DSCI 222: Python Code Collaboration, List Comprehension and Memory Management

Ignacio Segovia-Dominguez

West Virginia University

www.IgnacioSD.com

*Disclaimer. Slides may contain a mix of material from multiple authors: Ignacio Segovia-Dominguez (WVU), AWS, Anaconda, Google Colab, Microsoft, Medium.com, GeeksForGeeks.org, RealPython.com, Simplilearn.com, SaturnCloud, DigitalOcean, Python Docs.

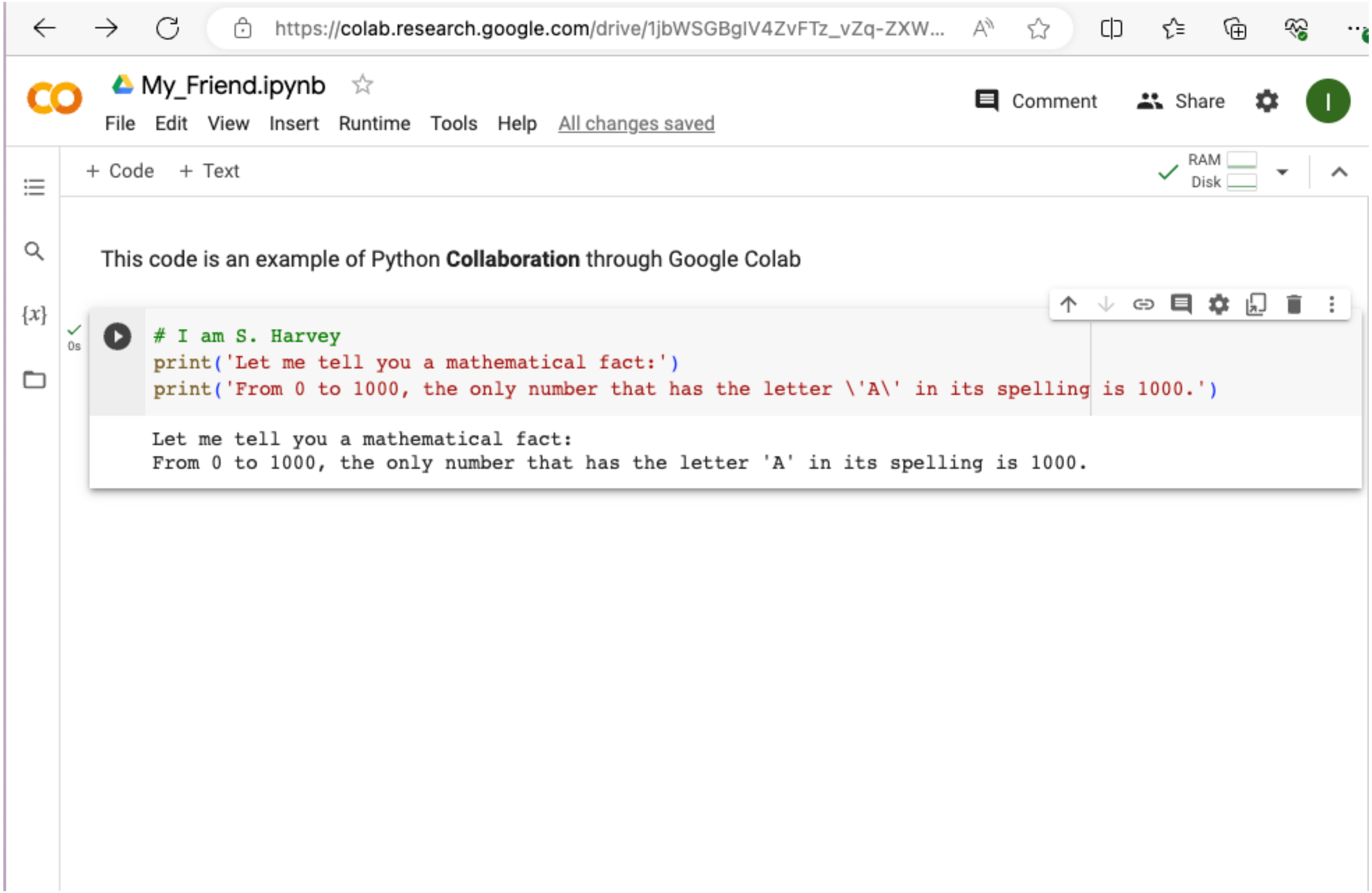
Collaboration while writing Python code

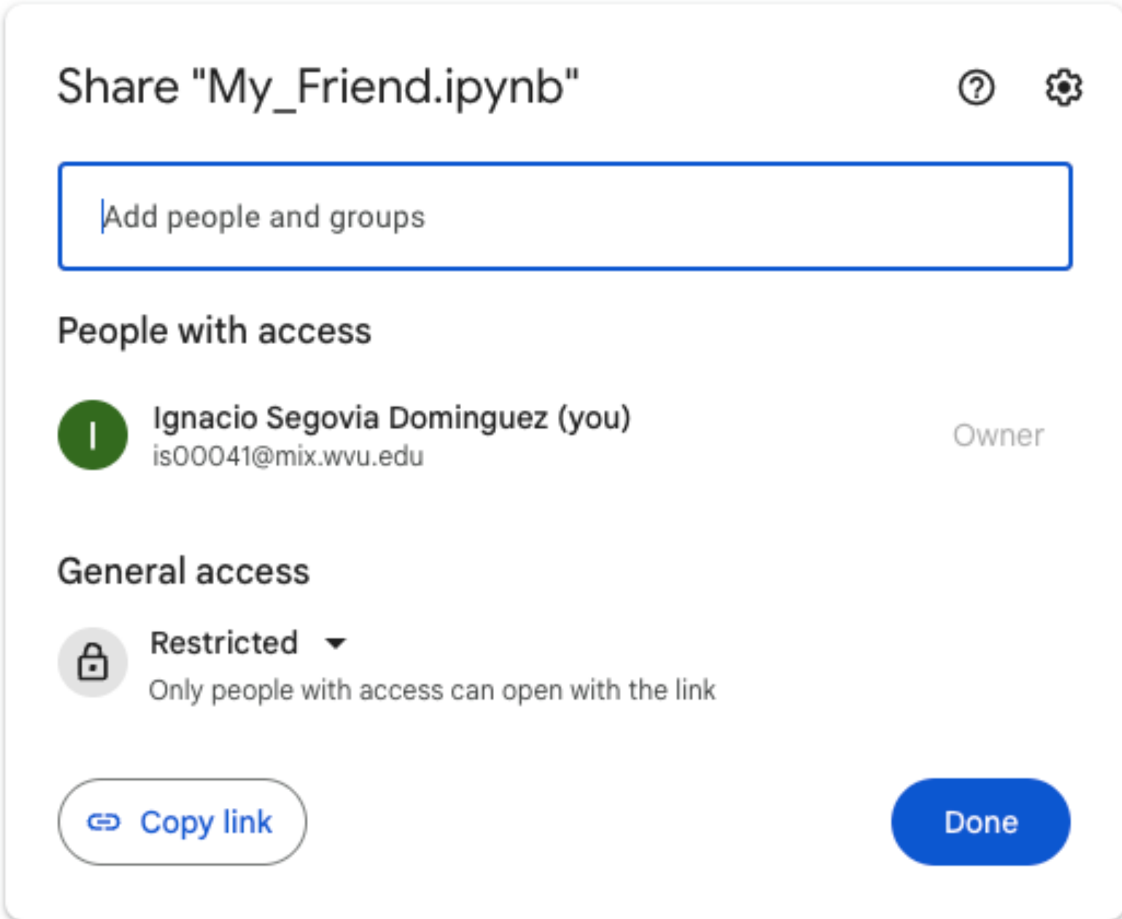
One of the most important features of Google Colab is the ability to share your work with others in real-time. You can collaborate with your team members, colleagues, or friends by sharing your Google Colab notebook with them.

