
# BETA 9
# GAMMA 43


#.............................................................................#
# ITERTOOLS MODULE: has three categories of functions
# Infinite iterators = if you do a for-loop using one of these, it will go on
    # forever until you have stopped the loop.
# A group of functions for common pre-processing on the collection of things
    # over which you are looping
# Combinatoric functions = make it easy to do calculations involving permutations
    # and combinations from a set.

# To illustrate itertools, we will construct a list of all possible hands in poker
import itertools

# possible number ranks in cards (2-10) along with jacks, queens, kings, and aces
ranks = list(range(2, 11)) + ['J', 'K', 'Q', 'A']
# This would give us a list composed of integers and strings
# So we will make all of them strings:
ranks = [str(rank) for rank in ranks]
print(ranks)

suits = ['Hearts', 'Clubs', 'Diamonds', 'Spades']

# List comprehension to combine ranks and suits:
deck = [card for card in itertools.product(ranks, suits)]

for index, card in enumerate(deck):
    print(1+index, card)

# Create a list of all the possible combinations of cards
hands = [hand for hand in itertools.combinations(deck, 5)]

print(f"The number of hands possible in poker is {len(hands)}")
```

```python
# 08-17-22 - EXCEPTIONS in PYTHON - SOCRATICA - https://youtu.be/nlCKrKGHSSk

# When Python encounters an error while running your code, it stops execution
# and raises an exception.
# An EXCEPTION is an object with a description of what went wrong and a
# TRACEBACK to where the problem occurred.
# There are TONS of different types of exceptions, but we will talk about
# the most common ones:

# Purposefully problematic code: SYNTAX ERRORS:

# for i in range(5)        <- Will raise a Syntax Error, without :
#    print('Hello, World!')

# Python will point exactly to where the syntax error occurred

# Other common exceptions:
# ZeroDivision - don't even try dividing by zero
# FileNotFound - if you refer to a file that the code cannot find
# TypeError - for using a type incorrectly
# ValueError - usually when dealing with mathematical operations.
# Use cmath module for complex numbers.
# There are many classes and subclasses of errors and exceptions
# Ex: LookupError has IndexError and KeyError as child errors

# ........................................................................... #

# The main way of dealing with errors is the TRY, EXCEPT, ELSE, FINALLY
# construction.
# You can have more than one exception clause, if necessary. So you can respond
# to different exceptions in different ways.
# In the TRY block, Python attempts to execute your code. If a problem occurs,
# it jumps to the first matching exception block.
# If no problem occurs, then after try, it skips all the excepts and goes
# to the else block.
# The FINALLY block will always execute.

# Write a function that reads the code of a binary file and returns the data.
# We will also measure the time required to do so.
import logging
import time

# Create a basic logger with debug level:
logging.basicConfig(filename='problems.log', level=logging.DEBUG)
logger = logging.getLogger()


def read_file_timed(path):
    """Return the contents of path and find out the time to do so."""
    start_time = time.time()
    try:
        f = open(path, mode='rb')
```

```
        data = f.read()
        return data
    # If the above is unable to execute, we will throw a customized error.
    except FileNotFoundError as err:
        logger.error(err)
        # raise will pass along the file not found error to user
        raise
    else:
        f.close()
    finally:
        stop_time = time.time()
        dt = stop_time - start_time

    logger.info("Time required for {file} = {time}".format(file=path, time=dt))
```

```
# 08-22-22 - GENERATORS - SOCRATICA - https://youtu.be/gMompY5MyPg

# Good for looping over large data that would otherwise crash your computer
# Good for going through seemingly infinite amounts of data

# A generator is a function that acts as an iterator. It generates the elements you
# loop over. It is like an on-demand iterable object.
# Typical iterators loop over data stored in memory, but generators save on memory.
# Generators use YIELD instead of return - temporarily passing control over to the
# code that is looping over the generator object's values until the generator runs
# out of yields.

def g():
    yield 1
    yield 2
    yield 3
print(g())
# PRINTS: <generator object g at 0x7faf181b7d10>
# It returns a generator object ^^^ rather than a number, and we can loop over it.

for x in g():
    print(x)
# PRINTS:
# 1
# 2
# 3

# Now for a function that yields each of the 26 characters of the English alphabet:
# String module gives access to commonly used sets of characters as strings
import string

def letters():
    for letter in string.ascii_lowercase:
        yield letter

for letter in letters():
    print(letter)
# PRINTS: the lower case alphabet one letter per line

# Generator function that yields all the prime numbers:

import itertools    # <- going to use the count(start, increment) function
def prime_numbers():
    # first prime is 2, all others negative. Handle 2 first:
    yield 2
    prime_cache = [2] # Cache of our prime numbers

    # Loop over all positive odd integers starting with 3
    for n in itertools.count(3, 2):
        is_prime = True     # Assuming n is prime

        # Check if n is divisible by any of the prime numbers in our cache
```

```
        for p in prime_cache:
            if n % p == 0:  # Thus it is not prime if so divisible
                is_prime = False
                break

        # Is it really prime?
        if is_prime:
            prime_cache.append(n)   # Add n to our cache
            yield n                 # yield n back as prime number

for p in prime_numbers():           # We can now loop over and print our primes.
    print(p)                        # Once p is over 100, we will stop looping
    if p > 100:                     # with this break statement, otherwise it
        break                       # will continue infinitely.


