python_project

February 28, 2024

1 Project: Python Essentials for Data Scientists

https://courses.dataschool.io/view/courses/python-essentials-for-data-scientists

2 Project: Part 1

2.1 Project Overview

This project is a series of 7 exercises inspired by Veritasium's excellent 22-minute video, The Simplest Math Problem No One Can Solve.

If you have the time, I recommend watching the entire video right now! (Watching it now won't "spoil" anything about the project.)

But if you don't have the time, you can simply watch the short section of video related to each exercise before working on it! (I'll link to that section in the instructions for each exercise.)

2.2 While Loops

[1]:
```
x = 4
```

[2]:
```
# if statement checks a condition, runs once
if x > 0:
    print('positive')
```

```
positive
```

[3]:
```
# for loop runs one or more times
for letter in 'hello':
    print(letter)
```

```
h
e
l
l
o
```

[4]:
```
# while loop checks a condition, runs as long as condition is true
# condition should eventually become false
while x > 0:
    print(x)
    x = x - 1
```

```
4
3
2
1
```

Article: Python "while" Loops (Real Python)

2.3 f-strings

[5]:
```
x = 4
```

[6]:
```
if x > 0:
    print('positive')
```

```
positive
```

[7]:
```
# you can pass multiple objects to the print function
if x > 0:
    print(x, 'is positive')
```

```
4 is positive
```

[8]:
```
# substitute an object into a string using an f-string
if x > 0:
    print(f'{x} is positive')
```

```
4 is positive
```

[9]:
```
name = 'Kevin'
```

[10]:
```
# call a method in an f-string
f'My name is {name.upper()}'
```

[10]:
```
'My name is KEVIN'
```

[11]:
```
# use a function in an f-string
f'My name is {name}, which has {len(name)} letters'
```

[11]:
```
'My name is Kevin, which has 5 letters'
```

[12]:
```
# evaluate an expression in an f-string
f'Divide {x} by 3 to get {x / 3}'
```

[12]:
```
'Divide 4 by 3 to get 1.3333333333333333'
```

[13]:
```
# use a "format specification" to format a number
f'Divide {x} by 3 to get {x / 3:.2f}'
```

[13]:
```
'Divide 4 by 3 to get 1.33'
```

Article: Python f-string tips & cheat sheets (Python Morsels)

2.4 Mathematical Operators

[14]:
```
# addition operator
5 + 3
```

[14]:
```
8
```

[15]:
```
# subtraction operator
5 - 3
```

[15]:
```
2
```

[16]:
```
# multiplication operator
5 * 3
```

[16]:
```
15
```

[17]:
```
# division operator
5 / 3
```

[17]:
```
1.6666666666666667
```

[18]:
```
# "true division" always returns a float
6 / 3
```

[18]:
```
2.0
```

[19]:
```
# "floor division" rounds down
5 // 3
```

[19]:
```
1
```

[20]:
```
6 // 3
```

[20]:
```
2
```

[21]:
```
# converting to an integer is equivalent to floor division
int(5 / 3)
```

[21]:
```
1
```

[22]:
```
int(6 / 3)
```

[22]: 2

[23]:
```
# round function rounds to the nearest integer
round(5 / 3)
```

[23]: 2

[24]:
```
# you can specify the number of decimal places
round(5 / 3, 2)
```

[24]: 1.67

[25]:
```
round(6 / 3, 2)
```

[25]: 2.0

[26]:
```
# modulo operator returns the remainder after division
5 % 3
```

[26]: 2

[27]:
```
6 % 3
```

[27]: 0

[28]:
```
7 % 3
```

[28]: 1

2.5 Project Exercise 1

This exercise relates to the following portion of the video: 0:00 to 3:31.

Define a function called `path()` that accepts one required argument called `start`. (We are requiring `start` to be a positive integer, but your function doesn't need to validate that it's a positive integer.)

Within the function, do the following:

- Create a list called `nums` in which the first element is `start`.
- Check if `start` is odd or even:
  - If odd, then multiply by 3 and add 1.
  - If even, then divide by 2. (Make sure the result is an integer, not a float.)
  - Append the resulting number to `nums`.
- Repeat this pattern until you reach the number 1, and then stop.
- Print out `nums`, which should start with `start` and end with 1.
- Use the `max()` function to calculate the maximum number reached, and print that out.
- Print out the number of steps it took to arrive at 1. (Inputting the starting number does not count as a step.)

For example, if you run `path(10)`, it should print out this:

```
path: [10, 5, 16, 8, 4, 2, 1]
max: 16
steps: 6
```

2.6 Solution to Exercise 1

[29]:
```python
def path(start):

    nums = [start]  # create a list with one number
    num = start     # we will modify num (rather than start)
```

[30]:
```python
def path(start):

    nums = [start]
    num = start

    while num > 1:          # stop running once num is 1

        if num % 2 == 0:    # check if even
            num = num // 2  # use floor division to return an integer
        else:
            num = num * 3 + 1

        nums.append(num)
```

[31]:
```python
def path(start):

    nums = [start]
    num = start

    while num > 1:

        if num % 2 == 0:
            num = num // 2
        else:
            num = num * 3 + 1

        nums.append(num)

    print(f'path: {nums}')
    print(f'max: {max(nums)}')        # you can pass any iterable to max
    print(f'steps: {len(nums) - 1}')  # starting number is not a step
```

[32]:
```python
path(10)
```

```
path: [10, 5, 16, 8, 4, 2, 1]
max: 16
steps: 6
```

[33]:
```
# functions without a return statement implicitly return None
path(10) == None
```

```
path: [10, 5, 16, 8, 4, 2, 1]
max: 16
steps: 6
```

[33]: True

[34]:
```
path(7)
```

```
path: [7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
max: 52
steps: 16
```

[35]:
```
path(26)
```

```
path: [26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
max: 40
steps: 10
```

[36]:
```
path(27)
```

```
path: [27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242,
121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175,
526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251,
754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158,
1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051,
6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976,
488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16,
8, 4, 2, 1]
max: 9232
steps: 111
```

3 Project: Part 2

3.1 Separating Functions

[37]:
```
# calculates the path and prints info about the path
def path(start):

    nums = [start]
    num = start

    while num > 1:

        if num % 2 == 0:
            num = num // 2
        else:
```

```
            num = num * 3 + 1

        nums.append(num)

    print(f'path: {nums}')
    print(f'max: {max(nums)}')
    print(f'steps: {len(nums) - 1}')
```

[38]:
```
# calculates the path and returns it
def get_path(start):

    nums = [start]
    num = start

    while num > 1:

        if num % 2 == 0:
            num = num // 2
        else:
            num = num * 3 + 1

        nums.append(num)

    return nums
```

[39]:
```
get_path(10)
```

[39]:
```
[10, 5, 16, 8, 4, 2, 1]
```

[40]:
```
# prints info about the path
def print_path_info(start):

    nums = get_path(start)
    print(f'path: {nums}')
    print(f'max: {max(nums)}')
    print(f'steps: {len(nums) - 1}')
```

[41]:
```
print_path_info(10)
```

```
path: [10, 5, 16, 8, 4, 2, 1]
max: 16
steps: 6
```

[42]:
```
# get_path and print_path_info don't need to use the same variable names
def print_path_info(s):

    n = get_path(s)
    print(f'path: {n}')
```

```
    print(f'max: {max(n)}')
    print(f'steps: {len(n) - 1}')
```

[43]:
```
print_path_info(10)
```

```
path: [10, 5, 16, 8, 4, 2, 1]
max: 16
steps: 6
```

Video: Variable Scope (Corey Schafer)