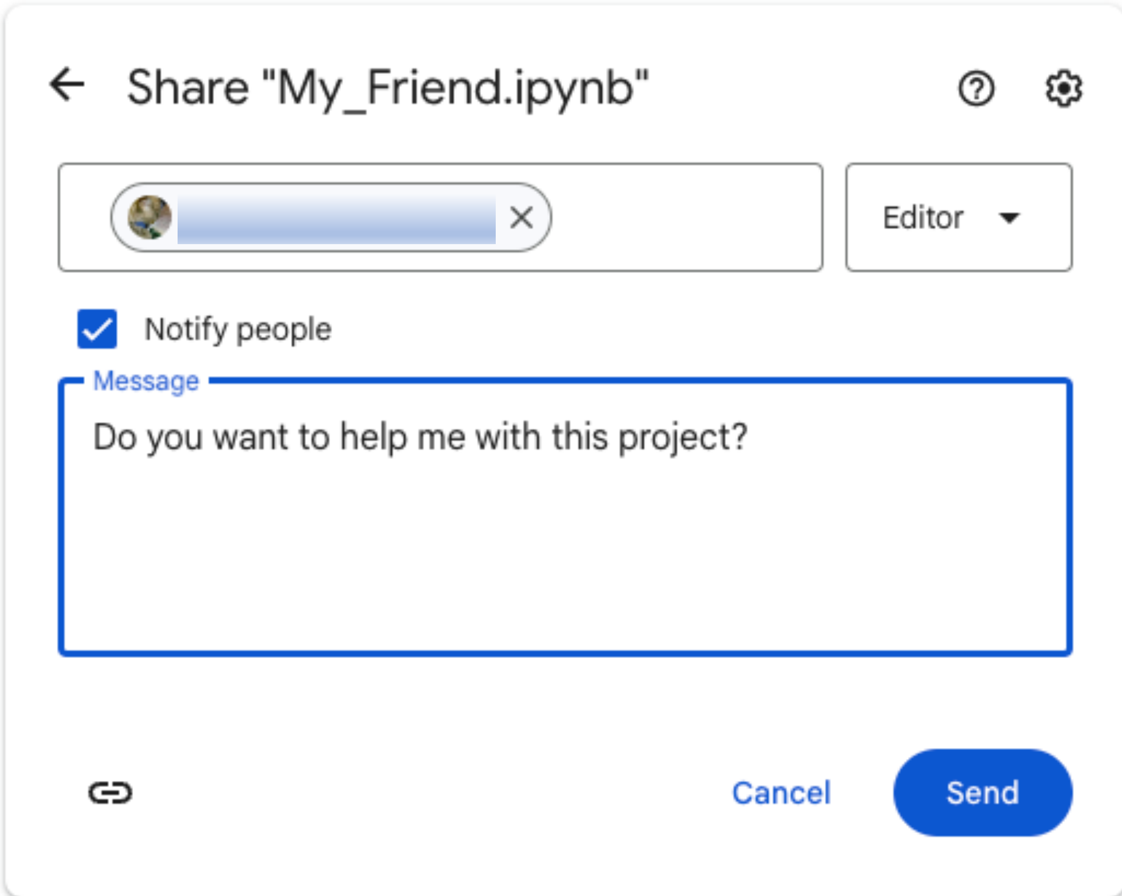
Sometimes you may need to share a Google Drive with all the members of Google Colab, especially if you are working on a group project where data needs to be shared among team members.