#  MORE COMPACT WAY TO MAKE GENERATOR: with a generator expression
# (similar to list comprehensions, but use parentheses instead of [])

squares = (z ** 2 for z in itertools.count(1))

for number in squares:
    print(number)

    if number > 500:
        squares.close() # Close method stops generator from generating more squares.

print(type(squares))
import sys
print(sys.getsizeof(squares))

# PRINTS:
# <class 'generator'>
# 104 (bytes)
# If we used a list comprehension, it would use an infinite number of bytes
```

```python
# 08-13-22 LISTS - Socratica - https://www.youtube.com/watch?v=XCcpzWs-CI4

# Used when you have key-value pairs of data, an input that is mapped to an output

# Example: collecting data for a social media post and start with collecting data for the post:
# user_id: 209
# message: D5 C5 E5 C4 G4
# language: English
# datetime = some date
# location = some coordinates

# dictionaries open with a curly brace and consist of key-value pairs separated by a colon,
# and if there are more than one pair of key-values, they are separated by a comma:
post = {'user_id': 209, 'message': 'D5 C5 E5 C4 G4', 'language': 'English',
        'datetime': 'some date', 'location': (44.590533, -104.715556)}

# Think of this dictionary with a map of 5 inputs (keys) and 5 outputs (values)
# This dictionary has multiple data types: an integer, 3 strings, and a tuple of floats
# You can also use the dict constructor to make dictionaries, since they are an instance
# of the dict class (In constructor, no quotes around key name, but yes quotes when adding:

# Question: Why when using the dict constructor do you not put message and language in
# quotes?

post_02 = dict(message='SS Cotopaxi', language='English')

# Add additional pieces of data by putting the key name in brackets and using = to assign
# a value.
post_02['user_id'] = 209
post_02['datetime'] = 'some date and time'

# To access information FROM a dictionary, also use these brackets:
print(post_02['user_id'])

# If you try to print information that is not in a dictionary, you will get a KeyError, which
# can be avoided by asking if it is in the dictionary first:
if 'location' in post_02:
    print(post_02['location'])
else:
    print('This post does not contain a location value.')

# You can also use the TRY-EXCEPT commands to avoid key error

try:
    print(post_02['location'])
except KeyError:
    print('This post does not contain a location value.')

# Dictionaries also have many class methods available, such as 'get'
# You can use the help function to find out what any of these methods does:
help(post_02.get)
# Prints: get(self, key, default=None, /)
```

```
#               Return the value for key if key is in the dictionary, else default.

# So we can attempt to get a location from post_02 and assign the default None if it has no location.
loc = post_02.get('location', None)
print(loc)

# It is common to iterate over all the key-value pairs in a dictionary. A good way to
# do this is to loop over all the keys and get the value for each key.
# The KEYS method gives us an object we can loop over that contains all the keys in the dictionary.
for key in post.keys():
    value = post[key]
    print(key, "=", value)

# This prints:
# user_id = 209
# message = D5 C5 E5 C4 G4
# language = English
# datetime = some date
# location = (44.590533, -104.715556)

# Dictionaries are not ordered data, so the data may print differently.

# Another way to iterate over all the key-value pairs is to use the ITEMS method, which will
# give you both the key and value in each step of the iteration:
for key, value in post.items():
    print(key, '=', value)

# To remove an item from a dictionary, you can use the POP or POPITEM method, which removes
# a single item from a dictionary, while the CLEAR method removes all

# pop(...)
# D.pop(k[,d]) -> v, remove specified key and return the corresponding value.

# popitem(self, /)
#  Remove and return a (key, value) pair as a 2-tuple.
```

```python
# 08-16-22 - Python and PRIME NUMBERS - Socratica - https://youtu.be/2p3kwF04xcA

# Prime numbers are the building blocks of whole numbers and are central to number theory.
# They are a key ingredient in cryptographic methods, like the RSA algorithm.
# Using Python to write algorithms to check if a number is a prime number.

# Composite numbers can be divided by themselves, 1, and at least one other number.
# Primes can only be divided by themselves and 1.
# 1 is called a UNIT and is neither prime nor composite.

# First step: check for all divisors from 2 to n-1, skipping 1 and n.

def is_prime_v1(n):
    """Return True if n is a prime number, and return False otherwise."""
    if n == 1:  # 1 is not a prime
        return False

    for d in range(2, n):  # Loop through all numbers from 2 to n-1
        if n % d == 0:  # Check if d (current number) can divide n evenly
            return False  # if so, n is not prime.

    return True  # if by the end of the loop we have not found another divisor
    # other than n and 1, n is a prime number, return True.


# Test the function:
for n in range(1, 21):
    print(n, is_prime_v1(n))

# ...........................................................................................#
# Now, compute the time it takes to check the numbers up to 100,000
import time

# t0 = time.time()    # Calling time function before and after loop to find out timing
# for n in range (1, 100000):
#     is_prime_v1(n)

# t1 = time.time()            # This method ends up taking a very long time, and we can do better.
# print("Time required = ", t1 - t0)

# To improve our function, we need to reduce the number of divisors we check.
# We only need to test the integers up to the square root of n, because after that, the
# factors just repeat but in reverse order:
# 12 = 12 X 1, 12 = 6 X 2, 12 = 4 X 3, 12 = square_root of 12 ^ 2 <- then it repeats backwards
# ...........................................................................................#

import math  # <- to work with square roots


# This time, only test divisors from 2 up to square root of n.

def is_prime_v2(n):
```

```
    """Return True if n is a prime number, and return False otherwise."""
    if n == 1:  # 1 is not a prime
        return False

    max_divisor = math.floor(math.sqrt(n))  # <- floor rounds down from the square root of n

    for d in range(2, max_divisor + 1):  # <- we add 1 to make sure we test by max divisor
        if n % d == 0:
            return False
    return True


print(n, is_prime_v2(n))  # <- testing to see that it works. It does.