3.2 Writing Docstrings

[44]:
```
# triple-quoted string at the start of a function becomes the docstring
def get_path(start):
    """Given a starting value (positive integer), return the path (list of␣
 ↪integers)."""

    nums = [start]
    num = start

    while num > 1:

        if num % 2 == 0:
            num = num // 2
        else:
            num = num * 3 + 1

        nums.append(num)

    return nums
```

[45]:
```
# multi-line docstring allows for more details
def get_path(start):
    """
    Given a starting value, return the path.

    Args:
        start (int): Positive starting value
    Returns:
        List of integers representing the path
    """

    nums = [start]
    num = start

    while num > 1:
```

```
        if num % 2 == 0:
            num = num // 2
        else:
            num = num * 3 + 1

        nums.append(num)

    return nums
```

[46]:
```
# docstring gets printed by help function
help(get_path)
```

```
Help on function get_path in module __main__:

get_path(start)
    Given a starting value, return the path.

    Args:
        start (int): Positive starting value
    Returns:
        List of integers representing the path
```

[47]:
```
def print_path_info(start):
    """
    Given a starting value, print the path information.

    Args:
        start (int): Positive starting value
    Returns:
        None
    """

    nums = get_path(start)
    print(f'path: {nums}')
    print(f'max: {max(nums)}')
    print(f'steps: {len(nums) - 1}')
```

[48]:
```
help(print_path_info)
```

```
Help on function print_path_info in module __main__:

print_path_info(start)
    Given a starting value, print the path information.

    Args:
        start (int): Positive starting value
    Returns:
```

```
None
```

Article: Docstring Formats (Real Python)

3.3 Project Exercise 2

Exercises 2, 3, and 4 all relate to the following portion of the video: 5:10 to 6:57.

Define a function (with a docstring) called `get_first_digits()` that accepts one required argument called `start`, which is a positive integer.

Within the function, do the following:

- Calculate the path, using `start` as the starting value.
- Return a list of integers containing the first digit of each number in that path. (You can use any method you like to extract the first digit, but I recommend using string indexing.)

For example, `get_first_digits(10)` should return `[1, 5, 1, 8, 4, 2, 1]`.

3.4 Solution to Exercise 2

[49]:
```
# convert integer to string
str(234)
```

[49]:
```
'234'
```

[50]:
```
# extract first digit and convert back to integer
int(str(234)[0])
```

[50]:
```
2
```

[51]:
```
int(str(2)[0])
```

[51]:
```
2
```

[52]:
```
def get_first_digits(start):
    """
    Given a starting value, return the first digit of each number in the path.

    Args:
        start (int): Positive starting value
    Returns:
        List of integers representing the first digit of each number in the path
    """
```

[53]:
```
def get_first_digits(start):
    """
    Given a starting value, return the first digit of each number in the path.

    Args:
```

```
        start (int): Positive starting value
    Returns:
        List of integers representing the first digit of each number in the path
    """

    nums = get_path(start)
    return [int(str(num)[0]) for num in nums]  # list comprehension
```

[54]:
```
get_path(10)
```

[54]:
```
[10, 5, 16, 8, 4, 2, 1]
```

[55]:
```
get_first_digits(10)
```

[55]:
```
[1, 5, 1, 8, 4, 2, 1]
```

[56]:
```
get_path(26)
```

[56]:
```
[26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

[57]:
```
get_first_digits(26)
```

[57]:
```
[2, 1, 4, 2, 1, 5, 1, 8, 4, 2, 1]
```

4 Project: Part 3

4.1 Classes

[58]:
```
# import sqrt function from the math module
from math import sqrt
```

[59]:
```
# call sqrt function, get back a return value
sqrt(49)
```

[59]:
```
7.0
```

[60]:
```
# import Counter class from the collections module
# class names usually start with capital letter
from collections import Counter
```

[61]:
```
# call Counter class, get back a Counter object (an "instance" of Counter)
Counter('hello')
```

[61]:
```
Counter({'h': 1, 'e': 1, 'l': 2, 'o': 1})
```

[62]:
```
# call str class, get back a str object
str(1011)
```

[62]:
```
'1011'
```

[63]:
```
s = str(1011)
type(s)
```

[63]:
```
str
```

[64]:
```
# call int class, get back an int object
int('1011')
```

[64]:
```
1011
```

[65]:
```
# call list class, get back a list object
list('1011')
```

[65]:
```
['1', '0', '1', '1']
```

[66]:
```
# count is a function defined for str class, thus strings have a count method
s.count('1')
```

[66]:
```
3
```

[67]:
```
# str class doesn't contain an append function
s.append('1')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[67], line 2
      1 # str class doesn't contain an append function
----> 2 s.append('1')

AttributeError: 'str' object has no attribute 'append'
```

Article & Video: What is a class? (Python Morsels)

4.2 Counter

[68]:
```
# pass an iterable to Counter, it counts how many times each element appears
# an iterable is anything you can loop through (string, list, dictionary, etc.)
Counter('hello')
```

[68]:
```
Counter({'h': 1, 'e': 1, 'l': 2, 'o': 1})
```

[69]:
```
# count how many times each integer appears in the list
Counter([1, 3, 7, 7, 7, 7, 1])
```

[69]:
```
Counter({1: 2, 3: 1, 7: 4})
```

[70]:
```
# Counter acts similar to a dictionary since it's a subclass of dict class
c = Counter([1, 3, 7, 7, 7, 7, 1])
type(c)
```

[70]:
```
collections.Counter
```

[71]:
```
# pass a key to Counter and it returns a value
c[7]
```

[71]:
```
4
```

[72]:
```
# use update method to count more things
c.update([6, 6, 7])
c
```

[72]:
```
Counter({1: 2, 3: 1, 7: 5, 6: 2})
```

[73]:
```
# list elements from most common to least common
c.most_common()
```

[73]:
```
[(7, 5), (1, 2), (6, 2), (3, 1)]
```

[74]:
```
# sort it by the first element in each tuple
sorted(c.most_common())
```

[74]:
```
[(1, 2), (3, 1), (6, 2), (7, 5)]
```

Python Documentation: Counter

4.3 Range

[75]:
```
# call range class, get back a range object
range(4)
```

[75]:
```
range(0, 4)
```

[76]:
```
# pass it an integer, get back 0 through that number (exclusive of it)
list(range(4))
```

[76]:
```
[0, 1, 2, 3]
```

[77]:
```
# first number defines the start (inclusive), second defines the end (exclusive)
list(range(0, 4))
```

[77]:
```
[0, 1, 2, 3]
```

[78]:
```
list(range(2, 4))
```

[78]:
```
[2, 3]
```

[79]:
```
# use range to run a for loop a set number of times
for num in range(4):
    print(num)
```

```
0
1
2
3
```

4.4 Project Exercise 3

Exercises 2, 3, and 4 all relate to the following portion of the video: 5:10 to 6:57.

Define a function (with a docstring) called `count_first_digits()` that accepts one required argument called `end`, which is a positive integer.

Within the function, do the following:

- For each integer 1 through `end`, calculate the first digit of every number in that path.
- Tally up all of the first digits across all of those paths.
- Return a sorted list of tuples, in which the first element of each tuple is the digit and the second element is how many times that digit appeared.

For example, `count_first_digits(3)` should return `[(1, 5), (2, 2), (3, 1), (4, 1), (5, 1), (8, 1)]`.

Hint: You can start with an empty Counter.

Once your function is working, try running `count_first_digits(5000)` and see what you notice.

4.5 Solution to Exercise 3

[80]:
```
# count_first_digits will build on top of get_first_digits
help(get_first_digits)
```

```
Help on function get_first_digits in module __main__:

get_first_digits(start)
    Given a starting value, return the first digit of each number in the path.