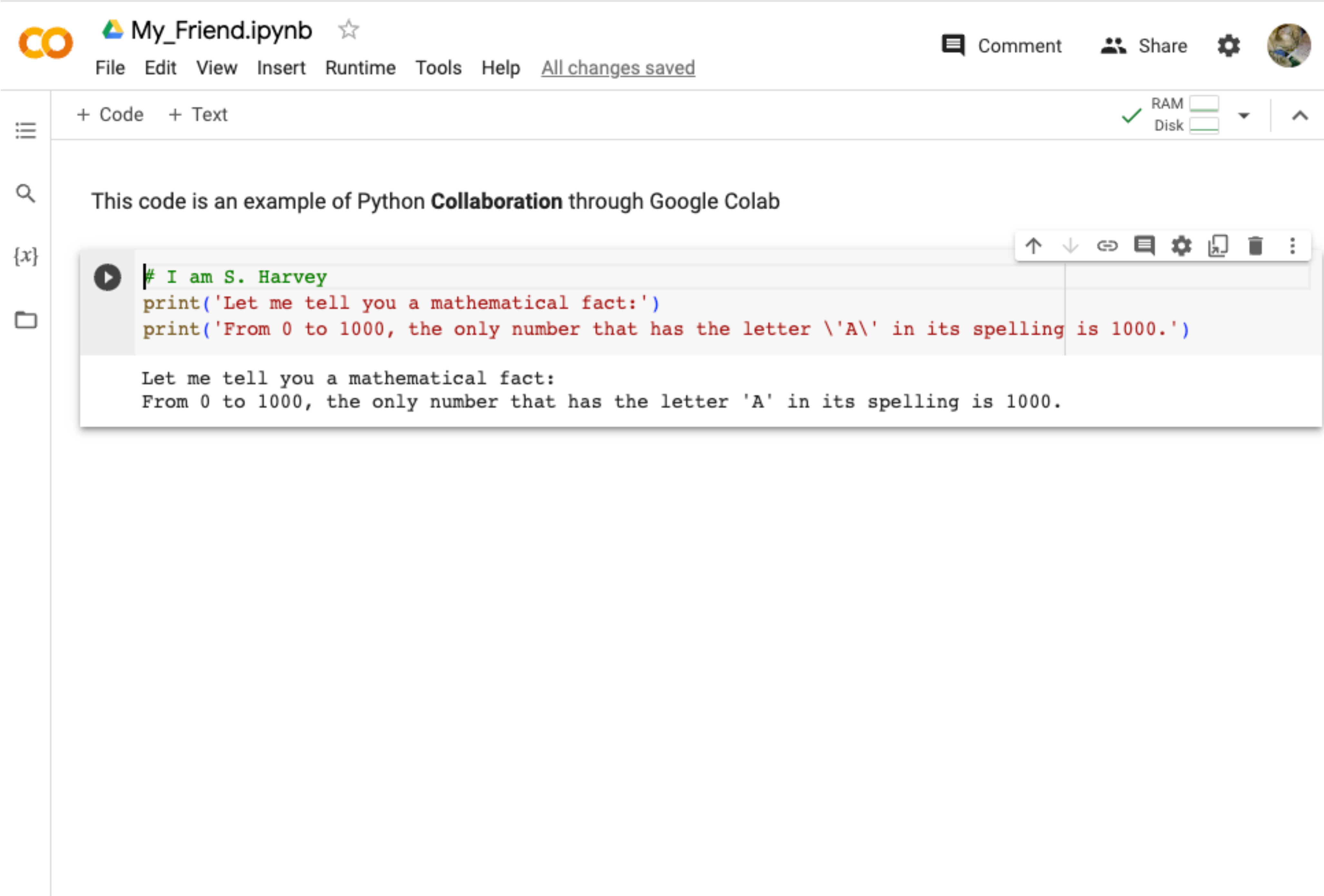
SaturnCloud

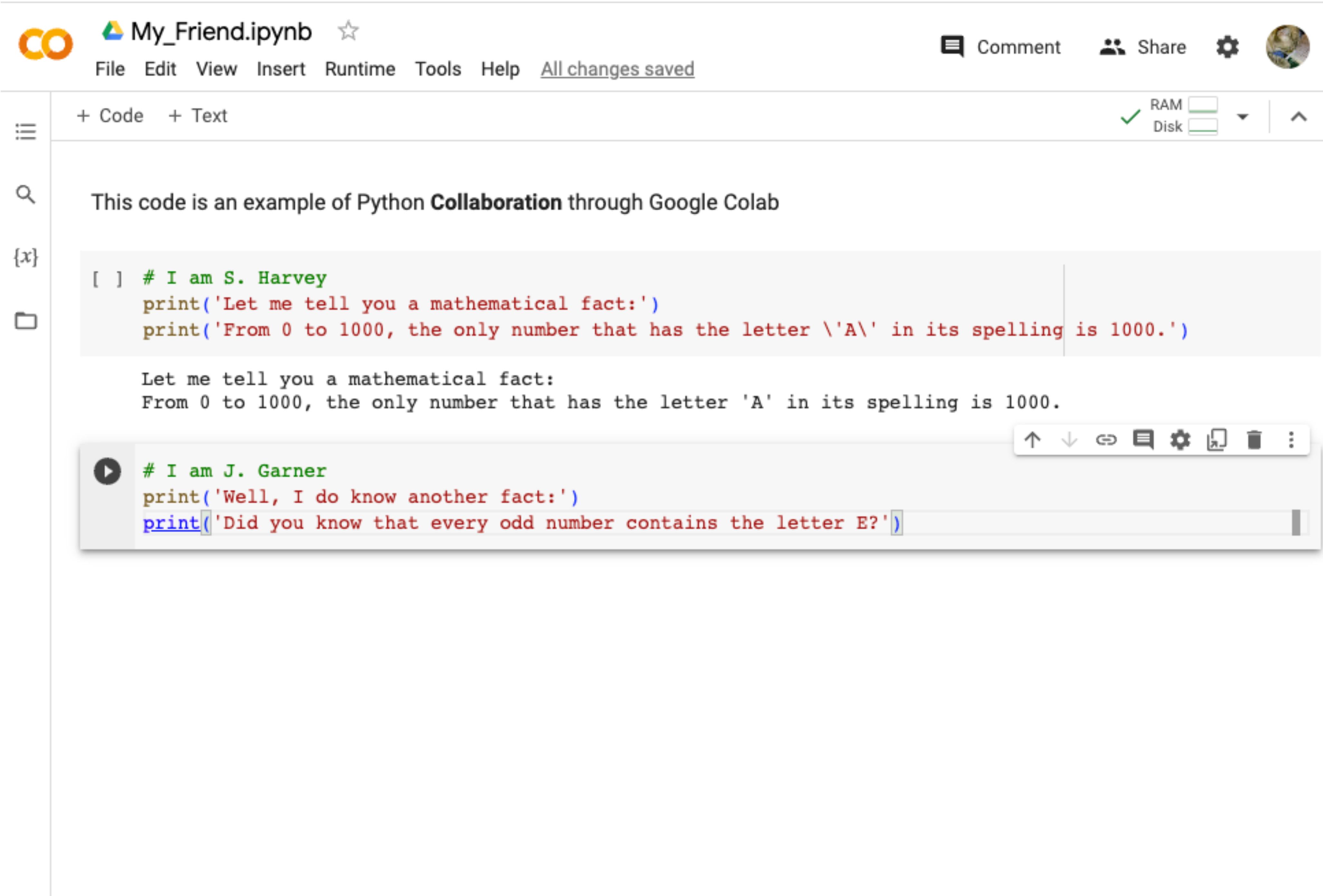
Other tools: Visual Studio Live Share, Code With Me (JetBrains), CodeTogether, GitLive, Github (Code versions), and much more…