# Now to see if it is faster than the first version:

t0 = time.time()
for n in range(1, 100000):
    is_prime_v2(n)
t1 = time.time()
print("Time required for version 2 = ", t1 - t0)  # PRINTS: Time required =  0.15463495254516602


# Version 2 takes a tiny fraction of the time version 1 took.

# .......................................................................................#
# There is, however, still room for improvement. In our loop, we go over all even integers
# and there is no reason to do so.
# We will now leave out integers greater than 2 that are even.

def is_prime_v3(n):
    """Return True if n is a prime number, and return False otherwise."""
    if n == 1:          # 1 is not a prime
        return False
    if n == 2:
        return False
    if n > 2 and n % 2 == 0:
        return False

    max_divisor = math.floor(math.sqrt(n))
    for d in range(3, max_divisor + 1, 2):  # <- This time we add a step value to skip evens
        if n % d == 0:                      # This will filter out half of all our operations
            return False
    return True


t0 = time.time()
for n in range(1, 100000):
    is_prime_v3(n)
t1 = time.time()
print("Time required for version 3 = ", t1 - t0)
```

```
# PRINTS: Time required for version 2 =  0.1607198715209961
#          Time required for version 3 =  0.09157681465148926

# Version 3 is almost twice as version 2.
# Look into subject of PSEUDO PRIMES - useful for building or cracking codes
# of extremely large numbers
```

```python
# 08-15-22 - RANDOM NUMBERS - SOCRATICA - https://youtu.be/zWL3z7NMqAs
# Random Module = high variety of functions for generating random numbers
# Good for games and Monte Carlo simulations

# WARNING: Numbers are only pseudo random with the Python module and should not be used
# for things like cryptography, etc.

import random
# dir(random)  <- Gives a list of the various funcions availble.
# We will use the random function, which returns a random number in the interval [0,1)
# This means it can return the number 0, but it can never return 1, signified by the
# open parentheses
# Display 10 random numbers from the interval [0,1)

for i in range(10):
    print(random.random())

# PRINTS:
# 0.13858663896059498
# 0.1929946880789366
# 0.4567729086905351
# 0.4806110226026603
# 0.29202033042693043
# 0.5519245785751102
# 0.22824189839569475
# 0.4394328413164742
# 0.9720256288475281
# 0.1551568037910266

# The random function represents uniform distribution, the probabilities of numbers being
# chosen are evenly spread out over the interval.

# Generate random numbers from the interval [3, 7)

def my_random():
    # Pick a random number, scale by the number that equals the difference between the
    # first number of your interval and the last. Shift the results up by the number
    # that represents the start of your interval, and return
    return 4*random.random() + 3
    # This will give us a random number between 3 and 7, since 4 is the difference
    # between the two, and we shift up by 3, the beginning of our interval

# Now, print 10 random numbers with this new random function:

for i in range(10):
    print(my_random())

# PRINTS:
# 5.001405992997202
# 5.380594209176506
# 3.411253829814249
```

```
# 5.478507732370224
# 4.023061076178072
# 4.712313263504037
# 4.503480354157892
# 6.142813462574594
# 3.7769175950334035
# 5.175132536056271

# The UNIFORM function from within the RANDOM module makes it easier to get random
# numbers from within an interval. But the examples above show how random.random
# can be used to generate customizable random number generators.

print(help(random.uniform))

# uniform(a, b) method of random.Random instance
    # Get a random number in the range [a, b) or [a, b] depending on rounding.

for i in range(10):
    print(random.uniform(3,7))

# PRINTS:
# 4.247857731256662
# 5.9709014331771275
# 4.601242756144457
# 4.752782730265457
# 4.278670269607018
# 4.182778998116497
# 5.440972444859195
# 6.7323491517536524
# 6.082411004744722
# 5.381939107234583

# Both random and uniform are uniform distributions.
# Often times though, other distributions are more preferable, for example
# NORMAL DISTRIBUTION, aka, the bell curve, based on the mean (average, where bell
# curve peaks) and standard deviation (how wide or narrow the curve is going
# out from the mean).

# For NORMAL DISTRIBUTION, use the NORMALVARIATE function, to which you must
# pass in the mean and the standard deviation.

# To print 20 numbers from a bell curve with a 0 mean and standard deviation of 1:

for i in range(20):
    print(random.normalvariate(0,1))

# PRINTS:            <- Bunched around the mean, 0
# 0.0901720865424814
# 0.49161628220402787
# 0.45427611584022276
# 1.487465984503258
# -0.5897630928234808
```

```
# 1.8214563333215432
# 1.0482769248437913
# -1.3062169087178548
# 0.3323780135289756
# 1.736488336357721
# 0.30990842135643687
# -0.11673472933075174
# -0.5572933915273687
# 1.1592818092763537
# 0.29770717273116154
# 1.9014547649237241
# -1.2502032426241523
# 1.7718965428883593

# The smaller the standard deviation, the more tightly grouped the resulting
# random numbers will be. And the larger the standard deviation, the more
# spread out they will be.

# DISCRETE PROBABILITY DISTRIBUTIONS:
# What if you want to simulate the roll of a die?
# use the RANDINT function! randint(min, max) - you will get a random whole number
# between the min and max you give it.

for i in range(20):
    print(random.randint(1, 6))


# RANDOM ELEMENT FROM A LIST: (RANDOM.CHOICE, and pass in the list of values to choose from)
# Apply this to Rock, Paper, Scissors

outcomes = ['rock', 'paper', 'scissors']
for i in range(20):
    print(random.choice(outcomes))
```

```python
# 08-14-22 - Socratica - Classes and Objects - https://youtu.be/apACNr7DC_s

# Think of a class as a template for creating objects with related data and functions that
# do interesting things with that data.
# Example will be a program to collect as much data as possible about users on a social
# media site:

# Define a class by typing class and the name of the class, which should have all words within
# it capitalized. Naming the class and typing pass is the simplest class possible. But it allows
# us to make users who go in our class.

# We will use pass for now, so that we can summarize objects in a class. More details below.
class User():
    pass


# To make a user, type in the name of the class it will belong to followed by parentheses.
# user1 is an instance or object of the User class, which in a way is calling a method of User.
user1 = User()

# To attach data to this object, type the name of the object, followed by . and a label for the data
# you want to add. Then give the specific data for that object that fits that label.
# A FIELD is data that is attached to an object, which stores data specific to the object it belongs to.
# Fields should not be capitalized. They should be lower case with words separated by underscores.
user1.first_name = 'Dave'
user1.last_name = 'Bowman'

# To access data about an object, you type it the same way you assigned it.
print(user1.first_name)
print(user1.last_name)

# The following variables are not attached to an object and just stand alone. The values are kept separate
# from those assigned to objects in our User class.
first_name = 'Arthur'
last_name = 'Clarke'
print(first_name, last_name)

print(user1.first_name, user1.last_name)

# With classes, there is no limit to the number of objects or instances you can make.
# To create more objects, use the exact same fields as in the first object, but now for a new object:
user2 = User()
user2.first_name = 'Frank'
user2.last_name = 'Poole'

print(user1.first_name, user1.last_name)
print(user2.first_name, user2.last_name)
```

```python

# You can attach additional information to your objects as desired, and they can be of any type.
user1.age = 37
user2.favorite_book = '2001: A Space Odyssey'


# Now, user1 and user2 have different fields from each other. If you try to print a field for an
# object that has not been assigned, you will get an AttributeError.