    Args:
        start (int): Positive starting value
    Returns:
        List of integers representing the first digit of each number in the path
```

[81]:
```
# count_first_digits(3) should tally up the three lists below
get_first_digits(1)
```

[81]:
```
[1]
```

[82]:
```
get_first_digits(2)
```

[82]:
```
[2, 1]
```

[83]:
```
get_first_digits(3)
```

[83]:
```
[3, 1, 5, 1, 8, 4, 2, 1]
```

[84]:
```
# create an empty Counter
digits = Counter()
digits
```

[84]:
```
Counter()
```

[85]:
```
# use the update method to add this list to the Counter
digits.update(get_first_digits(1))
digits
```

[85]:
```
Counter({1: 1})
```

[86]:
```
digits.update(get_first_digits(2))
digits
```

[86]:
```
Counter({1: 2, 2: 1})
```

[87]:
```
digits.update(get_first_digits(3))
digits
```

[87]:
```
Counter({1: 5, 2: 2, 3: 1, 5: 1, 8: 1, 4: 1})
```

[88]:
```
# convert digits Counter into a sorted list of tuples
sorted(digits.most_common())
```

[88]:
```
[(1, 5), (2, 2), (3, 1), (4, 1), (5, 1), (8, 1)]
```

[89]:
```
# range assumes a start of 0
list(range(3))
```

[89]:
```
[0, 1, 2]
```

[90]:
```
# count_first_digits(3) would need the values 1, 2, 3
list(range(1, 4))
```

[90]:
```
[1, 2, 3]
```

[91]:
```
def count_first_digits(end):
    """
    Given an ending value, return a tally of all first digits of every number
    across every path generated by the starting values 1 through end␣
 ↪(inclusive).
```

```
    Args:
        end (int): Positive ending value
    Returns:
        Sorted list of tuples in which the first element of each tuple is the
        digit and the second element is how many times that digit appeared
    """

    digits = Counter()

    for num in range(1, end + 1):
        digits.update(get_first_digits(num))

    return sorted(digits.most_common())
```

[92]:
```
count_first_digits(3)
```

[92]:
```
[(1, 5), (2, 2), (3, 1), (4, 1), (5, 1), (8, 1)]
```

[93]:
```
# roughly follows Benford's Low
count_first_digits(5000)
```

[93]:
```
[(1, 116648),
 (2, 67801),
 (3, 45331),
 (4, 48426),
 (5, 31109),
 (6, 20746),
 (7, 21962),
 (8, 21867),
 (9, 19078)]
```

5 Project: Part 4

5.1 Zip

[94]:
```
# zip function loops over two or more iterables and "zips" them together
# returns an iterator of tuples
names = ['Homer', 'Marge', 'Lisa', 'Bart']
roles = ['dad', 'mom', 'daughter', 'son']
zip(names, roles)
```

[94]:
```
<zip at 0x7ff5f8c88c00>
```

[95]:
```
# convert into a list of tuples
list(zip(names, roles))
```

[95]:
```
[('Homer', 'dad'), ('Marge', 'mom'), ('Lisa', 'daughter'), ('Bart', 'son')]
```

[96]:
```
# convert into a dictionary
dict(zip(names, roles))
```

[96]:
```
{'Homer': 'dad', 'Marge': 'mom', 'Lisa': 'daughter', 'Bart': 'son'}
```

[97]:
```
# convert into a tuple of tuples
tuple(zip(names, roles))
```

[97]:
```
(('Homer', 'dad'), ('Marge', 'mom'), ('Lisa', 'daughter'), ('Bart', 'son'))
```

5.2 Project Exercise 4

Exercises 2, 3, and 4 all relate to the following portion of the video: 5:10 to 6:57.

Define a function (with a docstring) called `percentage_first_digits()` that accepts one required argument called `end`, which is a positive integer.

This function should convert the output of `count_first_digits()` from whole numbers into percentages of the total.

For example, the output of `count_first_digits(5000)` is this:

```
[(1, 116648),
 (2, 67801),
 (3, 45331),
 (4, 48426),
 (5, 31109),
 (6, 20746),
 (7, 21962),
 (8, 21867),
 (9, 19078)]
```

`percentage_first_digits(5000)` should instead return this:

```
[(1, 0.297),
 (2, 0.173),
 (3, 0.115),
 (4, 0.123),
 (5, 0.079),
 (6, 0.053),
 (7, 0.056),
 (8, 0.056),
 (9, 0.049)]
```

Here's how I suggest writing the function:

1. Save the output from `count_first_digits()`.
2. Get all of the numbers (1 through 9) into a list.
3. Get all of the counts into another list.
4. Calculate the total of those counts.

5. Calculate the percentage of that total for each count.
6. Zip the numbers with the percentages.

5.3 Solution to Exercise 4

[98]:
```
# list of tuples
output = count_first_digits(5000)
output
```

[98]:
```
[(1, 116648),
 (2, 67801),
 (3, 45331),
 (4, 48426),
 (5, 31109),
 (6, 20746),
 (7, 21962),
 (8, 21867),
 (9, 19078)]
```

[99]:
```
# get the first element of each tuple
nums = [t[0] for t in output]
nums
```

[99]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

[100]:
```
# get the second element of each tuple
counts = [t[1] for t in output]
counts
```

[100]:
```
[116648, 67801, 45331, 48426, 31109, 20746, 21962, 21867, 19078]
```

[101]:
```
# sum the numbers in an iterable
total = sum(counts)
total
```

[101]:
```
392968
```

[102]:
```
percentages = [count / total for count in counts]
percentages
```

[102]:
```
[0.29683841941328554,
 0.17253567720526863,
 0.11535544878972334,
 0.12323140815537142,
 0.07916420675474847,
 0.052793102746279595,
 0.05588750229026282,
 0.05564575232588913,
```

```
 0.04854848231917103]
```

[103]:
```
# round the output to 3 decimal places
percentages = [round(count / total, 3) for count in counts]
percentages
```

[103]:
```
[0.297, 0.173, 0.115, 0.123, 0.079, 0.053, 0.056, 0.056, 0.049]
```

[104]:
```
list(zip(nums, percentages))
```

[104]:
```
[(1, 0.297),
 (2, 0.173),
 (3, 0.115),
 (4, 0.123),
 (5, 0.079),
 (6, 0.053),
 (7, 0.056),
 (8, 0.056),
 (9, 0.049)]
```

[105]:
```
def percentage_first_digits(end):
    """
    Given an ending value, return the percentage of first digits of every number
    across every path generated by the starting values 1 through end␣
 ↪(inclusive).