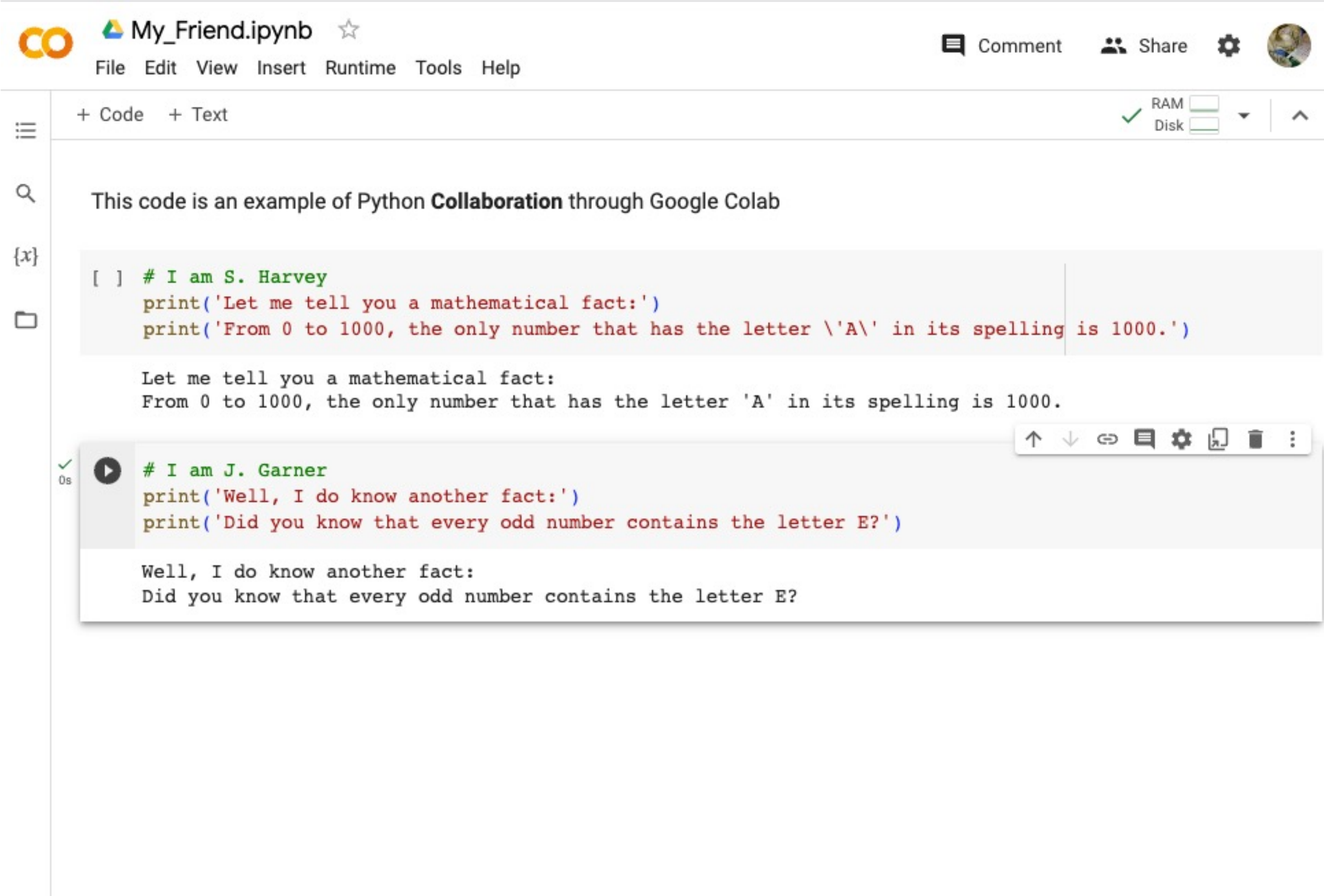

https://colab.research.google.com/drive/1jbWSGBgIV4ZvFTz_vZq-ZXW...
My_Friend.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment
Share
+ Code + Text
RAM
Disk
This code is an example of Python **Collaboration** through Google Colab
# I am S. Harvey
print('Let me tell you a mathematical fact:')
print('From 0 to 1000, the only number that has the letter \'A\' in its spelling is 1000.')
Let me tell you a mathematical fact:
From 0 to 1000, the only number that has the letter 'A' in its spelling is 1000.


Share "My_Friend.ipynb"
Add people and groups
People with access
Ignacio Segovia Dominguez (you)
is00041@mix.wvu.edu
Owner
General access
Restricted
Only people with access can open with the link
Copy link
Done


Share "My_Friend.ipynb"
Editor
Notify people
Message
Do you want to help me with this project?
Cancel
Send


My_Friend.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment
Share
+ Code + Text
RAM
Disk
This code is an example of Python **Collaboration** through Google Colab
# I am S. Harvey
print('Let me tell you a mathematical fact:')
print('From 0 to 1000, the only number that has the letter \'A\' in its spelling is 1000.')
Let me tell you a mathematical fact:
From 0 to 1000, the only number that has the letter 'A' in its spelling is 1000.


My_Friend.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment
Share
+ Code + Text
RAM
Disk
This code is an example of Python **Collaboration** through Google Colab
[ ]
# I am S. Harvey
print('Let me tell you a mathematical fact:')
print('From 0 to 1000, the only number that has the letter \'A\' in its spelling is 1000.')
Let me tell you a mathematical fact:
From 0 to 1000, the only number that has the letter 'A' in its spelling is 1000.
# I am J. Garner
print('Well, I do know another fact:')
print('Did you know that every odd number contains the letter E?')


My_Friend.ipynb
Comment
Share
File Edit View Insert Runtime Tools Help
+ Code + Text
RAM
Disk
This code is an example of Python Collaboration through Google Colab
[ ]
# I am S. Harvey
print('Let me tell you a mathematical fact:')
print('From 0 to 1000, the only number that has the letter \'A\' in its spelling is 1000.')
Let me tell you a mathematical fact:
From 0 to 1000, the only number that has the letter 'A' in its spelling is 1000.
0s
# I am J. Garner
print('Well, I do know another fact:')
print('Did you know that every odd number contains the letter E?')
Well, I do know another fact:
Did you know that every odd number contains the letter E?

Collaboration while writing Python code

One of the most important features of Google Colab is the ability to share your work with others in real-time. You can collaborate with your team members, colleagues, or friends by sharing your Google Colab notebook with them.

Sometimes you may need to share a Google Drive with all the members of Google Colab, especially if you are working on a group project where data needs to be shared among team members.

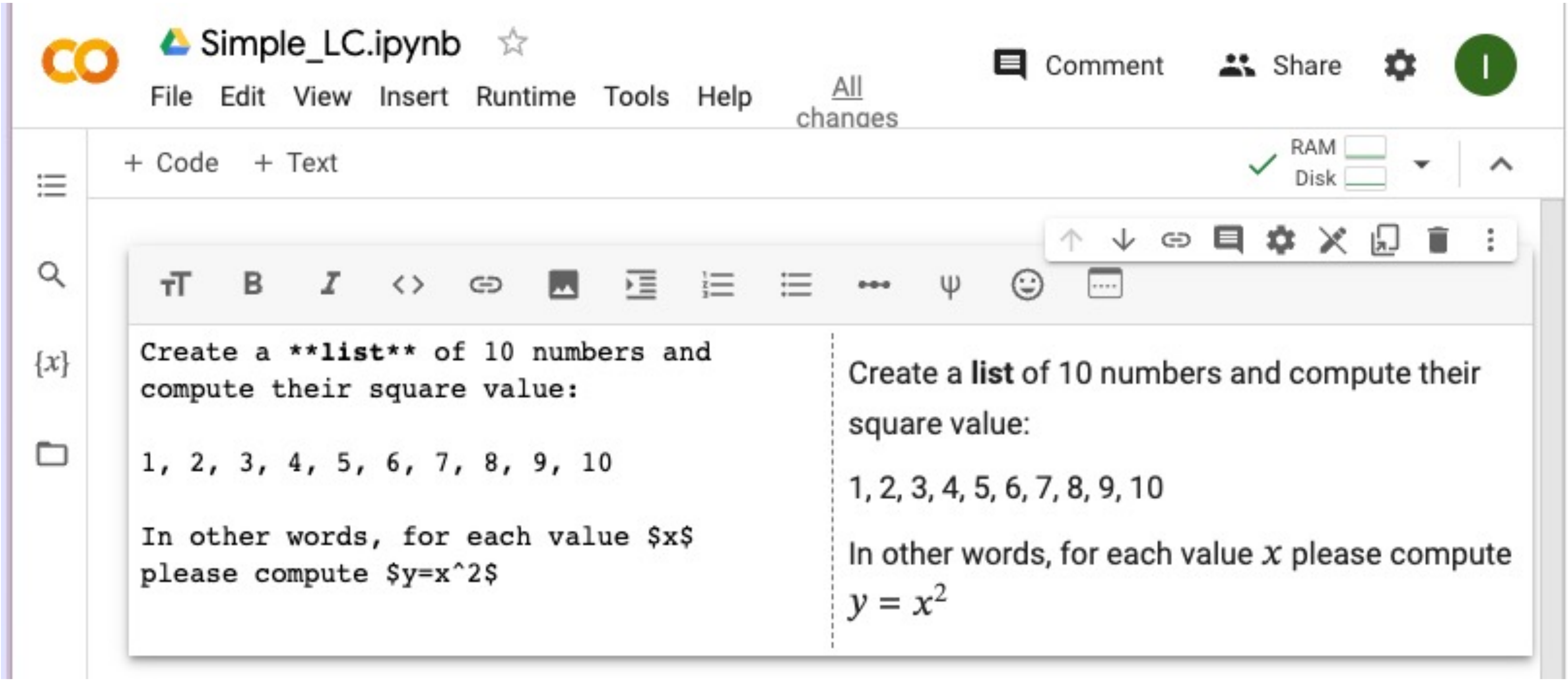
SaturnCloud

Other tools: Visual Studio Live Share, Code With Me (JetBrains), CodeTogether, GitLive, Github (Code versions), and much more…

Let’s move to another subject…

Any ideas to solve this problem?