# What separates classes and their objects from dictionaries and other types of data structures
# are the additional features available such as Methods, Initialization, Help text, etc.

# Now, we will define our User class and utilize all the other features, including init, etc.

# NOTE: When working with classes and their methods, when you are working inside of a class,
# the information you want included in the class must all be indented beneath the class.
# The moment you unindent to the level of the class itself, you have ended that class.

class UserExpanded:
    # When you create a docstring as shown below, you can call the help function on your
    # class and get back the information that pertains to that class.
    """ A user / member from the social media site we are compiling information for."""
    # A function inside of a class is called a METHOD. init is the initialization function,
    # aka a constructor. It is called every time you create a new instance of the class.
    # The first argument, self, refers to the object itself that you are creating.
    # Following self are the arguments you want to include in your instances in the class.
    def _init_(self, full_name, birthday):
        # The arguments need to be stored to fields inside the object, as follows:
        # The value on the right side of the = is the value provided when you create a user object.
        # The one following the self. is what stores the value. This is what you use to refer to the
        # value when working with your objects.
        self.name = full_name,
        self.birthday = birthday  # format = yyyymmdd

        # Exact first and last methods using the split method, dividing on the space between them.
        # They will be saved in an array, as two strings, which we can use to create the first
        # and last name variables.
        # We must use self. when creating these, or we get an attribute error. It needs self. in
        # order to be attached to the object. Otherwise, it is just a variable that is not
        # accessible outside the method init, where we currently are. It is only used when writing
        # the method.
        name_sections = full_name.split(' ')
        self.first_name = name_sections[0]
        self.last_name = name_sections[-1]


# Now, when we create an instance or object of this new class, we need to give it values for the
# fields that the init method expects. They will be assigned in the order they were initialized.

```

```
    # Create a method for the user class that will return the age of the user in years:
    def age(self):
        """Return the age of the user in years."""
        # We will compute the user's age, so we need to import the datetime module.
        import datetime
        # First get today's date (using specific date for purposes of training and consistent code:
        today = datetime.date(2001, 5, 12)
        # Convert the user's birthday into a date object (There is a shorter way, but this explains):
        yyyy = int(self.birthday[0:4])  # Extracting year, which is the first 4 characters in
        mm = int(self.birthday[4:6])    # the birthday string, the month, which is the 5th and 6th
        dd = int(self.birthday[6:8])    # and the day, which are the last two.
        dob = datetime.date(yyyy, mm, dd)   # This creates the date of birth from info gathered above.
        # If you compute the difference between today and the birthday, you get a time-delta object.
        # The time-delta object has a field called days. We can divide by 365 to get the age in years.
        age_in_days = (today - dob).days
        age_in_years = age_in_days / 365
        # Return the age as an integer
        return int(age_in_years)


user3 = UserExpanded('David Bowman', '19710321')

print(user3.name)
print(user3.birthday)
print(user3.first_name)
print(user3.last_name)

user4 = UserExpanded("David Bowman", '19710321')
print(user4.age())   # <- Since age is gotten by the method created above, we need the () to get it now.
```

```python
# 08-22-22 - SPECIAL METHODS - SOCRATICA - https://youtu.be/IkWrlRei0uA

# _MAGIC METHODS!_ (and apparently how to override them...)

class Snowflake:
    pass

flake = Snowflake()
print(dir(flake))

# This gives us some, but not all, of the special class methods and attributes:

# PRINTS: ['_class_', '_delattr_', '_dict_', '_dir_', '_doc_', '_eq_',
# '_format_', '_ge_', '_getattribute_', '_gt_', '_hash_', '_init_',
# '_init_subclass_', '_le_', '_lt_', '_module_', '_ne_', '_new_',
# '_reduce_', '_reduce_ex_', '_repr_', '_setattr_', '_sizeof_', '_str_',
# '_subclasshook_', '_weakref_']


# Some useful special methods:
# _eq_ - called to compare objects for equality
# _setattr_ - called to set an attribute to an object
# _dict_ - special attribute that contains all of the object attributes.

print(flake._dict_)   # PRINTS: {} (Currently an empty dictionary of attributes)

flake.first_name = "Jane"
flake.last_name = "Jones"

print(flake._dict_)   # Now PRINTS: {'first_name': 'Jane', 'last_name': 'Jones'}

class Martian:
    '''Someone who lives on Mars.'''    # Saved in the _doc_ method for classes
    def _init_(self, first_name, last_name):
        self.first_name = first_name    # Here is where these attributes are assigned
        self.last_name = last_name      # to the _dict_ for this class and its objects.

    # When you assign an attribute to an object, the setattr method is called:
    def _setattr_(self, name, value):
        print(f'>>> you set {name} = {value}')

m1 = Martian("Robert", "Boudreaux")     # This is calling _init_ to create a new object.
m1.arrival_date = '2037-12-21'

print(m1._dict_)
# PRINTS: {'first_name': 'Robert', 'last_name': 'Boudreaux', 'arrival_date': '2037-12-21'}

m2 = Martian("Klaus", "Hohlerfeld")

# The _setattr_ method now prints:
# >>> you set first_name = Klaus
# >>> you set last_name = Hohlerfeld
```

```
# But if we print the _dict_ for Klaus, it contains nothing. Because we redefined the
# setattr method, which usually is responsible for communicating with _dict_ to
# create and object's dict, that part has not been carried out. We did not make it a
# part of our setattr. If we add to setattr: self._dict_[name] = value, it will
# create the dictionary for the objects we create under the Martian class.