    Args:
        end (int): Positive ending value
    Returns:
        Sorted list of tuples in which the first element of each tuple is the
        digit and the second element is the percentage that digit appeared
    """

    output = count_first_digits(end)
    nums = [t[0] for t in output]
    counts = [t[1] for t in output]
    total = sum(counts)
    percentages = [round(count / total, 3) for count in counts]
    return list(zip(nums, percentages))
```

[106]:
```
count_first_digits(5000)
```

[106]:
```
[(1, 116648),
 (2, 67801),
 (3, 45331),
 (4, 48426),
 (5, 31109),
```

```
 (6, 20746),
 (7, 21962),
 (8, 21867),
 (9, 19078)]
```

[107]:
```
percentage_first_digits(5000)
```

[107]:
```
[(1, 0.297),
 (2, 0.173),
 (3, 0.115),
 (4, 0.123),
 (5, 0.079),
 (6, 0.053),
 (7, 0.056),
 (8, 0.056),
 (9, 0.049)]
```

[108]:
```
count_first_digits(3)
```

[108]:
```
[(1, 5), (2, 2), (3, 1), (4, 1), (5, 1), (8, 1)]
```

[109]:
```
# still works even though not all digits are present
percentage_first_digits(3)
```

[109]:
```
[(1, 0.455), (2, 0.182), (3, 0.091), (4, 0.091), (5, 0.091), (8, 0.091)]
```

6 Project: Part 5

6.1 Line Plots with Matplotlib

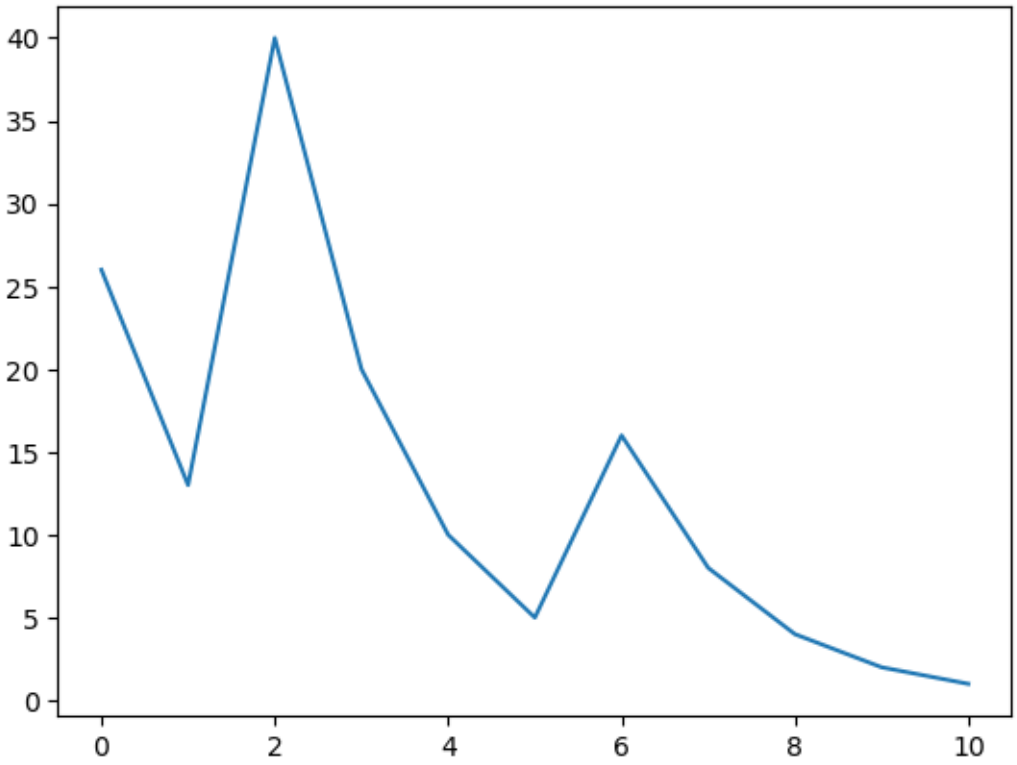
[110]:
```
# Matplotlib is not part of the Python standard library
import matplotlib.pyplot as plt
```

[111]:
```
# we want to plot this path
get_path(26)
```

[111]:
```
[26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

[112]:
```
# draw a line plot: x-axis is index of each list element, y-axis is value
plt.plot(get_path(26))
```

[112]:
```
[<matplotlib.lines.Line2D at 0x7ff5ea4570d0>]
```



[113]:
```
# Jupyter users: run this if you aren't seeing a plot
%matplotlib inline
```

[114]:
```
# Non-Jupyter users: run this if you aren't seeing a plot
plt.show()
```

[115]:
```
# path info matches the plot
print_path_info(26)
```

```
path: [26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
max: 40
steps: 10
```

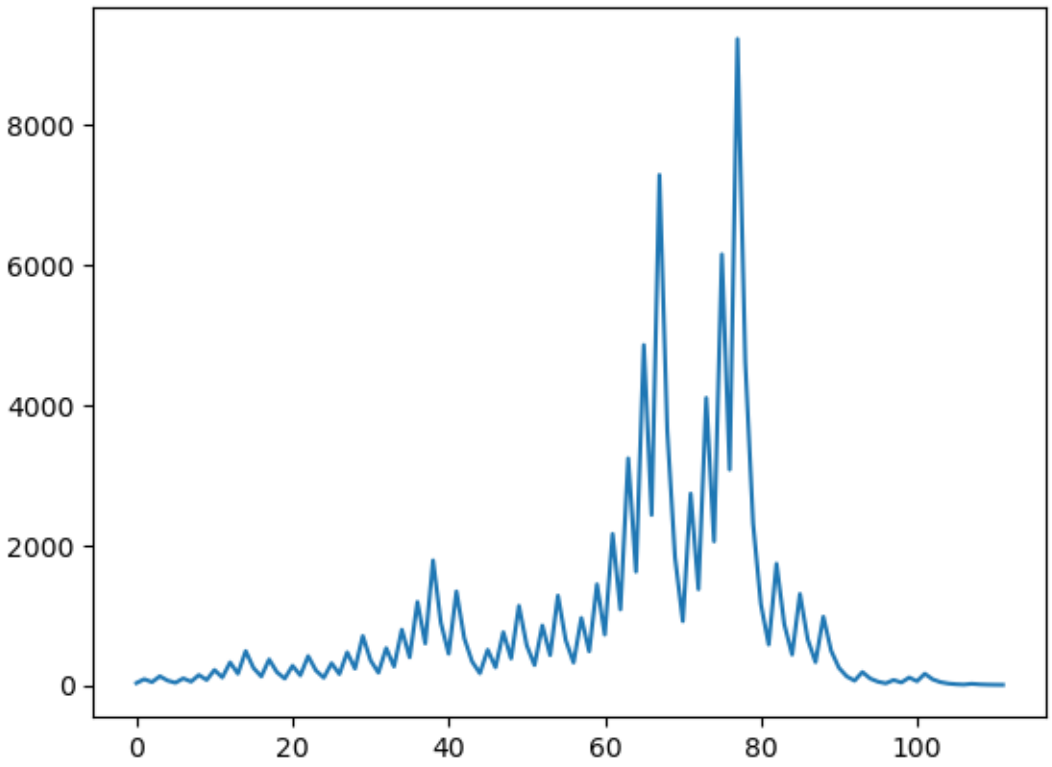
[116]:
```
print_path_info(27)
```

```
path: [27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242,
121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175,
526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251,
754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158,
1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051,
6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976,
488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16,
8, 4, 2, 1]
```

```
max: 9232
steps: 111
```

[117]:
```
# plot matches the path info
plt.plot(get_path(27))
```

[117]:
```
[<matplotlib.lines.Line2D at 0x7ff5f931d5a0>]
```



These are the plots we saw in the following portion of the video: 2:32 to 3:11.

