chapter_0_part2

July 25, 2021

0.1 Activity 17: Modelling a Fair Coin

Let's say your friend gives you a fair coin.

Let $P(H)$ = a, the probability of getting heads when you flip the coin. Of course, this also makes the probability of getting tails, $P(T)$, equal to 1 - a.

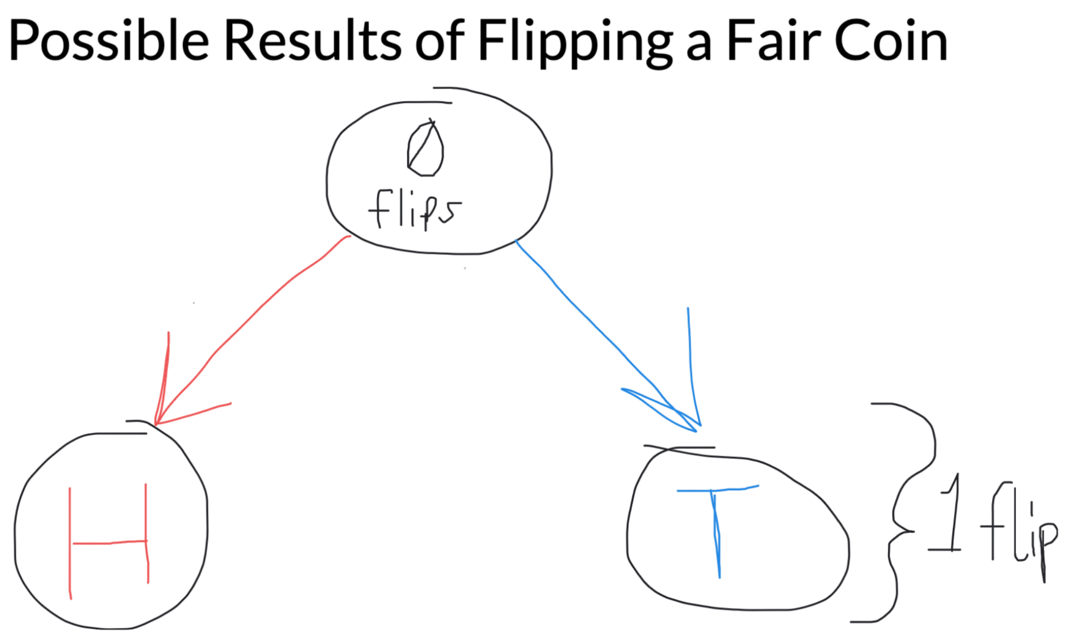
Now, say we toss this coin three times. What is the probability that we get heads on two out of the three flips?

Calculate this value by hand, and then please write Python code to verify your answer on 1,000 coin flips (hint: use the `random` module)!

0.1.1 The Mathematical Solution

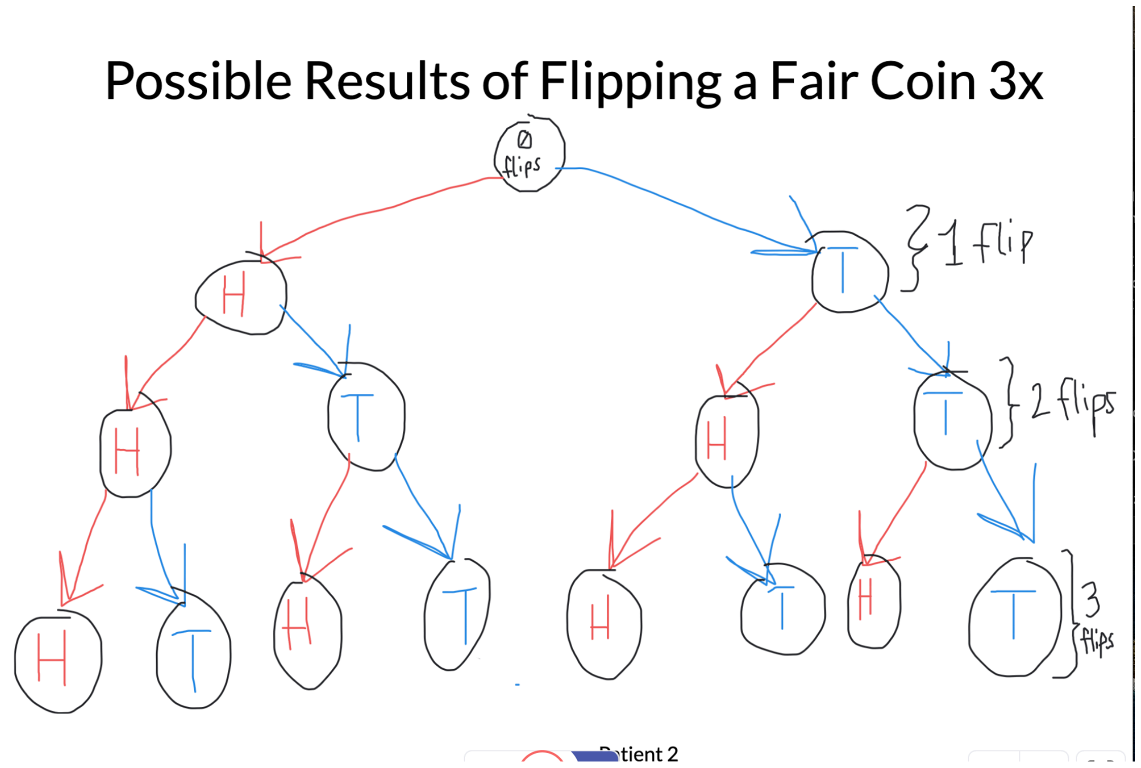
Let's start by breaking down the probabilities of 1 coin flip. As you may already know, this only has 2 outcomes, each of which are equally likely: heads or tails.