# I am going to continue just listening to this video, because these are not things
# that I will ever need to do, but it is good to know how all this works.

# the ._str_() method will give you and object's hexadecimal address in memory.
# the id(object) method will give you a base 10 integer id for the object.
```

```python
# 08-14-22 -  Map, Filter, and Reduce - Socratica - https://youtu.be/hUes6y2b--0

# These functions are primarily used with lists.

# MAP
import math


# Suppose we have a function that computes the area of a circle with radius(r).

def area_circle(r):
    """Calculate the area of a circle, with radius r."""
    return math.pi * (r ** 2)


# What if we want to compute the area of many different circles?
radii_list = [2, 5, 7.1, 0.3, 10]

# Method 1: Direct method of creating an empty list of areas and loop over the
# list of radii and append each computed area to the list at the end of each loop.

# Method 2: Use the MAP function, and do it all in one line.
# MAP takes two arguments: a function and your list, tuple, or other iterable object.
# Here, MAP will apply the area_circle function to each element in the list of radii.
# But the output of the map function when done this way is not a list. It is a map
# object, which is actual an iterator over the results.
print(map(area_circle, radii_list))      # <map object at 0x7f97a8209330>

# We can turn this into a list by passing the map to the list constructor
                                              #[12.566370614359172, 78.53981633974483,
print(list(map(area_circle, radii_list)))   # 158.36768566746147, 0.2827433388230814,
                                              # 314.1592653589793]

# HOW THE MAP FUNCTION WORKS:
# If you have an iterable collection like a list or tuple and want to apply a function
# to each piece of data in one short line:

# Data = a1, a2, a3, ... an
# Function = f
# map(f, a)          <- Returns - f(a1), f(a2), f(a3,) ... f(an) - iterated over

# Units: Celsius
# Desired Fahrenheit to Celsius Temps List
# List of temperature datas in tuples with the name of a city and temp in Celsius.
temps = [('Berlin', 29), ('Cairo', 36), ('Buenos Aires', 19), ('Los Angeles', 26),
         ('Tokyo', 27), ('New York', 28), ('London', 22), ('Beijing', 32)]

# Function to convert Celsius to Fahrenheit: that will take a tuple as the input and
# return a tuple with the same name but the temp in Fahrenheit instead of Celsius.
c_to_f = lambda data: (data[0],  9/5 * data[1] + 32 )

# Now we can create a list of data in Fahrenheit by mapping the converter function
```

```
# to our list of data.

print(list(map(c_to_f, temps)))
# Prints: [('Berlin', 84.2), ('Cairo', 96.8), ('Buenos Aires', 66.2),
# ('Los Angeles', 78.80000000000001), ('Tokyo', 80.6), ('New York', 82.4), ('London', 71.6), ('Beijing', 89.6)]

# FILTER Function: use to select certain pieces of data from a list, tuple, or other iterable collection of data.
# It filters out the data you do not need.

# Suppose you are analyzing some data, and you would like to select all values that are above the average.
# Import the statistics module since it contains the MEAN function:

import statistics

data = [1.3, 2.7, 0.8, 4.1, 4.3, -0.1]
avg = statistics.mean(data)
print(avg)      # Prints: 2.183333333333333

# To filter out the values above the average, we use filter similarly to how we use map.
# The first argument is a function, and the second is the data we want to apply the function to.
print(filter(lambda x: x > avg, data))      # Prints: <filter object at 0x7fb458253fd0>

# Once again, not a list, but this time a filter object, which is an iterator over the results.
print(list(filter(lambda x: x > avg, data)))        # Prints: [2.7, 4.1, 4.3]

print(list(filter(lambda x: x < avg, data)))        # Prints values below average: [1.3, 0.8, -0.1]

# REMOVING MISSING DATA: For when you are working with data that contains empty values
countries = ['', 'Argentina', 'Brazil', 'Chile', '', 'Columbia', 'Ecuador', '', '', 'Venezuela']

# Instead of a function this time for the first argument, we will pass None.
# This filters out all values that are treated as false.
print(list(filter(None, countries)))
# Prints: ['Argentina', 'Brazil', 'Chile', 'Columbia', 'Ecuador', 'Venezuela']

# In Python, values treated as false are an empty string, ''; zero, 0, 0.0, 0j; an empty tuple, ();
# an empty list, []; empty dictionary, {}; False; None; and those objects that signal to Python that
# it is a trivial instance.
# Be careful using FILTER in this way, since 0 is often a valid piece of information.

# REDUCE: No longer a built-in function and is now in functools. Use when needed, but most of the
# a for loop is more readable.
# It works similarly to map and filter in that you pass it a function and the data to which you
# want to apply the function:
# data = [a1, a2, a3, ... an]
# function: f(x, y)
```

```python
# reduce(f, data)
# STEP 1: val1 = f(a1, a2)
# STEP 2: val2 = f(val1, a3)
# STEP 3: val3 = f(val2, a4)
# ...
# STEP n-1: val_n-1 = f(val_n-2, an)
# return val_n-1

# In each step, it applies f to the output value and to the next term in the sequence.
# Once it has reached the last piece of data, it will return the final value.
# Alternatively, it computes this nested function:
# f(f(f(a1, a2), a3), a4),...an)

from functools import reduce
# Multiply all numbers in a list:
data = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
multiplier = lambda x, y: x*y
print(reduce(multiplier, data))     # Prints: 6469693230

# As a for loop:
product = 1
for x in data:
    product = product * x
print(product)      # Prints: 6469693230
```

```python
# 08-14-22 - LAMBDA EXPRESSIONS: Socratica - https://youtu.be/25ovCm9jKfA

# Lambda Expressions: Nameless functions. Commonly used for sorting and filtering data.
# Lambda is just a keyword that tells Python that what follows will be an anonymous,
# or nameless, function.