Matplotlib Documentation: Installation

6.2 Bar Plots

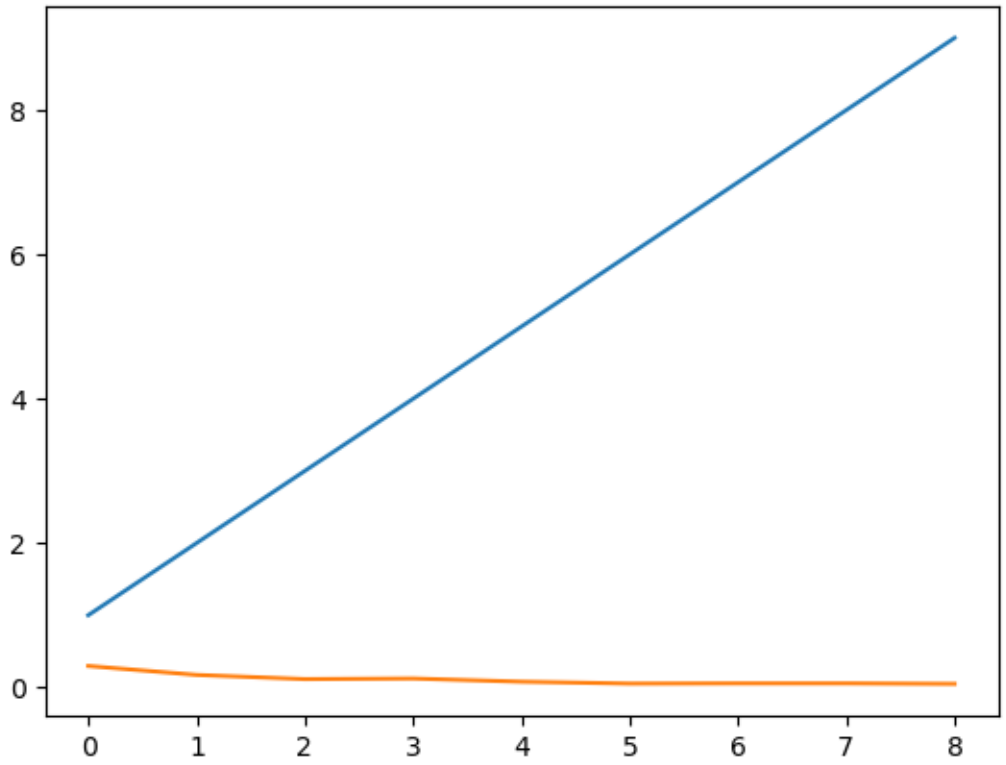
[118]:
```
# we want to plot this data
percentage_first_digits(5000)
```

[118]:
```
[(1, 0.297),
 (2, 0.173),
 (3, 0.115),
 (4, 0.123),
 (5, 0.079),
 (6, 0.053),
```

```
 (7, 0.056),
 (8, 0.056),
 (9, 0.049)]
```

[119]:
```
# two separate line plots is not what we wanted
plt.plot(percentage_first_digits(5000))
```

[119]:
```
[<matplotlib.lines.Line2D at 0x7ff5d8ada080>,
 <matplotlib.lines.Line2D at 0x7ff5d8ada0e0>]
```



[120]:
```
# we need two separate objects for the bar plot: nums and percentages
plt.bar()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[120], line 2
      1 # we need two separate objects for the bar plot: nums and percentages
----> 2 plt.bar()

TypeError: bar() missing 2 required positional arguments: 'x' and 'height'
```

[121]:
```
# list of tuples
output = percentage_first_digits(5000)
output
```

[121]:
```
[(1, 0.297),
 (2, 0.173),
 (3, 0.115),
 (4, 0.123),
 (5, 0.079),
 (6, 0.053),
 (7, 0.056),
 (8, 0.056),
 (9, 0.049)]
```

[122]:
```
# get the first element of each tuple
nums = [t[0] for t in output]
nums
```

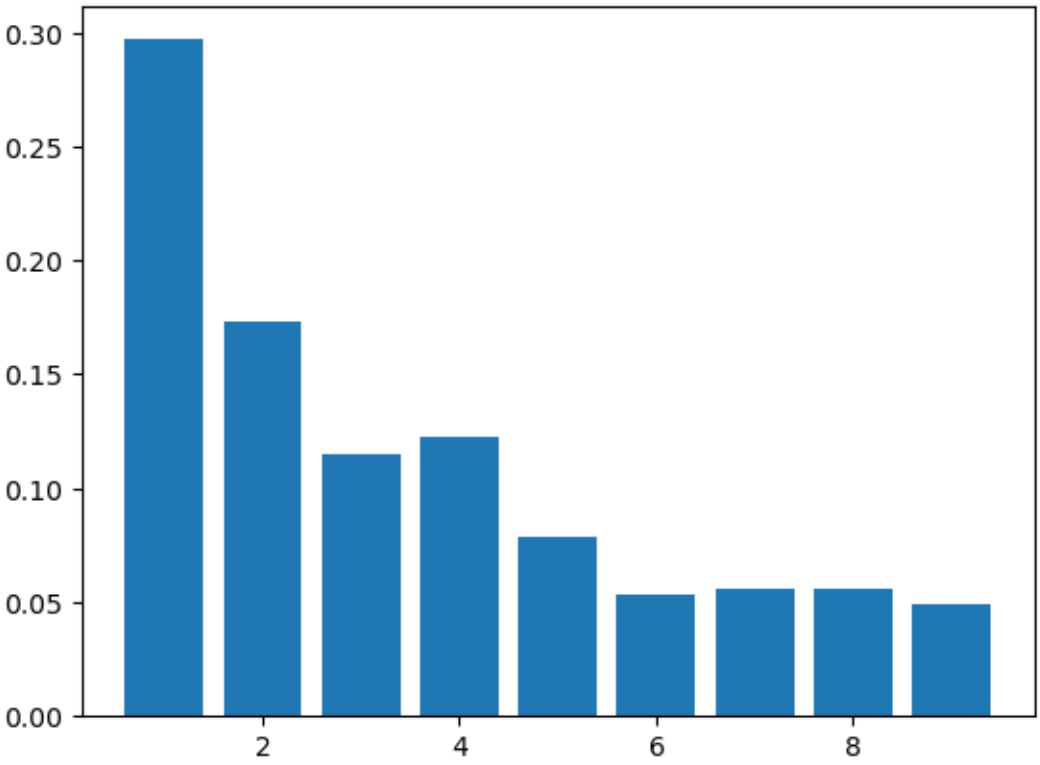
[122]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

[123]:
```
# get the second element of each tuple
percentages = [t[1] for t in output]
percentages
```

[123]:
```
[0.297, 0.173, 0.115, 0.123, 0.079, 0.053, 0.056, 0.056, 0.049]
```

[124]:
```
# draw a bar plot: x-axis is nums, y-axis is percentages
plt.bar(nums, percentages)
```

[124]:
```
<BarContainer object of 9 artists>
```



This is the plot we saw in the following portion of the video: 5:10 to 6:15.

6.3 Multiple Assignment

[125]:
```
# point on a graph
point = (5, -3)
```

[126]:
```
# use indexing to extract x and y coordinates
x = point[0]
y = point[1]
print(x)
print(y)
```

```
5
-3
```

[127]:
```
# better way is through multiple assignment (also called tuple unpacking)
x, y = point
print(x)
print(y)
```

```
5
-3
```

[128]:
```
# multiple assignment works with other iterables (but is most common with␣
↪tuples)
x, y = [10, 20]
print(x)
print(y)
```

```
10
20
```

[129]:
```
# multiple assignment is not limited to two objects
x, y, z = [10, 20, 30]
print(x)
print(y)
print(z)
```

```
10
20
30
```

[130]:
```
# number of elements must match the number of objects being assigned
x, y = [10, 20, 30]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[130], line 2
      1 # number of elements must match the number of objects being assigned
----> 2 x, y = [10, 20, 30]

ValueError: too many values to unpack (expected 2)
```

[131]:
```
# returns a tuple (by virtue of the comma)
def analyze_square(side):
    perimeter = side * 4
    area = side ** 2
    return perimeter, area
```

[132]:
```
analyze_square(10)
```

[132]:
```
(40, 100)
```

[133]:
```
# unpack the tuple into separate objects
p, a = analyze_square(10)
print(p)
print(a)
```

```
40
100
```