To give a visual, observe the following diagram which shows the possible outcomes for 1 random coin toss:



So how many ways can we have 2 heads on 3 of the coin flips?

As more and more flips happen, we will continue to have either heads or tails. Essentially, this means our original diagram can be expanded, and to show all the possibile results we could have:



As you can see, there are 8 total permutations of what 3 faces our coin will land on. As well, there are 3 in particular which have 2 heads: HHT, HTH, and THH (try to spot them in the diagram above, in case you are not convinced)!

Since all of the 8 permutations are equally likely to occur, we can then conclude our probability is `3/8`, or 0.375.

0.1.2 The Python Solution

To model this experiment in code, one approach we can take is:

```
Pseudocode for Modelling a Fair Coin
1. Examine the results of 1000 trials, where each trial is defined tossing the coin 3 times.
2. Then, we can keep track of an integer called `occurences` to record the number of trials in
3. Finally, we can compute the answer by dividing the number of trials where our desired outco
```

Before actually writing this function though, you may still have several questions. For example, how are we supposed to generate random events in Python?

Mini-Lesson: How to Simulate Random Events in Python The following code snippet provides a straightforward process you can follow for modelling probability. For more on the `random.choices` function, you may read more on the Python documentation as you wish.

[1]:
```
from random import choices

# A: first, define the possible outcomes that can occur
event = ['H', 'T']
# B: then, define their respective probabilities
weights = [0.3, 0.7]
# C: run the simulation!
for _ in range(10):
    # D: print the results, to verify it follows our assigned probabilities
    print(choices(event, weights))
```

```
['H']
['T']
['T']
['H']
['T']
['T']
['H']
['T']
['H']
['H']
```

Code Implementation From above, we know modelling a random event needs 3 parameters: 1. The possible events that can happen 2. The probabilities of each of those events 3. The number of trials to simulate

For this solution, let's leave out 1, since we already know this is for a fair coin.

Therefore, our function only needs to take in the remaining two input arguments: 1. The probability of landing on heads (since we can already compute the $P(T)$ with that information), 2. And the number of trials to run.

Using the pseudocode above and what we now know about the `random` module, we can therefore implement something like the code below to see if our mathematical answer is truly correct:

[2]:
```
def compute_proba(probability_heads, trials):
    # A: set the parameters of the random event
    events = ['H', 'T']
    weights = [probability_heads, 1 - probability_heads]
    # B: count how many times the desired outcome occurs
    occurences = 0
    for _ in range(trials):
        # C: check the results after 1 trial
        trial_results = [
          choices(events, weights)[0],  # 1st coin flip
          choices(events, weights)[0],  # 2nd coin flip
          choices(events, weights)[0]   # 3rd coin flip
        ]
        if trial_results.count("H") == 2:
            occurences += 1
    # D: return the probability
    return (occurences / trials)
```

Test Out the Python Solution

[3]:
```
a = 0.5 # the probability of landing on heads
num_trials = 1000
print(compute_proba(a, num_trials))  # computed probability after 1,000 trials
print(3*(a**2)*(1-a)) # expected probability - does it match with the above?
```

```
0.364
0.375
```

0.2 Activity 18: Remove Duplicates

You are given a list of numbers, `nums`. How would you implement Python code to return only the unique values from `nums`?

Note: Here are some assumptions that may help you on this problem: 1. You can return the unique numbers using any data type in Python. 2. They can be in any order 3. But, the input list itself is *immutable* - meaning, it must remain unchanged

0.2.1 Example Input

[4]:
```
nums = [2, 3, 2, 4, 1, 2]
```

0.2.2 Solution 1: Using a set

[5]:
```
def remove_duplicates_1(nums):
    return set(nums)
```

Test Out Solution 1

[6]:
```
print(remove_duplicates_1(nums))
```

```
{1, 2, 3, 4}
```

0.2.3 Solution 2: Using a `for` loop

[7]:
```
def remove_duplicates_2(nums):
    no_duplicates = []
    for num in nums:
        if num not in no_duplicates:
            no_duplicates.append(num)
    return no_duplicates
```

Test Out Solution 2

[8]:
```
print(remove_duplicates_2(nums))
```

```
[2, 3, 4, 1]
```

0.2.4 Solution 3: Using an Auxiliary `list`

In this scenario we can mark all the elements we discover are duplicates as `None`, and then make a new list that doesn't include those elements:

[9]:
```python
def remove_duplicates_3(nums):
    # A: make a copy of numbers
    numbers = nums.copy()
    # B: mark the duplicates
    for index in range(len(nums)):
        # C: if this number appears again, then it's a duplicate
        rest_of_list = numbers[index+1:]
        if numbers[index] in rest_of_list:
            numbers[index] = None
    # D: return all non-duplicates
    return [num for num in numbers if num is not None]
```

Test Out Solution 3

[10]:
```python
print(remove_duplicates_3(nums))
```

```
[3, 4, 1, 2]
```

0.2.5 Solution 4: Using the `del` keyword

This solution is just like above, except we instead just delete the duplicate values from the list, rsther than making a new one.

[11]:
```python
def remove_duplicates_4(nums):
    # A: make a copy of the input
    numbers = nums.copy()
    # B: delete the duplicates from the list
    original_length = len(numbers)
    num_checks = 0
    index = 0
    while num_checks < original_length:
        num_checks += 1
        rest_of_list = numbers[index+1:]
        # C: duplicate found
        if numbers[index] in rest_of_list:
            del numbers[index]
        # D: duplicate not found, so move on to the next indx
        else:
            index += 1
    # E: return only the non-duplicate numbers
    return numbers
```

Test Out Solution 4

[12]:
```python
print(remove_duplicates_4(nums))
```

```
[3, 4, 1, 2]
```

0.2.6 Solution 5: Using Map Reduce

In this solution, we'll combine elements of what we've learned previously about the `map` and `reduce` functions, as well as how the `set` data type works.