Create a **list** of 10 numbers and compute their square value:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10



Note that Google Colab, GitHub and other platforms uses Markdown code for texts formatting.

https://colab.research.google.com/notebooks/markdown_guide.ipynb#scrollTo=Lhfnlq1Surtk

https://docs.github.com/en/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-syntax

Simple_LC.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

Create a **list** of 10 numbers and compute their square value:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

In other words, for each value x please compute $y = x^2$

[1]
```
# The artless approach

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squared = []

for x in numbers:
  squared.append(x**2)

print(squared)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

List Comprehension

List comprehensions offer a succinct way to *create lists* based on existing lists. When using list comprehensions, lists can be built by leveraging *any iterable*, including strings and tuples.

Syntactically, list comprehensions consist of an iterable containing an **expression** followed by a **for** clause. This can be followed by *additional* **for** or **if** clauses, so familiarity with for loops and conditional statements will help you understand list comprehensions better.

List comprehensions provide an alternative syntax to creating lists and other sequential data types. While other methods of iteration, such as for loops, can also be used to create lists, list comprehensions may be preferred because they can limit the number of lines used in your program.

DigitalOcean

List Comprehension

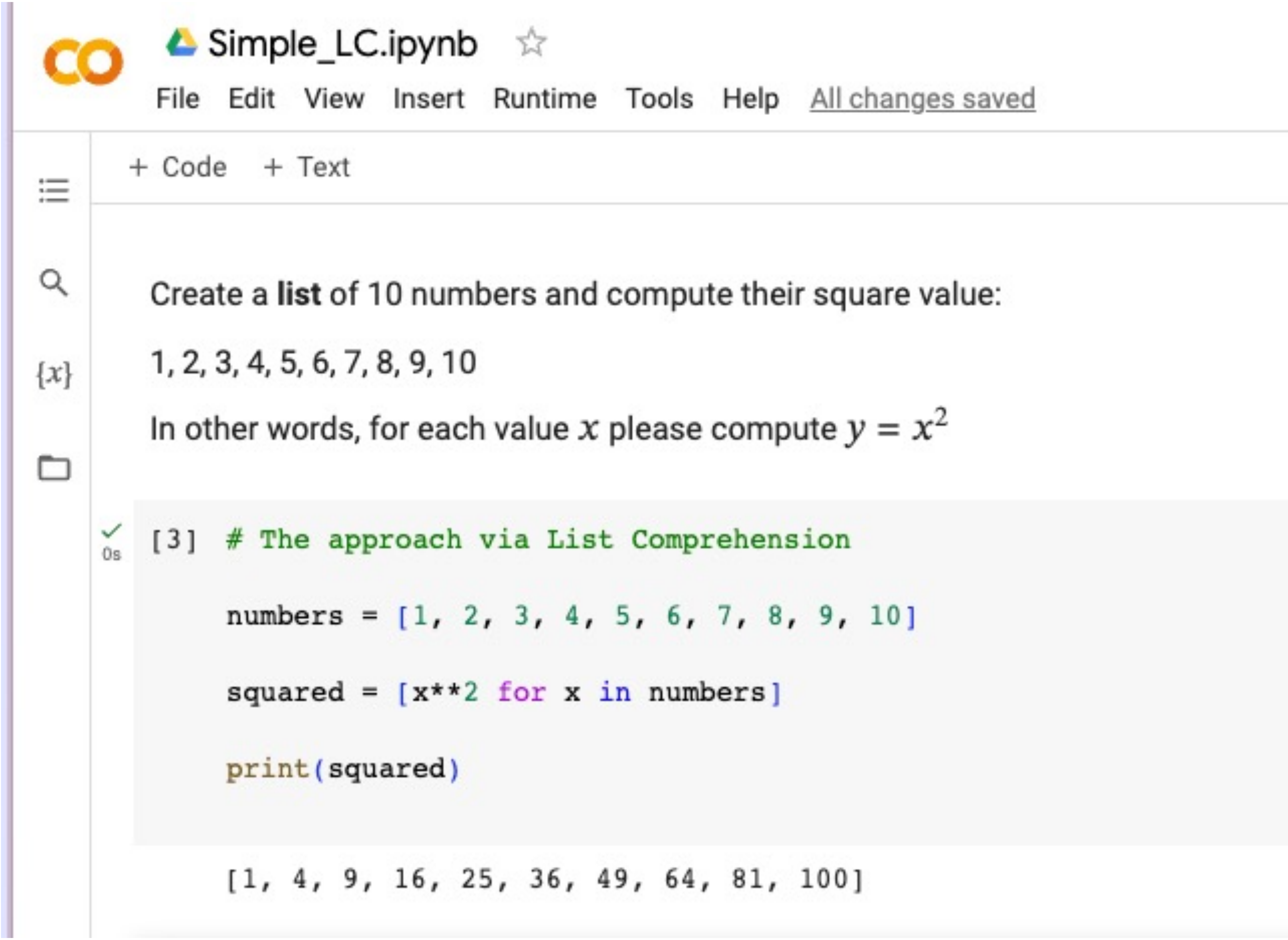
Python List comprehension provides a much more short syntax for creating a new list based on the values of an existing list.

Syntax: *newList =* ***[*** *expression(element)* ***for*** *element* ***in*** *oldList* ***if*** *condition* ***]***

Parameter:

- ***`expression`***: *Represents the operation you want to execute on every item within the iterable.*
- ***`element`***: *The term "variable" refers to each value taken from the iterable.*
- ***`iterable`***: *specify the sequence of elements you want to iterate through.(e.g., a list, tuple, or string).*
- ***`condition`***: *(Optional) A filter helps decide whether or not an element should be added to the new list.*

Return:*The return value of a list comprehension is a new list containing the modified elements that satisfy the given criteria.*


Simple_LC.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Create a list of 10 numbers and compute their square value:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
In other words, for each value x please compute y = x2
[3]
# The approach via List Comprehension
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squared = [x**2 for x in numbers]
print(squared)
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
0s

List Comprehension

Let’s explore more sophisticated constructions…

Simple_LC.ipynb

File Edit View Insert Runtime Tools Help All changes

Comment Share

+ Code + Text

RAM
Disk

To add conditional statements

Create a list with all the **even** numbers between -1 and 21.

AS a gently reminder, an even number is an integer of the form $n = 2k$, where k is an integer. Since the even numbers are integrally divisible by two, the congruence $n \equiv (mod 2)$ holds for even n.

Simple_LC.ipynb

File Edit View Insert Runtime Tools Help All changes

Comment Share

+ Code + Text

RAM
Disk

To add conditional statements

Create a list with all the **even** numbers between -1 and 21.

AS a gently reminder, an even number is an integer of the form $n = 2k$, where k is an integer. Since the even numbers are integrally divisible by two, the congruence $n \equiv (mod 2)$ holds for even n.

Simple_LC.ipynb

File Edit View Insert Runtime Tools Help All changes

Comment Share

+ Code + Text

RAM Disk

To add conditional statements

Create a list with all the **even** numbers between -1 and 21.

AS a gently reminder, an even number is an integer of the form $n = 2k$, where k is an integer. Since the even numbers are integrally divisible by two, the congruence $n \equiv (mod 2)$ holds for even n.