[134]:
```
shapes = {'triangle':3, 'square':4, 'pentagon':5}
```

[135]:
```
# looping through a dictionary loops through the keys
# during each loop, a shape name is implicitly assigned to "name"
for name in shapes:
    print(name)
```

```
triangle
square
pentagon
```

[136]:
```
# returns a list-like object containing the keys and values
shapes.items()
```

[136]:
```
dict_items([('triangle', 3), ('square', 4), ('pentagon', 5)])
```

[137]:
```
# implicit and multiple assignment from each tuple into "name" and "sides"
for name, sides in shapes.items():
    print(f'A {name} has {sides} sides.')
```

```
A triangle has 3 sides.
A square has 4 sides.
A pentagon has 5 sides.
```

Article & Video: Tuple unpacking (Python Morsels)

6.4 Unpacking into a Function Call

[138]:
```
shapes
```

[138]:
```
{'triangle': 3, 'square': 4, 'pentagon': 5}
```

[139]:
```
# returns a list-like object containing the values
shapes.values()
```

[139]:
```
dict_values([3, 4, 5])
```

[140]:
```
# convert it into an actual list
sides = list(shapes.values())
sides
```

[140]:
```
[3, 4, 5]
```

[141]:
```
# print each of the values separated by a space
print(sides[0], sides[1], sides[2])
```

```
3 4 5
```

[142]:
```
# unpack the iterable into a function call using the asterisk operator
# loop over "sides", get one item at a time, pass as separate arguments to print
```

```
print(*sides)
```

```
3 4 5
```

[143]:
```
# this use of the asterisk operator only works within a function call
*sides
```

```
  Cell In[143], line 2
    *sides
    ^
SyntaxError: can't use starred expression here
```

[144]:
```
# list of tuples
pairs = list(shapes.items())
pairs
```

[144]:
```
[('triangle', 3), ('square', 4), ('pentagon', 5)]
```

[145]:
```
# print each tuple separated by a space
print(*pairs)
```

```
('triangle', 3) ('square', 4) ('pentagon', 5)
```

[146]:
```
# pass the three tuples to zip, get back a list of two tuples
list(zip(*pairs))
```

[146]:
```
[('triangle', 'square', 'pentagon'), (3, 4, 5)]
```

6.5 Project Exercise 5

In the Bar Plots lesson, we created a list of tuples called `output`:

```
output = percentage_first_digits(5000)
output
[(1, 0.297),
 (2, 0.173),
 (3, 0.115),
 (4, 0.123),
 (5, 0.079),
 (6, 0.053),
 (7, 0.056),
 (8, 0.056),
 (9, 0.049)]
```

Then, we used list comprehensions to create the `nums` and `percentages` objects:

```
nums = [t[0] for t in output]
nums
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]

percentages = [t[1] for t in output]
percentages
[0.297, 0.173, 0.115, 0.123, 0.079, 0.053, 0.056, 0.056, 0.049]
```

Your task is to create `nums` and `percentages` from `output` in a single line of code by using the asterisk operator and multiple assignment.

6.6 Solution to Exercise 5

[147]:
```
# goal: aggregate first elements into "nums" and second elements into␣
↪"percentages"
output = percentage_first_digits(5000)
output
```

[147]:
```
[(1, 0.297),
 (2, 0.173),
 (3, 0.115),
 (4, 0.123),
 (5, 0.079),
 (6, 0.053),
 (7, 0.056),
 (8, 0.056),
 (9, 0.049)]
```

[148]:
```
# list of nine tuples has become a list of two tuples
list(zip(*output))
```

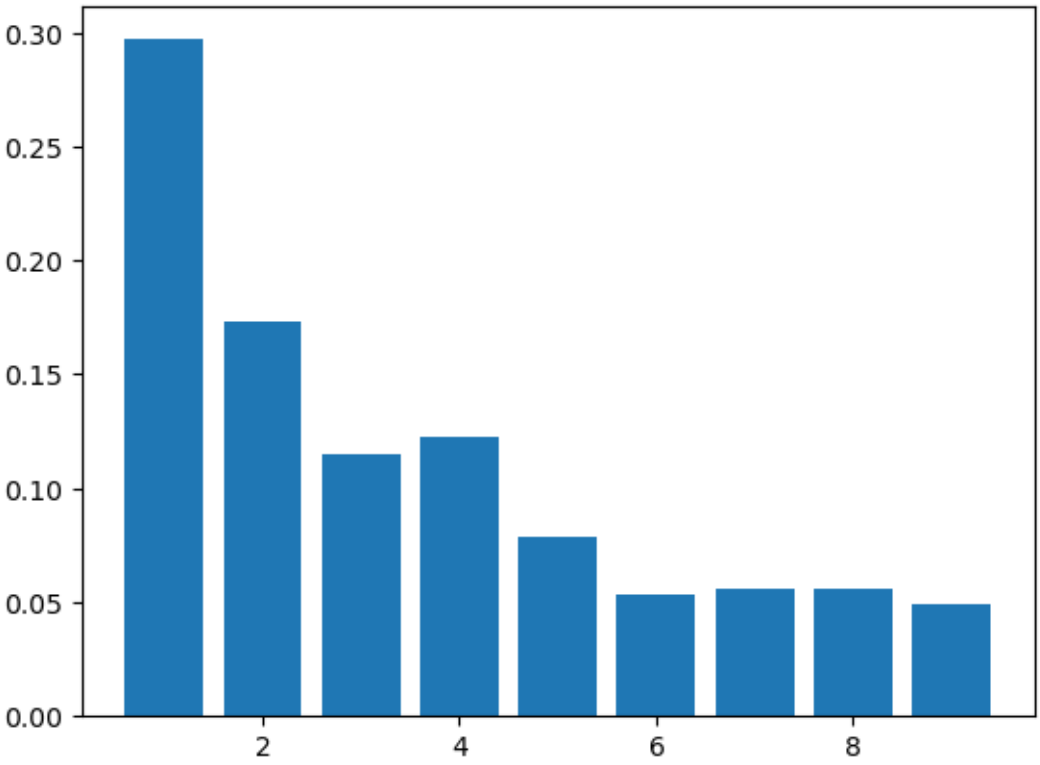
[148]:
```
[(1, 2, 3, 4, 5, 6, 7, 8, 9),
 (0.297, 0.173, 0.115, 0.123, 0.079, 0.053, 0.056, 0.056, 0.049)]
```

[149]:
```
# unpack the list using multiple assignment
nums, percentages = list(zip(*output))
print(nums)
print(percentages)
```

```
(1, 2, 3, 4, 5, 6, 7, 8, 9)
(0.297, 0.173, 0.115, 0.123, 0.079, 0.053, 0.056, 0.056, 0.049)
```

[150]:
```
# bar plot
plt.bar(nums, percentages)
```

[150]:
```
<BarContainer object of 9 artists>
```



7 Project: Part 6

7.1 Dictionary Comprehensions

[151]:
```
words = ['data', 'science', 'python']
```

[152]:
```
# create a list of word lengths
length = []
for word in words:
    length.append(len(word))
length
```

[152]: [4, 7, 6]

[153]:
```
# use a list comprehension instead
[len(word) for word in words]
```

[153]: [4, 7, 6]

[154]:
```
# create a dictionary of words and their lengths
word_length = {}
for word in words:
```

```
    word_length[word] = len(word)
word_length
```

[154]:
```
{'data': 4, 'science': 7, 'python': 6}
```

[155]:
```
# use a dictionary comprehension instead
{word:len(word) for word in words}
```

[155]:
```
{'data': 4, 'science': 7, 'python': 6}
```

7.2 Project Exercise 6

This exercise relates to the following portion of the video: 14:00 to 14:34.