The first step is to make a new list from `nums`, where each individual element goes inside its own `set`. We can do this quickly using the `map` function:

`iterable_sets = map(lambda x:{x}, nums)`

Then, we can essentially boil down the `nums` list to just its unique elements, by using the `reduce` function.

Python provides a `set.union` method: simply put, this allows us to efficiently combine all the unique elements found in two sets together, into one.

The `set.union` is the function we will use in our call to `reduce`, so we must first define a function to wrap around it - this can be done easily using the `lambda` keyword:

`collect_all_unique = lambda set1, set2: set1.union(set2)`

Finally, we can go ahead and call the `reduce` function on our new list, as well as our function:

`return reduce(collect_all_unique, iterable_sets)`

And voilà! The function you see below is an example of "Map Reduce", a programming concept that's very popular in data engineering, and you will probably see more and more as you work with larger datasets.

[13]:
```
# Solution 5:
from functools import reduce

def remove_duplicates_map_reduce(nums):
    # A: place each element in nums into a set by itself
    iterable_sets = map(lambda x:{x}, nums)
    # B: define a function to combine sets - collecting only unique numbers
    collect_all_unique = lambda set1, set2: set1.union(set2)
    # C: decompose the iterable down into a single set
    return reduce(collect_all_unique, iterable_sets)
```

Test Out Solution 5

[14]:
```
print(remove_duplicates_map_reduce(nums))
```

```
{1, 2, 3, 4}
```

0.3 Activity 19: Combine Dictionaries

You are given two Python dictionaries, `d1` and `d2`. How would you implement Python code to combine all the key-value pairs from `d1` and `d2` together?

Note: If there are any keys in the `d2` that are already present in the `d1`, then place the corresponding values together into a list, and make that the new value in the combined dictionary. Also, both the `d1` and `d2` are must not be modified by your solution.

0.3.1 Example Inputs

```
Inputs:
d1 = {'a': 10, 'b': 20, 'c': 30}
d2 = {'b': 40, 'd': 50}

Output:
{
  'a': 10,
  'b': [20, 40], <-- notice how we combined the values for the 'b' key!
  'c': 50,
  'd': 50
}
```

[15]:
```python
d1 = {'a':10, 'b':20, 'c':30}
d2 = {'b': 40, 'd':50}
```

0.3.2 Solution Code

[16]:
```python
def combine_dicts(d1, d2):
    # A: init a third dict
    combined = d1.copy()
    # B: add the key-value pairs from the second
    for key_d2, value_d2 in d2.items():
        # C: if an value has already been mapped to this key, include both
        if key_d2 in combined:
            value_d1 = d1[key_d2]
            combined[key_d2] = [value_d1, value_d2]
        # D: otherwise, just add the key-value pair
        else:
            combined[key_d2] = value_d2
    # E: return the new dict
    return combined
```

Test Out the Solution

[17]:
```python
print(combine_dicts(d1, d2))
```

```
{'a': 10, 'b': [20, 40], 'c': 30, 'd': 50}
```

0.4 Activity 20: Flatten a Matrix

You are given a 2D list, `matrix`. How would you write code to flatten `matrix` into a 1-dimensional Python `list`?

0.4.1 Example Input

What we mean here by "flatten a matrix" is to collect all the elements into a 1D list.

In this 1D list, elements should be in the same order as one would see if they were reading the `matrix` row-wise - look at the example below to see this in action:

```
Input:
matrix = [
  [8, 2, 3],
  [9, 1, 9],
  [5, 4, 1]
]

Output:
[8, 2, 3, 9, 1, 9, 5, 4, 1]
```

[18]:
```python
matrix = [[8, 2, 3], [9, 1, 9], [5, 4, 1]]
```

0.4.2 Solution 1: Nested `for` loops

[19]:
```python
def flatten_1(matrix):
    # A: init the 1-dimensional array
    ls = []
    # B: add all the elements to it row-wise
    for row in matrix:
        for element in row:
            ls.append(element)
    # C: return the populated list
    return ls
```

Test Out Solution 1

[20]:
```python
print(flatten_1(matrix))
```

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

0.4.3 Solution 2: `for` loop with `list.extend`

Although the above solution works, we can save ourselves a few keystrokes by alternatively using the `list.extend` method, in order to quickly load all the elements from the `row` variable into our 1D list.

Note: Solution 2 is not necessarily execute faster than Solution 1 - we only mean it is "quicker" in that it is easier to write out 1 `for` loop rather than two:

[21]:
```python
def flatten_2(matrix):
    # A: init the 1-dimensional array
    row_vector = []
    # B: like before, add all the elements
    for row in matrix:
        row_vector.extend(row)
    # C: return the populated list
    return row_vector
```

Test Out Solution 2

[22]:
```
print(flatten_2(matrix))
```

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

0.5 Activity 21: Compute the Trace

The *trace of a square matrix* is a concept from linear algebra. According to Wikipedia, it is “defined to be the sum of elements on the main diagonal (from the upper left to the lower right) of A.” Here, the Wikipedia editors simply use A as a variable to represent our square matrix.

You are given a 2D list of numbers, `matrix`. Note that `matrix` is guaranteed to have the same number of rows as it has columns, because it is a square. How would you write Python code to compute its trace?

0.5.1 Example Input

[23]:
```
matrix = [[8, 2, 3], [9, 1, 9], [5, 4, 1]]
```

0.5.2 Solution: Using `enumerate`

[24]:
```
def trace(matrix):
    sum_diagonal = 0
    for index_row, row in enumerate(matrix):
        sum_diagonal += matrix[index_row][index_row]
    return sum_diagonal
```

Test Out Solution Code

[25]:
```
print(trace(matrix))
```

```
10
```

0.6 Activity 22: Transpose of a Matrix

Another popular matrix operation is to compute the *transpose*. We flip the elements in the matrix over the diagonal producing a new matrix.