# Write a function to compute (3x + 1)
def f(x):
    return 3 * x + 1

print(f(2))  # <- Prints 7

# Now let's do this with an anonymous function / lambda expression.
# Start by typing lambda, followed by your inputs and a colon, and then the expression that
# will be the return value.
# lambda x: 3*x + 1     LAMBDA + INPUTS + COLON + EXPRESSION
# Now to use it, we need to give it a name or use it inside some other function or code.
g = lambda x: 3 * x + 1
print(g(2))  # <- Prints 7

# LAMBDAS with MULTIPLE INPUTS:
# Write a function to take the first and last name of a user and combine it into the full
# name so that it can be displayed completely on a user interface.
# Using strip takes out the leading and trailing white space. And .title makes sure that
# only the first letter of the names are capitalized. (Humans are sloppy.)

full_name = lambda fn, ln: fn.strip().title() + ' ' + ln.strip().title()
example = full_name("   leonhard", "EULER")  # <- Messy user input into first name and last name fields
print(example)      # <- Prints: Leonhard Euler

# Remember: (Optional Name) = LAMBDA + zero or more INPUTS + COLON + a single EXPRESSION (the return value)
# They cannot be used for multi-line functions

# EXAMPLES:
# lambda : "What is my purpose?"
# lambda x: 3*x + 1
# lambda x, y: (x*y) **0.5     # Geometric mean
# lambda x, y, z: 3/(1/x + 1/y + 1/z) # Harmonic mean
# lambda x1, x2, x3 ...xn <expression>

# Lambdas where we do not give it a name: We have a list of scifi authors to organize by last name.
# Some how initials, some have middle names, etc. We will write a function that extracts the last
# name and uses that as the sorting value.

scifi_authors = ['Isaac Asimov', 'Ray Bradbury', 'Robert Heinlein', 'Arthur C. Clarke', 'Frank Herbert',
                 'Orson Scott Card', 'Douglas Adams', 'H.G. Wells', 'Leigh Brackett']

# Lists have a built-in method, sort, which we will use. We will split on the blank space, access
```

```
# the last part of the name element by using (-1), and convert the string to lower case, to ensure
# the sorting is not case sensitive.
scifi_authors.sort(key = lambda name: name.split(' ')[-1].lower())
print(scifi_authors)
# The list is now in alphabetical order:
# Prints: ['Douglas Adams', 'Isaac Asimov', 'Leigh Brackett', 'Ray Bradbury', 'Orson Scott Card',
        # 'Arthur C. Clarke', 'Robert Heinlein', 'Frank Herbert', 'H.G. Wells']


# Write a function that makes functions.
# Working with quadratic functions: f(x) = ax^2 + bx + c

def build_quadratic_function(a, b, c):
    """Returns the function f(x) = ax^2 + bx + c"""
    return lambda x: a*x**2 + b*x + c

f = build_quadratic_function(2, 3, -5)
print(f(2))
print(f(1))
print(f(0))

print(build_quadratic_function(3, 0, 1)(2))    # 3x^2 + 1 evaluated for x = 2
```

```python
# LIST MAKING and LIST COMPREHENSION
# FOR LOOPS -> LIST COMPREHENSION
# ........................................................................................ #
# LISTS NUMBER 1:
fruits = ['apples', 'banana', 'raspberries', 'blueberries', 'grapefruit', 'dragonfruit']
a_fruit_list = []
for fruit in fruits:
    if 'a' in fruit:
        a_fruit_list.append(fruit)
print(a_fruit_list)

fruits = ['apples', 'banana', 'raspberries', 'blueberries', 'grapefruit', 'dragonfruit']
a_fruit_list = list(filter(lambda fruit: 'a' in fruit, fruits))
print("Fruits with a in their name: ", a_fruit_list)

# ........................................................................................ #
# LISTS NUMBER 2:
dogs = ['chihuahua', 'labrador', 'terrier', 'mutt', 'poodle', 'dingo', 'boxer', 'golden']
dogs_with_e = []
for dog in dogs:
    if 'e' in dog:
        dogs_with_e.append(dog)
print(dogs_with_e)

dogs = ['chihuahua', 'labrador', 'terrier', 'mutt', 'poodle', 'dingo', 'boxer', 'golden']
dogs_with_e = list(filter(lambda dog: "e" in dog, dogs))
print("Dogs with e in their name: ", dogs_with_e)

# ........................................................................................ #
# LISTS NUMBER 3:
pizzas = ['pepperoni, meat', 'cheese', 'margherita', 'pineapple', 'meat-lovers, meat', 'white']
pizzas_meat = []
for pizza in pizzas:
    if 'meat' in pizza:
        pizzas_meat.append(pizza)
print(pizzas_meat)

pizzas = ['pepperoni, meat', 'cheese', 'margherita', 'pineapple', 'meat-lovers, meat', 'white']
pizzas_meat = list(filter(lambda pizza: 'meat' in pizza, pizzas))
print("Pizzas with meat: ", pizzas_meat)

# ........................................................................................ #
# LISTS NUMBER 4:
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
squares = []
for number in numbers:
    square = number * number
    squares.append(square)
print(squares)
```

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
squares = list(map(lambda number: number ** 2, numbers))
print("Squares = ", squares)

# ........................................................................................ #
# STRINGS NUMBER 1 (example from "master_notebook" to work from:
# a)
sentence = "The bear went over the mountain."
vowels = [v for v in sentence if v in "aeiou"]
print("Strings Number 1a: ", vowels)

# b)
sentence = "If you're happy and you know it, clap your hands!"
def is_consonant(letter):
    vowels = 'aeiou'
    return letter.isalpha() and letter.lower() not in vowels
consonants = [i for i in sentence if is_consonant(i)]
print("Strings Number 1b: ", consonants)

# ........................................................................................ #
# MORE COMPLICATED NUMBER 2:
prices = (12.00, 14.75, 15.00, 45.98, 54.00, 34.65)
def signed_price(price):
    return (f'${round(price):.2f}')
rounded_prices = [signed_price(i)for i in prices if i > 20]
print("More Complicated 2: ", rounded_prices)

# ........................................................................................ #
# MORE COMPLICATED NUMBER 3:
ages = (12, 8, 3, 15, 13, 4, 11, 17)
older_children = [age for age in ages if age > 12]
print("More Complicated 3: ", older_children)

# ........................................................................................ #
# MORE COMPLICATED NUMBER 4:
days = ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday')
weekend = ['Saturday', 'Sunday']
print([i for i in days if i not in weekend])
# QUESTION: Why did I get <generator object <genexpr> at 0x7facc056f6f0> when I printed until I
# put square brackets inside of the print statement?
# Because it is a generator if you have just parentheses or is inside a function, which means
# it won't actually do the comprehension until something else iterates on it.

# ........................................................................................ #
# MORE COMPLICATED NUMBER 5:
people = {'Jill': 'female', 'John': 'male', 'Hektor': 'male', 'Ellen': 'female', 'Lilly': 'female', 'Bill': 'male'}
men = {name for name, gender in people if gender == 'male'}
print(men)
# QUESTION: How do I get it to only print the men's names?