Define a function (with a docstring) called `get_max_nums()` that accepts one required argument called `end`, which is a positive integer.

Within the function, do the following:

- For each integer 1 through `end`, calculate the path that starts with that integer.
- Return a dictionary in which each key is the starting number and each value is the maximum number reached during its path. (Use a dictionary comprehension for this.)

For example `get_max_nums(5)` should return `{1: 1, 2: 2, 3: 16, 4: 4, 5: 16}`.

7.3 Solution to Exercise 6

[156]:
```
# get_max_nums(5) would need the values 1, 2, 3, 4, 5
end = 5
list(range(1, end + 1))
```

[156]:
```
[1, 2, 3, 4, 5]
```

[157]:
```
# path for starting number of 3
get_path(3)
```

[157]:
```
[3, 10, 5, 16, 8, 4, 2, 1]
```

[158]:
```
# maximum number reached during the path
max(get_path(3))
```

[158]:
```
16
```

[159]:
```
# combine into a dictionary comprehension
# key is starting number, value is maximum number reached during the path
{num:max(get_path(num)) for num in range(1, end + 1)}
```

[159]:
```
{1: 1, 2: 2, 3: 16, 4: 4, 5: 16}
```

[160]:
```
def get_max_nums(end):
    """
    Given an ending value, return the maximum number reached during every path
    generated by the starting values 1 through end (inclusive).

    Args:
        end (int): Positive ending value
    Returns:
        Dictionary in which each key is the starting number and each value is␣
 ↪the
        maximum number reached during its path
    """

    return {num:max(get_path(num)) for num in range(1, end + 1)}
```

[161]:
```
get_max_nums(5)
```

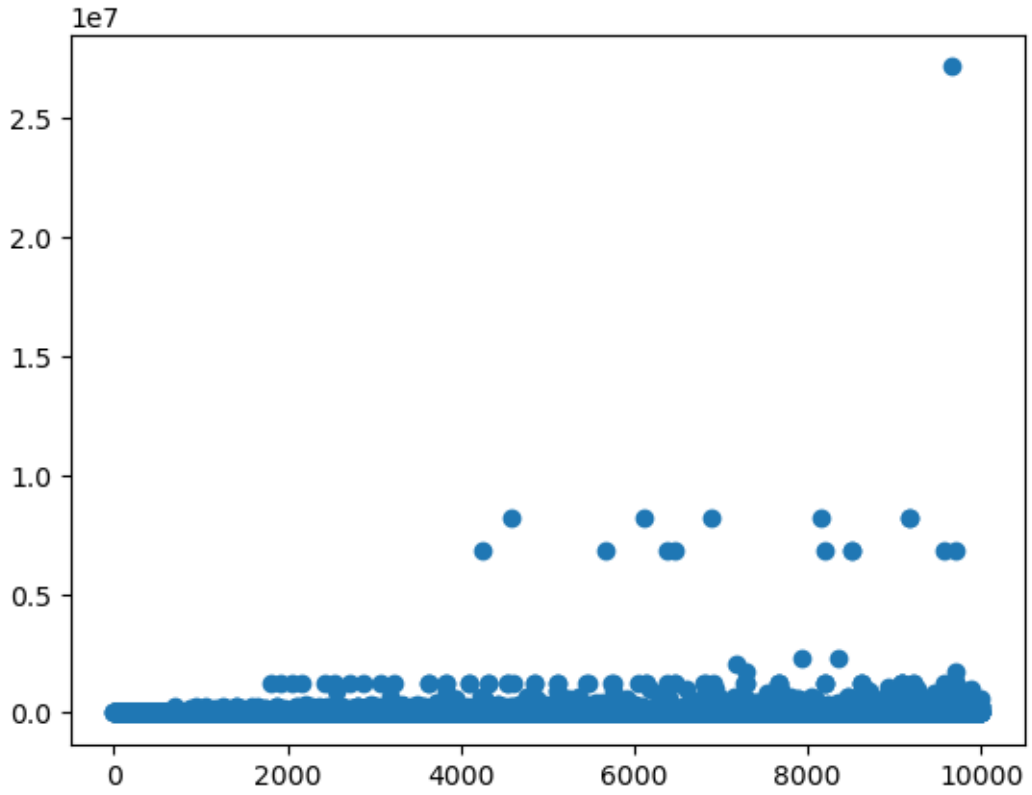
[161]:
```
{1: 1, 2: 2, 3: 16, 4: 4, 5: 16}
```

[162]:
```
max_nums = get_max_nums(10000)
```

[163]:
```
# underscores within numbers are ignored
max_nums = get_max_nums(10_000)
```

[164]:
```
# scatterplot: x-axis is starting numbers, y-axis is maximum numbers reached
plt.scatter(x=max_nums.keys(), y=max_nums.values())
```

[164]:
```
<matplotlib.collections.PathCollection at 0x7ff5d8b2e320>
```

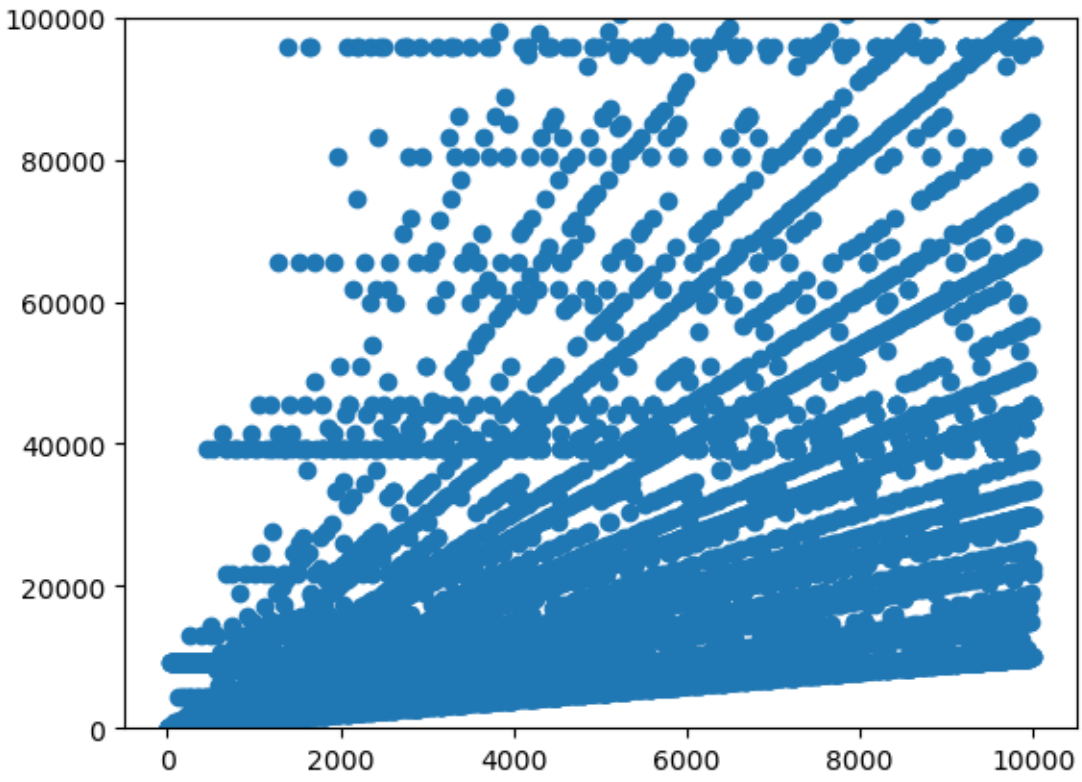


[165]:
```
# outlier in the upper right corner of the plot
max(get_path(9663))
```