You are given a 2D list of numbers, `matrix`. How would you write Python code to compute the tranpose of `matrix`?

0.6.1 Example Input

Put another way, the transpose matrix is just what the original `matrix` would be, if you were to read the elements in said `matrix` going up and down vertically, rather than left to right horizontally.

Therefore, if we had a matrix like the following:

```
matrix = [
    [8, 2, 3],
    [9, 1, 9],
    [5, 4, 1]
]
```

Then we would expect its transpose to be the following output:

```
transposed_matrix = [
    [8, 9, 5],  # first column
    [2, 1, 4],  # second column
    [3, 9, 1]   # third column
]
```

[26]:
```
matrix = [[8, 2, 3], [9, 1, 9], [5, 4, 1]]
```

0.6.2 Solution

[27]:
```
def transpose(matrix):
    # A: init the transpose matrix using the dims of the original
    transposed_matrix = [[] for _ in range(len(matrix[0]))]
    # B: iterate over each column
    for col_index in range(len(matrix[0])):
        # C: collect the elements in this column
        for row in matrix:
            transposed_matrix[col_index].append(row[col_index])
    # D: return the transpose of the matrix
    return transposed_matrix
```

Note: in the solution above, it is also possible to implement step A in the following way:

```
transpose = [[]]*len(matrix[0])
```

However, using a list comprehension with the `range()` function is the more efficient way to initialize our matrix, and thus is preferred.

Test Out The Solution

[28]:
```
print(transpose(matrix))
```

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

0.7 Activity 23: Compressing Data

You are given a list of tuples, `employees`, which represents a company's records of its employees (see below cell for an example input). Each tuple object contains two strings: the first is for the employee's department (e.g. `'Sales', Marketing, Accounting, etc.`), and the second is for their full name.

Some of the records in `employees` share the same department, and some of the names appear more than once. How would you implement Python code to group all the employees in the same department together, and to remove the duplicate names from appearing more than once?

Hint 1: this is a great place to use a ***dict****!*

Hint 2: but how would you handle multiple employees being in the same department?

0.7.1 Example Input

[29]:
```
employees = [
    ('Sales', 'John Doe'),
    ('Sales', 'Martin Smith'),
    ('Accounting', 'Jane Doe'),
    ('Marketing', 'Elizabeth Smith'),
    ('Marketing', 'Elizabeth Smith'),
    ('Marketing', 'Adam Doe'),
    ('Marketing', 'Adam Doe'),
    ('Marketing', 'Adam Doe')
]
```

0.7.2 Solution 1: Using the `dict` type, with `list` values

This solution follows a pattern that is probably becoming more and more familiar to you - here it is in pseudocode:

```
How to Decompress Data in Python using a "dict":
A: init a new dictionary
B: iterate over our data (i.e. key-value pairs):
    a: if I haven't seen this key before, map it to a new iterable containing the value
    b: if I have seen the key before, then add the value to the iterable (that's already been r
C: return the dictionary!
```

And here is the actual code:

[30]:
```
def compress_data_1(employees):
    # A: init a new dictionary
    compressed = dict()
    # B: iterate over our data (i.e. key-value pairs):
    for department, employee in employees:
        # a: if I haven't seen this key before,
        if department not in compressed:
            # map it to a new iterable containing the value
            compressed[department] = [employee]
        # b: if I have seen the key before,
        else:  # department in compressed
            # then add the value to the iterable
            dept_employees = compressed[department]
            if employee not in dept_employees:
                dept_employees.append(employee)
            # (that's already been mapped to this key)
            compressed[department] = dept_employees
    # C: return the dictionary!
    return compressed
```

Test Out Solution 1

[31]:
```
print(compress_data_1(employees))
```

```
{'Sales': ['John Doe', 'Martin Smith'], 'Accounting': ['Jane Doe'], 'Marketing':
['Elizabeth Smith', 'Adam Doe']}
```

0.7.3 Solution 2: Using `dict` with `set` values

This solution is an alternative to the above. It still follows the same pattern as before; however, rather than using a `list` object to encapsulate the values in our dictionary, we instead use the `set` data type.

Note that in this approach, the `set` data type allows us to prevent duplicate names from being added to our new dictionary *much* more efficently than when we were using the `list` data type!

[32]:
```
def compress_data_2(employees):
    # A: init a new dictionary
    compressed = dict()
    # B: iterate over our data (i.e. key-value pairs):
    for department, employee in employees:
        # a: if I haven't seen this key before,
        if department not in compressed:
            # map it to a new iterable (in this case, a set) w/ the value
            compressed[department] = { employee}
        # b: if I have seen the key before,
        else:  # department in compressed
            # then add the value to the set (auto-checks for duplicates!)
            compressed[department].add(employee)
    # C: return the dictionary!
    return compressed
```

Test Out Solution 2

[33]:
```
print(compress_data_2(employees))
```

```
{'Sales': {'John Doe', 'Martin Smith'}, 'Accounting': {'Jane Doe'}, 'Marketing':
{'Elizabeth Smith', 'Adam Doe'}}
```

0.7.4 Solution 3: Using `defaultdict`

Fortunately, the pattern you have practiced here is so common, the Python development community created a *subclass* of the `dict` data type just to make it easier to implement.

To do this, we can simply instantiate an object of the `defaultdict` class, and pass in the name of one of Python's iterable data types as an argument to the constructor, such as `list` or `set`. This basically tells Python what kind of data type we can the values in our `defaultdict` to have.

Finally, we can go ahead and pass the key-value pairs to our dictionary, as you can see below.