```
[5] even_number = [x for x in range(0, 21) if x % 2 == 0]
    print(even_number)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Simple_LC.ipynb

File Edit View Insert Runtime Tools Help All changes

Comment Share

+ Code + Text

RAM Disk

To add conditional statements

Create a list with all the **even** numbers between -1 and 21.

AS a gently reminder, an even number is an integer of the form $n = 2k$, where k is an integer. Since the even numbers are integrally divisible by two, the congruence $n \equiv (mod 2)$ holds for even n.

```
[5] even_number = [x for x in range(0, 21) if x % 2 == 0]
    print(even_number)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

```
[8] list(range(0, 21))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Simple_LC.ipynb

File Edit View Insert Runtime Tools Help All changes

Comment Share

+ Code + Text

RAM
Disk

Nested IF Statements

Create a list of numbers that are both, divisible by 3 and divisible by 5.

Save in the list all the numbers between 0 and 100 that meet both conditions.

Simple_LC.ipynb

File Edit View Insert Runtime Tools Help All changes

Comment Share

+ Code + Text RAM Disk

Nested IF Statements

Create a list of numbers that are both, divisible by 3 and divisible by 5.

Save in the list all the numbers between 0 and 100 that meet both conditions.

[9]
```
number_conditions = [x for x in range(-1, 100) if x % 3 == 0 if x % 5 == 0]
print(number_conditions)
```

```
[0, 15, 30, 45, 60, 75, 90]
```

Simple_LC.ipynb

Nested IF Statements

Create a list of numbers that are both, divisible by 3 and divisible by 5.

Save in the list all the numbers between 0 and 100 that meet both conditions.

```
[9] number_conditions = [x for x in range(-1, 100) if x % 3 == 0 if x % 5 == 0]
    print(number_conditions)
```

```
[0, 15, 30, 45, 60, 75, 90]
```

```
[12] 47%3
```

```
2
```

Simple_LC.ipynb

File Edit View Insert Runtime Tools Help All changes

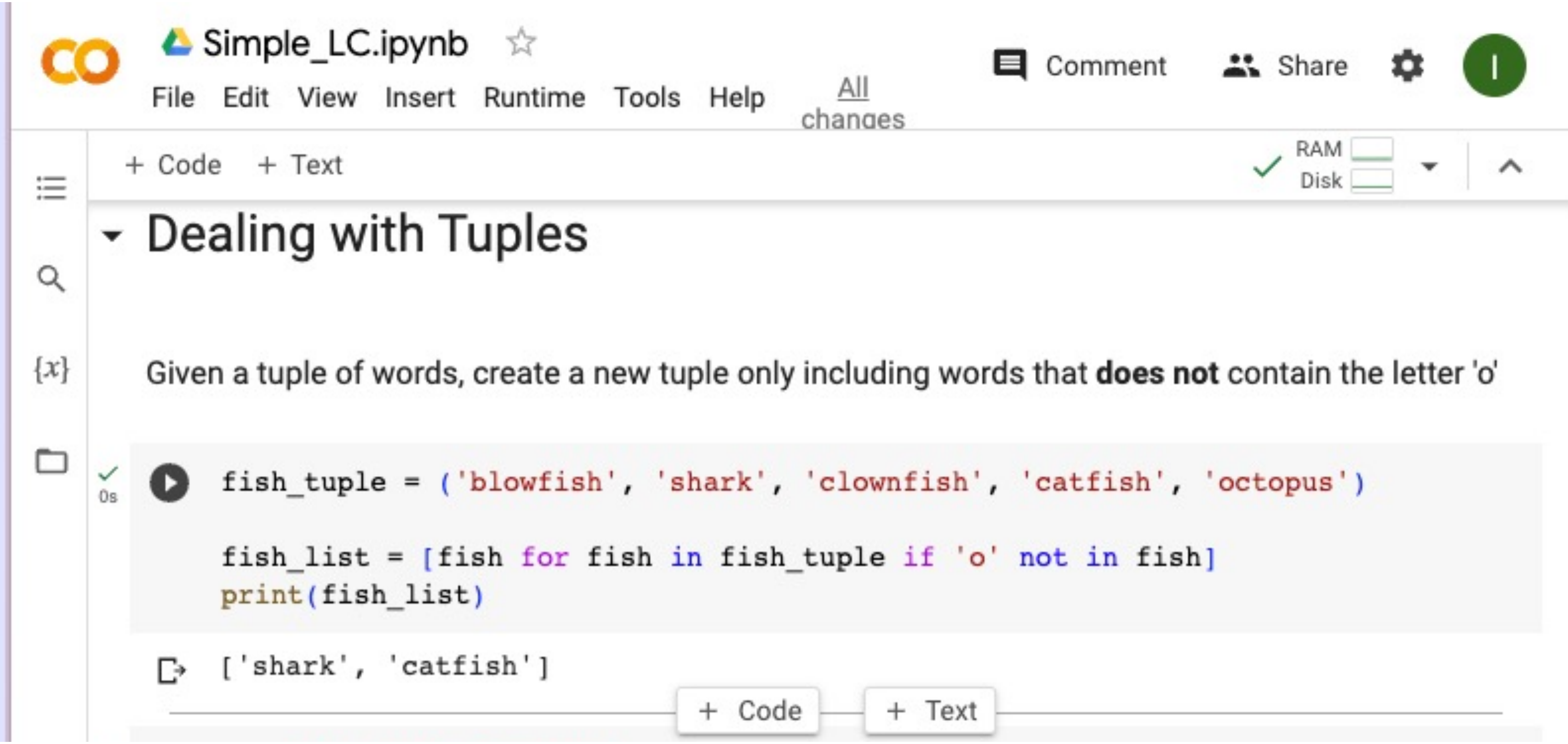
Comment Share

+ Code + Text

RAM Disk

Dealing with Tuples

Given a tuple of words, create a new tuple only including words that **does not** contain the letter 'o'


Simple_LC.ipynb
File Edit View Insert Runtime Tools Help All changes
Comment
Share
+ Code + Text
RAM
Disk
Dealing with Tuples
Given a tuple of words, create a new tuple only including words that **does not** contain the letter 'o'
fish_tuple = ('blowfish', 'shark', 'clownfish', 'catfish', 'octopus')
fish_list = [fish for fish in fish_tuple if 'o' not in fish]
print(fish_list)
['shark', 'catfish']
+ Code + Text

Simple_LC.ipynb

File Edit View Insert Runtime Tools Help All changes

Comment Share

+ Code + Text

RAM Disk

Dealing with Tuples

Given a tuple of words, create a new tuple only including words that **does not** contain the letter 'o'