```

```python
# ........................................................................................ #
# MORE COMPLICATED NUMBER 6 (ZIP):

dogs = ['Chester', 'Francis', 'George', 'Bully', 'Felix', 'Sandy']
owners = ['Betty', 'Alfred', 'Benjamin', 'Tammy', 'Lucy', 'Hank']
dogs_owners = list(zip(dogs, owners))
print(dogs_owners)
# QUESTION: Why do I get [<zip object at 0x7fab50162d00>] if I do not make dogs_owners a list ?
# Zip is a generator, and list function runs the generator and iterates through and adds to list one by one.

# As a dictionary
dogs = ['Chester', 'Francis', 'George', 'Bully', 'Felix', 'Sandy']
owners = ['Betty', 'Alfred', 'Benjamin', 'Tammy', 'Lucy', 'Hank']
dogs_owners = dict(zip(dogs, owners))
print(dogs_owners)
# Dictionary version is a 'mapping', mapping a key to a value vs the one above that returns a list of tuples.

# ........................................................................................ #
# https://www.youtube.com/watch?v=AhSvKGTh28Q
# Socratica: List Comprehension || Python Tutorial

# Lists = collection of data inside of brackets, separated by commas
# List Comprehensions = also surrounded by brackets but with for loops and conditionals
# [expression for value in collection, followed by for loop, followed by conditional]
# Can have more than one conditional, and only items matching all claueses will be added to list.
# Can loop over more than one collection:
        # [expr for val_1 in collection_1 and expr for val_2 in collection_2]

# Examples:

squares = []
for i in range(1, 101):
    squares.append(i**2)

# List comprehension version of for loop above.
squares_2 = [i**2 for i in range(1, 101)]

# Remainders for squares 1-100 when divided by 5
remainders_by_5 = [i**2 % 5 for x in range(1, 101)]

# Remainders for squares 1-100 when divided by 11
remainders_by_11 = [i**2 % 11 for x in range(1, 101)]

# Quadratic Reciprocity:
# p_remainders = [x**2 % p for x in range(0, p)]
# len(p_remainders) = (p+1) / 2
# QUESTION: Explain ^^^^^ - Look it up