Note: the `defaultdict` class can behave very differently, based on which data type you pass to it, (or if you even pass one at all). The data type you pass therefore depends on the behavior you need for your use case. You may read more about these behaviors of `defaultdict` and more, in the Python documentation.

[34]:
```python
from collections import defaultdict

def compress_data_3(employees):
    # A: init a new dictionary
    compressed = defaultdict(set)
    # B: iterate over our data (i.e. key-value pairs):
    for department, employee in employees:
        # C: this line will take care of the rest
        compressed[department].add(employee)
    # D: return the dictionary!
    return compressed
```

Test Out Solution 3 This was by far the quickest solution to code - nevertheless, the output will not be exactly the same though:

[35]:
```python
print(compress_data_3(employees))
```

```
defaultdict(<class 'set'>, {'Sales': {'John Doe', 'Martin Smith'}, 'Accounting':
{'Jane Doe'}, 'Marketing': {'Elizabeth Smith', 'Adam Doe'}})
```

0.8 Activity 24: Fibonacci Calculator

The Fibonacci Sequence is a famous concept from mathematics, originally developed to model the growth of populations. Please read the Wikipedia article here to get a basic understanding of how to calculate numbers in the Fibonacci Sequence (aka "Fibonacci numbers") by hand.

In this exercise, we ask you: how would you write a Python program to implement the `n`-th Fibonacci number?

0.8.1 Example Input

In the Fibonacci Sequence, `n` can be any nonnegative integer. For example:

[36]:
```python
n = 20  # fib(20) equals 6,765 by hand
```

0.8.2 Solution 1: Using Recursion

The sequence is predefined for a few base cases - for example, the 0th Fibonacci number is 0, and the 1st Fibonacci number is 1.

However after that, the `n`-th Fibonacci number is simply defined as the sum of the previous two numbers. Therefore, this problem lends itself well to using recursion:

[37]:
```python
def fib_recursive(n):
    # Base Cases
    if n == 0:
        return 0
    elif n == 1:
        return 1
    # Recursive Case
    else:
        return fib_recursive(n - 1) + fib_recursive(n - 2)
```

Test Out Solution 1

[38]:
```
print(fib_recursive(n))
```

```
6765
```

0.8.3 Solution 2: Using a `dict`

As the value of `n` increases, it can also be useful to use a data structure to keep track of previous Fibonacci numbers. This can allow our function to take less time than if it was using recursion, because we are able to access the previous two Fibonaaci numbers in constant time.

See an example of this below, using a `dict`:

[39]:
```
def fib_dictionary(n):
    # Base Cases
    dic_fib = {0:0, 1:1}
    # Recursive Cases - like before, we can go up to the nth number:
    for i in range(2, n + 1):
        dic_fib[i] = dic_fib[i - 1] + dic_fib[i - 2]
    return dic_fib[n]
```

Test Out Solution 2

[40]:
```
print(fib_dictionary(n))
```

```
6765
```

0.8.4 Solution 3: Using a `list`

Similar to using a `dict`, we could also use a `list`:

[41]:
```
def fib_list(n):
    # Base Cases:
    y = [0, 1]
    # Recursive Cases
    for _ in range(2, n + 1):
        # here, we can get the previous two numbers using list splicing
        new_fib_value = sum(y[-2:])
        y.append(new_fib_value)
    return y[-1] # last element of y is our desired number
```

Test Out Solution 3

[42]:
```
print(fib_list(n))
```

```
6765
```

0.8.5 Solution 4: Using local variables

While the previous two solutions work, they might cost us too much space if our input `n` value is large enough.

Fortunately, one insight into this problem is we don't actually need to store all the previous Fibonacci numbers we calculate - rather, we can just keep track of the previous two using local variables:

[43]:
```
def fib_local_vars(n):
  # Base Cases - these are equal to fib(0) and fib(1)
  base1, base2 = 0, 1
  # Caculate our nth value,
  prev1 = base1
  prev2 = base2
  # move our two previous values to just before fib(n)
  for _ in range(2, n):
    temp = prev2
    prev2 = prev1 + prev2
    prev1 = temp
  # finally, return fib(n - 2) + fib(n - 1)
  return prev1 + prev2
```

Test Out Solution 4

[44]:
```
print(fib_local_vars(n))
```

```
6765
```

0.8.6 Solution 5: Using a Generator

At this point, you may be wondering: "how could we possibly optimize our function any further? I thought we had already reached the best time and space complexity!"

Intro to Generators

Well, the truth is although our function has reached optimal performance (as measured by Big O), in can still perform slowly on large datasets. This is because our code executes *synchronously* - that is to say, we have to wait for it to finish completely, before getting the return value.

In the real world, this means we can still run into issues on massively sized input data. For example, let's say you have a function to preprocess a dataset of 870,000 color images, each of which is 2532x1170 pixels. Once preprocessed, we will use these images to train a machine learning model.

Wouldn't you prefer being able to train the model on the first say, 100 images, as soon as they've been preprocessed, rather than having to wait for the function to work through all 870,000 images first? In the first scenario, this means we want to make our function *asynchronous* - we allow it to execute at the same time as another task is happening (in this case, our model training). In short, this allows both tasks to take less time overall.

How Generators Work This is where generators come in - generators make code asynchronous, because they utilize *lazy execution*. Instead of going through all the iterations at one time, they wait for us to call them first - and even then, they only evaluate the value of the next iteration. *Another key difference:* after that iteration is over, the generator *deallocates* the memory it used to compute that value. This means the generator never takes up more memory than what's needed for a single iteration.

How to Define a Generator Function

To define a generator in Python is easy - just use the `yield` keyword in the body of the `for` or `while` loop:

[45]:
```
def fib_generator(n):
    num1, num2 = 0, 1
    for _ in range(1, n + 2):
        # this returns the current Fibonacci number asynchronously
        yield num1
        # move on to the next
        num1, num2 = num2, num1 + num2
```

Test Out Solution 5 How to Return Values from a Generator

The `yield` keyword in our function above removes the need to use the `return` keyword - however, the two do not operate exactly the same way.