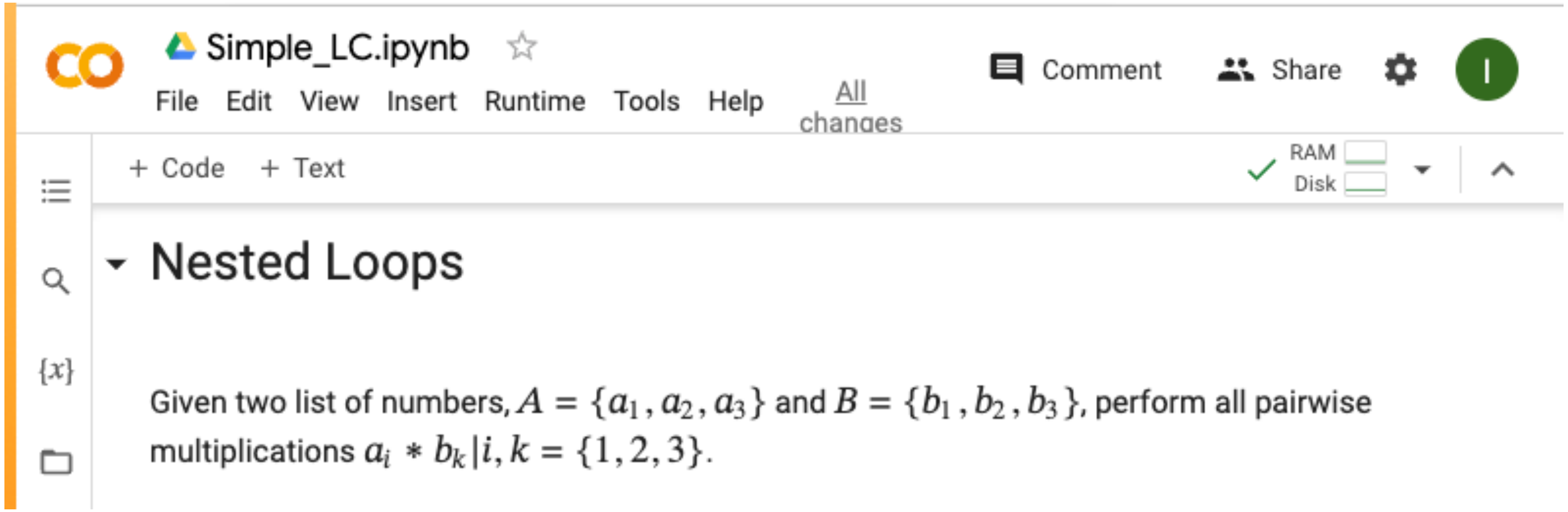
```
fish_tuple = ('blowfish', 'shark', 'clownfish', 'catfish', 'octopus')

fish_list = [fish for fish in fish_tuple if 'o' not in fish]
print(fish_list)
```

```
['shark', 'catfish']
```

+ Code + Text

[29]
```
a = 'The world is yours'
print('you' in a)
print('you' not in a)
```

```
True
False
```


Simple_LC.ipynb
File Edit View Insert Runtime Tools Help All changes
Comment
Share
+ Code + Text
RAM
Disk
Nested Loops
Given two list of numbers, $A = \{a_1, a_2, a_3\}$ and $B = \{b_1, b_2, b_3\}$, perform all pairwise multiplications $a_i * b_k | i, k = \{1, 2, 3\}$.

Simple_LC.ipynb

File Edit View Insert Runtime Tools Help All changes

Comment Share

+ Code + Text

RAM Disk

Nested Loops

Given two list of numbers, $A = \{a_1, a_2, a_3\}$ and $B = \{b_1, b_2, b_3\}$, perform all pairwise multiplications $a_i * b_k | i, k = \{1, 2, 3\}$.

0s

```python
# The artless approach

my_list = []

for x in [20, 40, 60]:
  for y in [2, 4, 6]:
    my_list.append(x * y)

print(my_list)
```

```
[40, 80, 120, 80, 160, 240, 120, 240, 360]
```

Simple_LC.ipynb

File Edit View Insert Runtime Tools Help All changes

Comment Share

+ Code + Text

RAM Disk

Nested Loops

Given two list of numbers, $A = \{a_1, a_2, a_3\}$ and $B = \{b_1, b_2, b_3\}$, perform all pairwise multiplications $a_i * b_k | i, k = \{1, 2, 3\}$.

```
# The artless approach

my_list = []

for x in [20, 40, 60]:
  for y in [2, 4, 6]:
    my_list.append(x * y)

print(my_list)
```

```
[40, 80, 120, 80, 160, 240, 120, 240, 360]
```

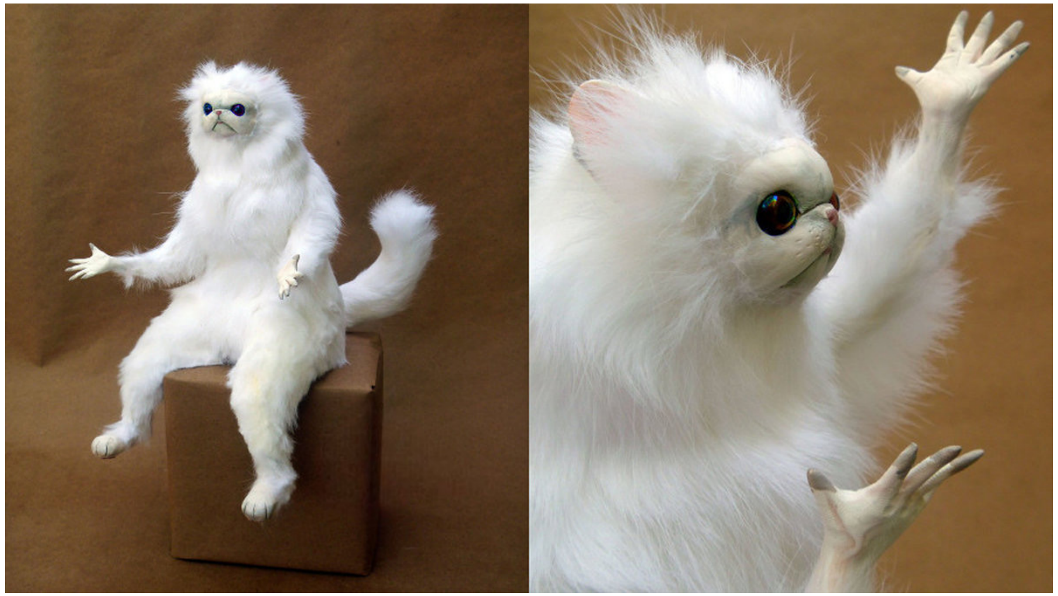
[31]

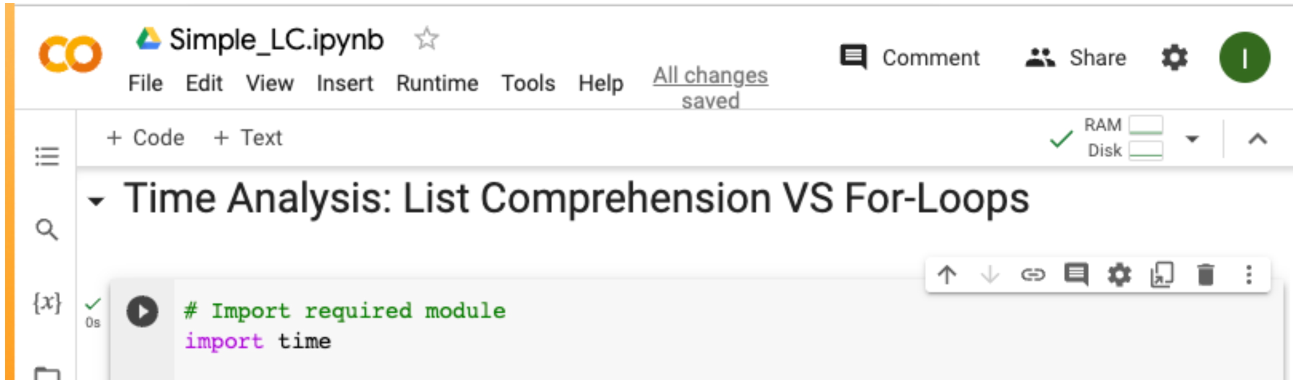
```
# Using List Comprehension

my_list = [x * y for x in [20, 40, 60] for y in [2, 4, 6]]
print(my_list)
```

```
[40, 80, 120, 80, 160, 240, 120, 240, 360]
```