```

```python
# Pull out movies that start with G:
movies = ['Star Wars', 'Ghandi', 'Casablanca', 'Gone with the Wind', 'Citizen Cane',
          'Gattaca', 'Raiders of the Lost Arc', '2001: A Space Odyssey', 'Groundhog Day']
gmovies = []
for movie in movies:
    if movie.startswith('G'):
        gmovies.append(movie)

gmovies = [movie for movie in movies if movie.startswith('G')]

# Now, movies is a list of tuples that also contains year of release - Get all made before 2000
movies = [('It\'s a Wonderful Life', 1946), ('Spirited Away', 2001), ('No Country for Old Men', 2007),
          ('Gone with the Wind', 1926), ('Citizen Cane', 1941), ('Gattaca', 1997), ('Groundhog Day', 1993),
          ('The Aviator', 2004)]
pre2k_movies = [movie for (movie, year) in movies if year < 2000]
print(pre2k_movies)

# SCALAR MULTIPLICATION:
v = [2, -3, 1]
product = [x*4 for x in v]
print(product)

# CARTESIAN PRODUCT:
# If A and B are sets, the Cartesian product set of pairs, (a, b), where a is in A and b is in B.
# A = {1, 3}
# B = {x,y}
# AxB = {(1,x), (1,y), (3,x), (3,y)}

A = [1, 3, 5, 7]
B = [2, 4, 6, 8]
# Use two for loops as shown here:
cartesian_product = [(a, b) for a in A for b in B]
print(cartesian_product)
```

```python
# 08-15-22 - RANDOM WALK and MONTE CARLO SIMULATION - Socratica - https://youtu.be/BfS2H1y6tzQ

# RANDOM WALK - Direction is chosen at random every step along the way.
# What is the longest random walk you can take and on average end up 4 blocks
# or fewer from home?

# Write a function that simulates a random walk of n blocks for the challenge:
# (Way 1 will be simple, clear, and straight forward, while Way 2 will be
# focused on being short and using Python shortcuts to cut the length of
# the function in half.
#..........................................................................#
import random


def random_walk(n):
    """ This will simulate a random walk. Return coordinates after n blocks
    of a random walk. Your position throughout the function will have an
    x and y coordinate, both starting at 0."""
    x = 0
    y = 0
    # n is how many blocks long our random walk is.
    for i in range(n):
        # We will choose from a list of the four possible directions
        step = random.choice(['N', 'S', 'E', 'W'])
        # The following expresses the changes in our coordinates depending on
        # direction we walk in.
        if step == 'N':
            y = y + 1
        elif step == 'S':
            y = y - 1
        elif step == 'E':
            x = x + 1
        else:
            x = x - 1
    return (x, y)


# To test the function, let's take 25 random walks, each 10 blocks long
for i in range(25):
    walk = random_walk(10)

    # For each walk, display the coordinates and distance from home.
    # The distance from home is the sum of the absolute value of the
    # x and y coordinates.
    print(walk, 'Distance from home = ',
          abs(walk[0]) + abs(walk[1]))

#..........................................................................#
# MORE COMPACT VERSION!

def random_walk_2(n):
```

```
    """More concise version of the random walk function above with
    same objective and return."""
    # Our x and y assignments can be done in one line, assigning the
    # first value to the first variable and second to second.
    x, y = 0, 0

    for i in range(n):
        # This time instead of randomly choosing N, S, E, or W, we will
        # choose a random pair of numbers, dx and dy, (difference in x and
        # difference in y) which will contain the values we will add or
        # subtract from x and y. The following coordinates represent the
        # choices of N, S, E, and W and the coordinate shift that goes
        # with each.
        (dx, dy) = random.choice([(0, 1), (0, -1), (1, 0), (-1, 0)])
        # Now use dx and dy to update x and y:
        x += dx
        y += dy
    return x, y


# Testing the new function:
# for i in range(25):
#    walk = random_walk_2(10)
# print(walk, 'Distance from home = ',
#      abs(walk[0]) + abs(walk[1]))

#..................................................................................#
# MONTE CARLO METHOD TO SOLVE: What is the longest random walk you can take and on
# average end up 4 blocks or fewer from home?

# We will perform thousands of random trials and compute the percentage of random
# walks that end up 4 blocks or fewer from home. If farther than 4 blocks, we will
# take transportation home.
# To get an accurate output, we will take 10000 random walks for each walk length.

number_of_walks = 44000

# Estimate the probability you will walk home for walks of length 1 to 30 blocks.
for walk_length in range(1, 31):
    # Keeps track of how many walks end up in walks 4 blocks or fewer from home.
    no_transport = 0
    # Now for our Monte Carlo loop of 10000 samples:
    for i in range(number_of_walks):
        # First, get a random walk of length walk_length
        (x,y) = random_walk_2(walk_length)
        # Next compute the distance from home. If the distance is less than 4 blocks
        # from home, increment the no_transport counter.
        distance = abs(x) + abs(y)
        if distance <= 4:
            no_transport += 1
        # We can now computer the percentage of walks that ended with a walk home.
        # It is just the fraction of 10000 random walks that required no transport.
```

```
no_transport_percentage = float(no_transport) / number_of_walks
    # Finally, print out the results for this walk size:
print("Walk size = ", walk_length,
      "/ % of no transport = ", 100*no_transport_percentage)
```

```python
# 08-15-22 - Recursion, the Fibonacci Sequence and Memoization - SOCRATICA - https://youtu.be/Qk0zUZW-U_M
# COUNTING BUNNY REPRODUCTION (which follows fibonacci sequence)
# To write a function employing the Fibonacci sequence, we must use recursion, and to make the function
# efficient, we will use memoization.

# fibonacci = 1, 1, 2, 3, 5, 8, 13, 21

# The Fibonacci sequence works like this: the first two numbers are 1s, but after that, each number is
# the sum of the two numbers that come before.
# GOAL: Write a fast and clearly-written function returning the nth term of the Fibonacci sequence.

def fibonacci(n):
    if n == 1:
        return 1
    elif n == 2:
        return 1
    elif n > 2:
        return fibonacci(n - 1) + fibonacci(n - 2)
        # This is where the recursion happens: the previous two terms are added together
        # and equal the next term

    # We will try this out now on the first 10 terms, ranging to 11, since RANGE FUNCTION will go to
    # the second to last term when it runs.


for n in range(1, 11):
    print(n, ':', fibonacci(n))  # Prints 2 columns, n for the loop we are on, and the Fibonacci
    # sequence integer at that loop level

# for n in range(1, 101):
# print(n, ":", fibonacci(n)) # The function slows down greatly after the first dozen or so loops
# The recursion here makes the computer repeat itself over and over needlessly

# MEMOIZATION = the cure for this recursive and demanding function:
# Idea = Cache the values = store the values for recent function calls so future calls do not need
# to repeat the work.

# 1) IMPLEMENTING MEMOIZATION EXPLICITLY to see how it works:
fibonacci_cache = {}  # For storing recent function calls

# Rewrite fibonacci function to check if the nth value we are on is already in our cache.
# If it is, simply return it.
def fibonacci_memo(n):
    if n in fibonacci_cache:
        return fibonacci_cache[n]
    # Otherwise, compute the Nth term, cache it, and return it.
```

```
    if n == 1:
        return 1
    elif n == 2:
        return 1
    elif n > 2:
        value = fibonacci_memo(n - 1) + fibonacci_memo(n - 2)
    # Cache the value
    fibonacci_cache[n] = value
    return value

# Now, it will print the first 100 very quickly.
# for n in range(1, 101):
  #   print(n, ":", fibonacci_memo(n))

# Try the first 1000 - WHOA!
# for n in range(1, 1001):
   #  print(n, ":", fibonacci_memo(n))


# 2) USE BUILT-IN PYTHON TOOL that makes MEMOIZATION trivial:
# This time, more simply put (We will use our first version of the function with it.)

from functools import lru_cache      # <- Stands for Least Recently Used Cache
                                     # Provides a 1-line way to implement memoization
@lru_cache(maxsize=1000)     # <- max values to cache, by default, it is 120
def fibonacci_func(n):
    # Check that input is a integer, or the tool will not work
    if type(n) != int:
        raise TypeError("n must be an integer.")    # Raise type error if not an integer
    # Check that the integer is positive, or it also will not work
    if n < 1:
        raise ValueError("n must be a positive integer.") # Raise value error if not positive
    if n == 1:
        return 1
    elif n == 2:
        return 1
    elif n > 2:
        return fibonacci_func(n - 1) + fibonacci_func(n - 2)

for n in range(1, 501):
    print(n, ':', fibonacci_func(n))

# Now, print up to 50:
for n in range(1, 51):
    print(fibonacci_func(n))

# The numbers grow quickly in size.
# Now, compute the ratio between consecutive terms:
for n in range(1, 51):
    print(fibonacci_func(n+1) / fibonacci_func(n))

# The ratio between consecutive terms converges to the golden ratio by the last 10 or so
```

```
# prints: 1.618033988749895
```