This is because *generators are designed to be used like any other iterable* - they compute a series of values, rather than a singular value. Therefore, if we simply tried to print the return value of the function above, we would actually just get a `generator` object.

Instead, we can use the following options:

Option 1: Using the `next()` function

In this option we iterate over the values generated by the `fib_generator` using an outer `for` loop. In a given iteration, we are able to get the actual value that was computed by using the built-in `next()` function.

[46]:
```
# A: iterate over the first n Fibonacci numbers
fib_nums = fib_generator(n)
for i in range(n + 1):
    # B: using next() is what moves the generator on to the next Fibonacci num
    next_num = next(fib_nums)
    # C: if we have reached the n-th Fibonacci number, print it!
    if i == n:
        print(next_num)
```

```
6765
```

Option 2: Directly Using a `for` loop

In practice, we normally don't use the `next()` function when working with generators, since it is already called under the hood when we use a `for` loop.

Therefore, we can also do the following:

[47]:
```
# A: iterate over the first n Fibonacci numbers
i = 0
for fib_num in fib_generator(n):  # << this implictly calls the next() function
    # B: if we have reached the n-th Fibonacci number, print it!
    if i == n:
        print(fib_num)
    # C: otherwise, move on
    i += 1
```

```
6765
```

For more on generators, you may read on in the Python documentation.

0.9 Activity 25: String Interpolation

Strings are a popular data type to use for communicating messages to the user - however, we often need to embed other kinds of data into our `str` objects. This is called *string interpolation*.

For example, let's say you are given 2 `list` objects:

1. One lists the names of several people;
2. One lists their respective ages

How would you implement Python code to greet each person, and inform them of their age?

0.9.1 Example Input

For our solution, let's say you start with a template message such as the following:

```
Hello <name>, your age is: <age>.
```

Using this template, an example input and output for our function will look like the following:

```
Input:
names = ["Khiem", "Alex", "Mike"]
ages = [24, 25, 26]

Output:
Hello Khiem, your age is: 24.
Hello Alex, your age is: 25.
Hello Mike, your age is: 26.
```

This activity essentially is asking you to construct `str` objects that contain both the name and age of each person. This requires you to embed an `int` into a `str`; and this is what exemplifies a common use of string interpolation.

[48]:
```
names = ["Khiem", "Alex", "Mike"]
ages = [24, 25, 26]
```

0.9.2 Solution 1: Format String

One of the easiest and most readable ways to perform string interpolation in Python is using a *format-string*, sometimes called "f-string" for short.

Observe how this technique allows us to easily combine data from our two lists into 1 string called `message`, in the body of the `for` loop below:

[49]:
```python
def inform_ages_1(names, ages):
  for name, age in zip(names, ages):
      print(f"Hello {name}, your age is: {age}.")
```

Test Out Solution 1

[50]:
```python
inform_ages_1(names, ages)
```

```
Hello Khiem, your age is: 24.
Hello Alex, your age is: 25.
Hello Mike, your age is: 26.
```

0.9.3 Solution 2: Using `str.format()`

Before format-strings were introduced in Python 3.6, a common way to perform string interpolation was through the `str.format()` method. You may read more about its syntax in the Python documentation.

[51]:
```python
def inform_ages_2(names, ages):
  for name, age in zip(names, ages):
      print("Hello {}, your age is: {}.".format(name, age))
```

Test Out Solution 2

[52]:
```python
inform_ages_2(names, ages)
```

```
Hello Khiem, your age is: 24.
Hello Alex, your age is: 25.
Hello Mike, your age is: 26.
```

0.10 Activity 26: String Concatenation

One common string operation is joining two or more strings together, to form a larger one - this is known as *concatenation*.

Let's challenge your string concatenation skills: for example, say you are given a list of strings, `strings`, and another string that is just a single word - `word`.

How would you implement Python code to concatenate the elements in `strings` together, with the `word` string placed in-between every pair of elements in the output?

0.10.1 Example Input

```
Input:
strings = [
  'How was work?',
  'day, ',
  'dollar.'
]
word = ' Another '

Output:
How was work? Another day, Another dollar

Explanation:
The word ' Another ' is placed in between the 1st and 2nd elements in the list, as well as the
```

[53]:
```python
strings = ['How was work?', 'day,', 'dollar.']
word = ' Another '
```

0.10.2 Solution 1: Using String Addition

Strings in Python are just another kind of iterable. Therefore, Python lets us concatenate strings with the + operator, and that itself will create a larger string:

[54]:
```python
def concatenate_strings_1(strings, word):
    # A: init the output string
    message = ''
    # B: construct the message
    for index, string in enumerate(strings):
        # add the next string to the output
        message = message + string
        # also add the word (if we haven't reached the last element yet)
        if index != len(strings) - 1:
            message = message + word
    # C: display the output string
    print(message)
```

Test Out Solution 1

[55]:
```python
concatenate_strings_1(strings, word)
```

```
How was work? Another day, Another dollar.
```

0.10.3 Solution 2: Using `str.join()`

The `str` type in Python also comes with a method to make concatenation much easier: `str.join()`.

This method is preferred to using addition. Not only is it more Pythonic, `str.join()` is also costs less computational resources. You may read more about its syntax in the Python documenation.

Please see below an example of how this works:

[56]:
```python
def concatenate_strings_2(strings, word):
    # construct + display the output string
    print(word.join(strings))
```

Test Out Solution 2

[57]:
```
concatenate_strings_2(strings, word)
```

```
How was work? Another day, Another dollar.
```

0.11 Activity 27: Age Record

You are given two lists of equal size:

1. `names`: this contains the names of several people, and
2. `ages`: this has the ages of those people. The values are ordered such that for any index `i`, `ages[i]` is the age of the person with the name located at `names[i]`.