[165]: 27114424

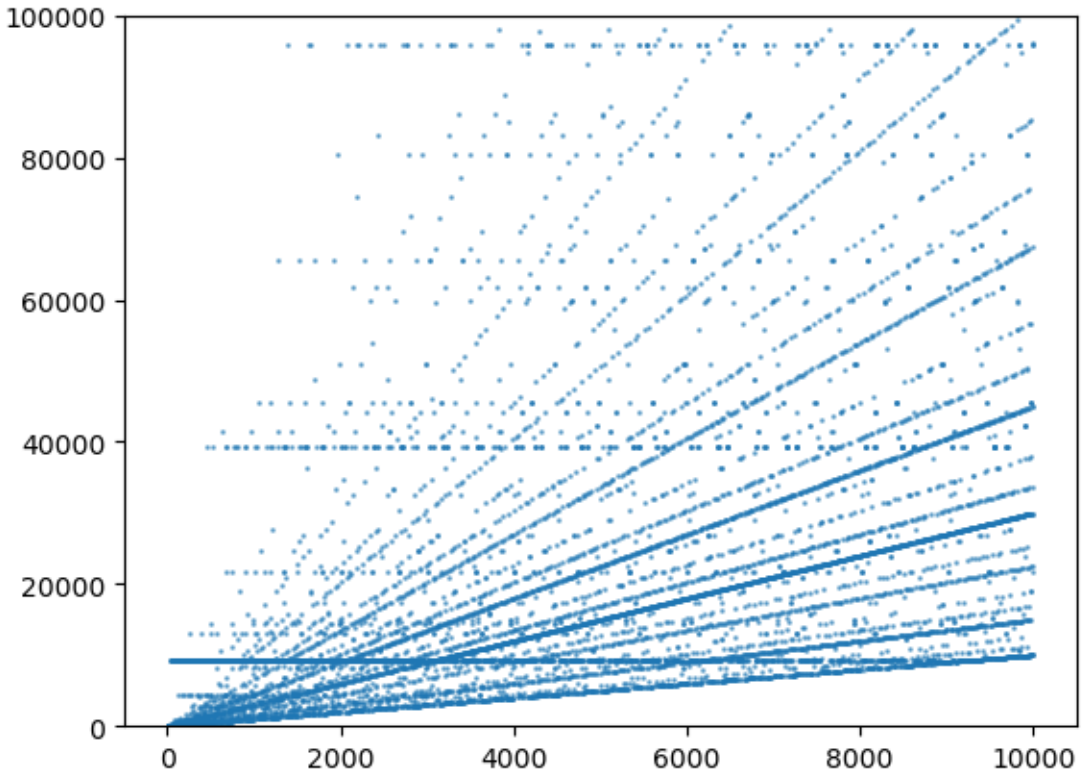
[166]:
```
# set the y-axis limit to 100,000
plt.scatter(x=max_nums.keys(), y=max_nums.values())
plt.ylim(0, 100_000)
```

[166]: (0.0, 100000.0)



[167]:
```
# set "s" to make the points smaller, set "alpha" to add transparency
plt.scatter(x=max_nums.keys(), y=max_nums.values(), s=1, alpha=0.5)
plt.ylim(0, 100_000)
```

[167]: (0.0, 100000.0)



8 Project: Part 7

8.1 Membership Operators

[168]:
```
x = 2
```

[169]:
```
# comparisons return either True or False
x > 0
```

[169]: True

[170]:
```
x == 0
```

[170]: False

[171]:
```
x != 0
```

[171]: True

[172]:
```
# logical "or" returns True if either comparison is True
x == 0 or x == 1
```

[172]: False

[173]:
```
x == 0 or x == 1 or x == 2
```

[173]: True

[174]:
```
# preferable to use the membership operator "in"
# returns True if x is a member of the list
x in [0, 1]
```

[174]: False

[175]:
```
x in [0, 1, 2]
```

[175]: True

[176]:
```
# in operator works with tuples
x in (0, 1, 2)
```

[176]: True

[177]:
```
# in operator works with strings
'science' in 'data science'
```

[177]: True

[178]:
```
'I' in 'team'
```

[178]: False

[179]:
```
word_length
```

[179]: {'data': 4, 'science': 7, 'python': 6}

[180]:
```
# in operator checks dictionary keys
'science' in word_length
```

[180]: True

[181]:
```
# in operator doesn't check dictionary values
7 in word_length
```

[181]: False

[182]:
```
# not operator does negation
not True
```

[182]: False

[183]:
```
not False
```

[183]: True

[184]:
```
# "not in" membership operator is the opposite of the "in" operator
'science' not in word_length
```

[184]: False

[185]:
```
'math' not in word_length
```

[185]: True

[186]:
```
# membership operators are useful for conditional statements and while loops
if x in [0, 1]:
    print('x is a binary number')
```

[187]:
```
if x not in [0, 1]:
    print('x is not a binary number')
```

```
x is not a binary number
```

8.2 Project Exercise 7

This exercise relates to the following portion of the video: 14:34 to 15:09.

Modify the `get_path()` function (and its docstring) so that the `start` argument can be any integer.

- If `start` is positive, the path should stop once it reaches 1.
- If `start` is negative, the path should stop once it reaches -1, -5, or -17.
- If `start` is zero, the path should only contain the integer 0.

For example:

`get_path(10)` should return `[10, 5, 16, 8, 4, 2, 1]`

`get_path(-3)` should return `[-3, -8, -4, -2, -1]`

`get_path(-9)` should return `[-9, -26, -13, -38, -19, -56, -28, -14, -7, -20, -10, -5]`

`get_path(-200)` should return `[-200, -100, -50, ..., -68, -34, -17]`

`get_path(0)` should return `[0]`

8.3 Solution to Exercise 7

[188]:
```
# start with our existing get_path function
def get_path(start):
    """
    Given a starting value, return the path.

    Args:
        start (int): Positive starting value
```

```
    Returns:
        List of integers representing the path
    """

    nums = [start]
    num = start

    while num > 1:

        if num % 2 == 0:
            num = num // 2
        else:
            num = num * 3 + 1

        nums.append(num)

    return nums
```

[189]:
```
def get_path(start):
    """
    Given a starting value, return the path.

    Args:
        start (int): Positive or negative starting value
    Returns:
        List of integers representing the path
    """

    nums = [start]
    num = start

    while num not in [1, -1, -5, -17, 0]:  # list of stopping values

        if num % 2 == 0:
            num = num // 2
        else:
            num = num * 3 + 1

        nums.append(num)

    return nums
```

[190]:
```
# ends in a loop of 4, 2, 1
get_path(10)
```

[190]:
```
[10, 5, 16, 8, 4, 2, 1]
```

[191]:
```
# ends in a loop of -2, -1
get_path(-3)
```

[191]:
```
[-3, -8, -4, -2, -1]
```

[192]:
```
# ends in a loop of -14, -7, -20, -10, -5
get_path(-9)
```

[192]:
```
[-9, -26, -13, -38, -19, -56, -28, -14, -7, -20, -10, -5]
```

[193]:
```
# ends in a loop of 18 numbers from -50 to -17
get_path(-200)
```

[193]:
```
[-200,
 -100,
 -50,
 -25,
 -74,
 -37,
 -110,
 -55,
 -164,
 -82,
 -41,
 -122,
 -61,
 -182,
 -91,
 -272,
 -136,
 -68,
 -34,
 -17]
```

[194]:
```
get_path(0)
```

[194]:
```
[0]
```