How would you implement Python code to map each person's name to their age, all in one data structure?

0.11.1 Example Input

This activity is a classic example of when to use a `dict` object:

```
Input:
ages = ['15', '27', '67', '102']
names = ['Jessica', 'Daniel', 'Edward', 'Oscar']

Output:
{
  'Jessica': '15',
  'Daniel': '27',
  'Edward': '67',
  'Oscar': '102'
}
```

[58]:
```
ages = ['15', '27', '67', '102']
names = ['Jessica', 'Daniel', 'Edward', 'Oscar']
```

0.11.2 Solution 1: `zip()` Revisited

In a separate activity, we saw how using the `zip()` function made it easier to iterate over multiple lists at the same time, and how this is meaningful when those lists have values at each index that correspond to one another.

Now let's see how using `zip()` can also make it easier to construct a Python dictionary:

[59]:
```
def age_record(names, ages):
    records = {}
    for age, name in zip(ages, names):
        records[name] = age
    return records
```

Test Out Solution 1

```
[60]: age_record(names, ages)
```

```
[60]: {'Jessica': '15', 'Daniel': '27', 'Edward': '67', 'Oscar': '102'}
```

0.12 Activity 28: Group Anagrams

You are given a list of strings, `strings`. How would you implement Python code to group together the strings in this list that are anagrams of one another?

Assume that strings only contain lowercase English letters.

0.12.1 Example Input

In this activity, please return a 2D list.

```
Input:
strings = [ 'eat', 'ate', 'apt', 'pat', 'tea', 'now' ]

Output:
[
  ['eat', 'ate', 'tea'],
  ['apt', 'pat'],
  ['now']
]

Explanation:
Each index of the output contains a list of words, which all contain the same distribution of l
```

```
[61]: strings = ['eat', 'ate', 'apt', 'pat', 'tea', 'now']
```

0.12.2 Solution 1: Brute Force

Note: this solution also introduces the `itertools.chain` function - this is a utility in Python that allows us to efficiently "flatten" an multidimensional array.

For example, if we have a 2-dimensional array such as the following:

```
[
  [1, 2, 3],
  [4, 5, 6]
]
```

Then it would return back an iterator that would output `[1, 2, 3, 4, 5, 6]`. For more on this function, please see the Python documentation.

This can be useful when we want to see if a certain value exists along any axis in a matrix, as you will see in the following example:

```
[62]: import itertools
      from collections import Counter
```

```
def group_anagrams_1(strings):
    # A: init output matrix with maximum no. of groupings
    groups = [[] for _ in range(len(strings))]
    # B: group strings that are anagrams
    for index in range(len(strings)):
        first_word = strings[index]
        # check every subsequent string
        for second_index in range(index, len(strings)):
            second_word = strings[second_index]
            # ensure the strings are anagrams
            if Counter(first_word) == Counter(second_word):
                # and that the second string is not already in a group
                if second_word not in list(itertools.chain(*groups)):
                    groups[index].append(second_word)
    # C: return the groups of anagrams
    return [element for element in groups if element != []]
```

Test Out Solution 1

[63]:
```
print(group_anagrams_1(strings))
```

```
[['eat', 'ate', 'tea'], ['apt', 'pat'], ['now']]
```

0.12.3 Solution 2: Using Unicode Values

This solution takes less time asymptotically than the one above. There were **two main issues** with the former solution:

Issue 1: Duplicated Work in Solution 1 we needed a nested `for` loop to compare each string, with every subsequent string.

However, in this solution we introduce `ord()`, a built-in function in Python that gives the corresponding Unicode value for a given character. For example:

- `ord('a')` returns 97…
- `ord('b')` returns 98…
- `ord('c')` returns 99…

…and so on and so forth. So just as strings themselves represent *arrays of characters*, we can now represent strings as *arrays of the Unicode values* that correspond to each of their characters.

How exactly do we do this? We can implement this as a helper function called `encode()`, which takes in a string, and computes a histogram of the Unicode values represented by a given word using a `list` with 26 index positions (one for each letter in the English alphabet):

[64]:
```
def encode(word):
    '''Computes a histogram of Unicode values represented in a string.'''
    histogram = [0 for _ in range(26)]
    for letter in word:
        # find an index for this letter's Unicode value
        histogram[ord(letter) - ord('a')] += 1
    # cast the histogram as a tuple, so it can be used a key in a dictionary
    return tuple(histogram)
```

So how does `encode()` speed us up? Now that we have a standardized way to represent the strings, we can avoid a nested `for` loop to compare them all with each other upfront. Instead, we can simply focus on encoding each string one at a time. Finally, we can add them to a `defaultdict`, which essentially takes over making the groups for us.

Issue 2: Bottlenecks Speaking of `defaultdict` - why is this so much better than using a 2D array to represent the groups as seen in Solution 1?

This is because in Solution 1, we needed to repeatedly check whether the `second_word` we found to be an anagram had already been added to one of the groups or not using `itertools.chain`. This is a linear time operation, and we can be avoid it with a better understanding of the problem.

Under the hood, a `defaultdict` (as well as the related `dict` and `Counter` types) uses a `hash` function to add new values - and this function outputs the same value for the same input.

Recall that anagrams are just strings with the same distribution of characters. The `encode()` function gives us a way to express these distributions with the same output from the `hash` function (also called the *hash code*):

[65]:
```
# Observe: the hash() function computes the same values for encoded anagrams
anagram1, anagram2 = 'cat', 'tac'
hist1, hist2 = encode(anagram1), encode(anagram2)
print(f'The first string has the following hash code: {hash(hist1)}.')
print(f'And its anagram string has an identical hash code: {hash(hist2)}.')
```

```
The first string has the following hash code: 3800677941862037860.
And its anagram string has an identical hash code: 3800677941862037860.
```

Ultimately, this means that we are already guaranteed that anagrams will end up in the same group in our `defaultdict`, because only those strings will have equivalent hash codes.

0.12.4 Bringing It All Together

The full implementation of Solution 2 follows below:

[66]:
```
from collections import defaultdict

def group_anagrams_2(strings):
    # A: init dictionary to store groups
    groups = defaultdict(list)
    # B: place each string in one of the groups
    for word in strings:
        # a: enode the string as a histogram of Unicode values
        unicode_value_histogram = encode(word)
        # b: place this word in the appropiate group
        groups[unicode_value_histogram].append(word)
    # C: return the groups of anagrams
    return list(groups.values())
```

Test Out Solution 2

```
[67]: print(group_anagrams_2(strings))
```

```
[['eat', 'ate', 'tea'], ['apt', 'pat'], ['now']]
```

0.13 Activity 29: Modelling Exponential Growth

Consider a person working in a company. Their base salary is $115K per year. At that company, their salary increases at a rate of 3% annually.

How would you implement Python code to show what amount this person's salary will be each year, for the next 10 years they work at the company? You may round your answers to the nearest thousand.

0.13.1 Example Input

Recall that the formula to calculate the quantity of some variable P is the following:

$P(t) = P0\ (1 + r) * t$

Where: - $P(t)$ = the value of P in a given year - $P0$ = the initial value of P - r = the percentage at which P increases each year - t = the year in which P reaches the value of $P(t)$

To soldify this concept, try to calculate what salary our employee will have after 10 years of working at the company:

```
Inputs:
base_salary = 115
rate = 0.03
years = 10

Output:
$155,000

Explanation
Substituting the variables into the formula above, we get the following:
P(10) =  115 (1.03) * 10
P(10) = 154.55, which approximates to $155K
```

```
[68]: base_salary = 115
      rate = 0.03
      years = 10
```

0.13.2 Solution 1: Using a `for` loop

While the above example shows how the calculate the person's salary for a given year, we can also calculate it for each year using a `for` loop. The salary that the person earns each year is essentially just what they earned last year, plus an additional 3% on whatever amount that last year's salary was:

[69]:
```python
def salary_increase(base_salary, rate, years):
    # A: init a list of salaries this person will have
    salaries = list()
    # B: calculate the salaries they will have each yr
    prev_salary = base_salary
    for year_num in range(years):
        # a: the salary paid to the person is rounded to the nearest thousand
        next_salary = (1 + rate) * prev_salary
        paid_salary = round(next_salary, 0)
        salaries.append(paid_salary)
        # b: prepare for next year's calculation
        prev_salary = next_salary
    # C: return the list of salaries
    return salaries
```

Test Out Solution 1

[70]:
```python
print(salary_increase(115, 0.03, 10))
```

```
[118.0, 122.0, 126.0, 129.0, 133.0, 137.0, 141.0, 146.0, 150.0, 155.0]
```

0.14 Activity 30: Function Composition

At this point, we presume you understand how functions work: - they take in values, - and output values.

When the output of one function becomes the input for another, we have a special name for it: *function composition*.

In this activity, you are given a function `f`, an initial value `a0`, and some number of iterations to perform, `m`. How would you write Python code to show how the value of `a` changes as it is passed to `f`, over `m` iterations?

Return the values $a1$, $a2$, $a3$, and $a4$ using a `list`. Please round your answers to 8 decimal places.

0.14.1 Example Input

In mathematics, whenever we have a function that takes in the output of another as a parameter, then we call that a *compound function*.

In this activity, `f` is a compound function (note that you could also call it a *recursive function*, since it takes the output of itself as a parameter).

Consider the function $f(x) = \frac{x+n}{2x}$. Let's see how an initial value of `1.0` changes over `4` iterations:

```
Input:
f = lambda x: (x+n/x)/2  # our function
n = 2
a0 = 1.0  # the initial value
m = 4  # the number of iterations

Output
```

```
[1.5, 1.41666667, 1.41421569, 1.41421356]

Explanation:
This output is the result of the following:
[
    round(output, 8) for output in (
        f(a0),
        f(f(a0)),
        f(f(f(a0))),
        f(f(f(f(a0))))
    )
]
```

[71]:
```python
f = lambda x: (x+n/x)/2  # our function
n = 2
a0 = 1.0  # the initial value
m = 4  # the number of iterations
```

0.14.2 Solution 1: Using a Generator

While you could simply implement the code in the "Explanation" section above, it wouldn't be very readable, and it would stop working the moment the value of `m` changes.

Therefore, it is much more preferable to create a function to complete this activity. In a separate activity we can do something like this using a `generator` function in Python, via the `yield` keyword:

[72]:
```python
def compound_function(f, a, m):
    # compute the value of a after each iteration
    for _ in range(m):
        # apply the function
        a = f(a)
        # output the result (don't forget to round!)
        yield round(a, 8)
```

Test Out Solution 1 Recall we can use a `for` loop to iterate over the values returned by a `generator` object:

[73]:
```python
for output in compound_function(f, a0, m):
    print(output)
```

```
1.5
1.41666667
1.41421569
1.41421356
```

0.14.3 Special Note: Convergence

Did you notice anything special about $f(x)$ in Activity 30? *Hint:* try increasing the number of iterations `m`, and trying out different values of `n` - then see if you spot a familiar pattern!

The Answer: As the number of iterations `m` increases, the outputs of the compound function $f(x)$ converges to $\sqrt{n}$.

We can even verify this by comparing it with the result of the `numpy.sqrt` function!

```
[74]: import numpy as np
      np.sqrt(n)
```

```
[74]: 1.4142135623730951
```