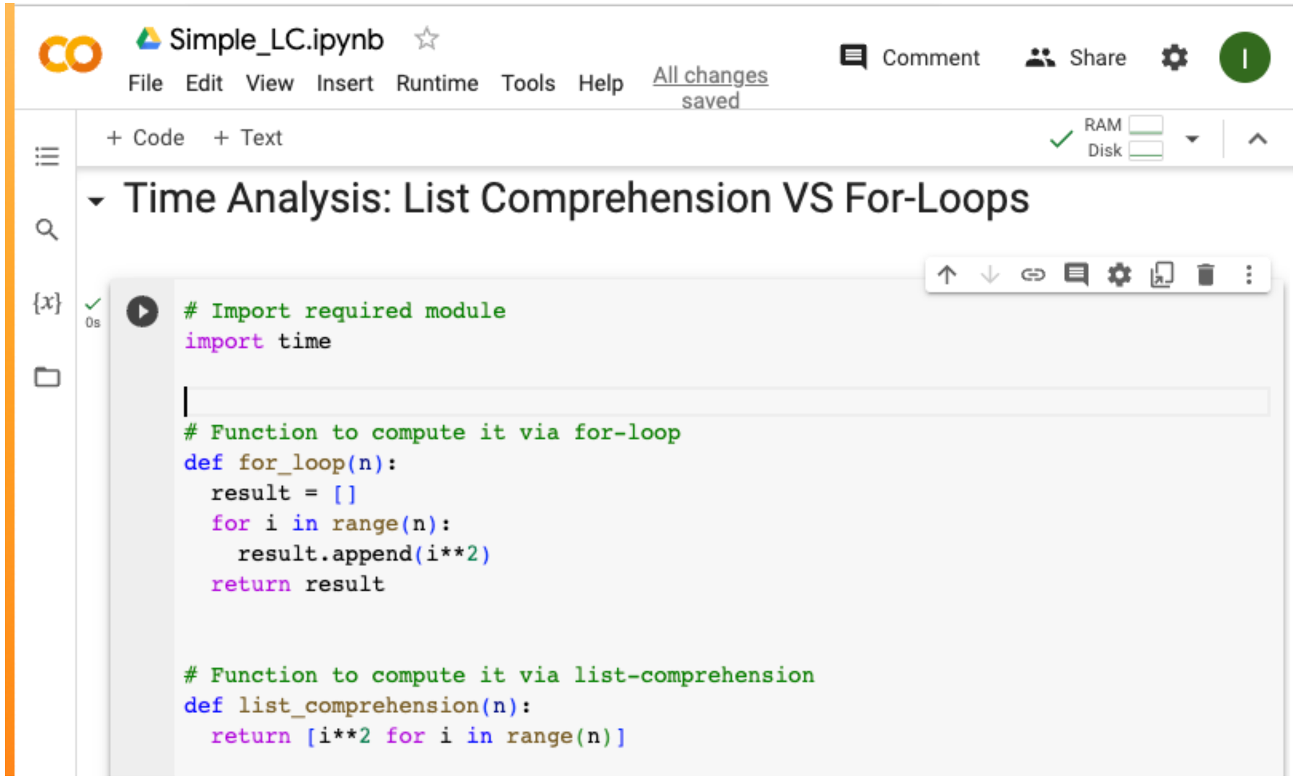
List Comprehension

Why do we care about writing code via List Comprehension if at the end we got the same answer? right?





For the Unix system, January 1, 1970, 00:00:00 at **UTC** is epoch.



Simple_LC.ipynb
File Edit View Insert Runtime Tools Help
All changes saved
Comment
Share
+ Code
+ Text
RAM
Disk
Time Analysis: List Comprehension VS For-Loops
# Import required module
import time
# Function to compute it via for-loop
def for_loop(n):
  result = []
  for i in range(n):
    result.append(i**2)
  return result
# Function to compute it via list-comprehension
def list_comprehension(n):
  return [i**2 for i in range(n)]

Simple_LC.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

+ Code + Text

RAM
Disk

Time Analysis: List Comprehension VS For-Loops

```
# Import required module
import time

# Function to compute it via for-loop
def for_loop(n):
  result = []
  for i in range(n):
    result.append(i**2)
  return result


# Function to compute it via list-comprehension
def list_comprehension(n):
  return [i**2 for i in range(n)]


# Calculate the time taken by for_loop()
begin = time.time()
for_loop(10**6)
end = time.time()

# Display time taken by for_loop()
print('Time taken -> for_loop:', round(end-begin, 2))

# Calculate the time taken by list_comprehension()
begin = time.time()
list_comprehension(10**6)
end = time.time()

# Display time taken by for_loop()
print('Time taken -> list_comprehension:', round(end-begin, 2))
```

Simple_LC.ipynb

File Edit View Insert Runtime Tools Help All changes saved

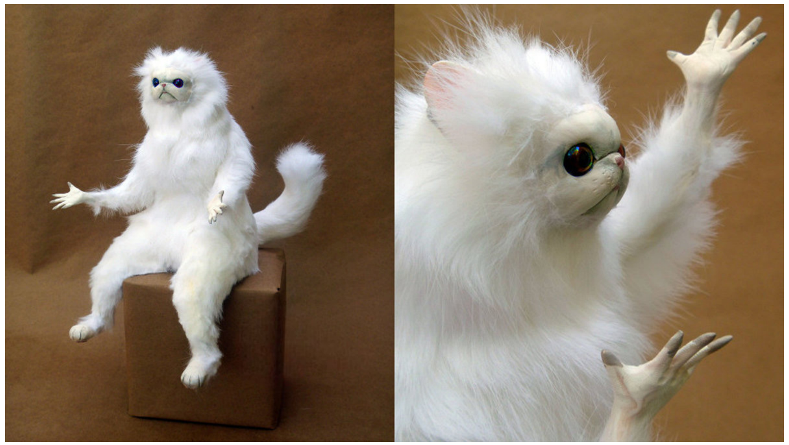
Time Analysis: List Comprehension VS For-Loops

```python
# Import required module
import time

# Function to compute it via for-loop
def for_loop(n):
  result = []
  for i in range(n):
    result.append(i**2)
  return result


# Function to compute it via list-comprehension
def list_comprehension(n):
  return [i**2 for i in range(n)]


# Calculate the time taken by for_loop()
begin = time.time()
for_loop(10**6)
end = time.time()

# Display time taken by for_loop()
print('Time taken -> for_loop:', round(end-begin, 2))

# Calculate the time taken by list_comprehension()
begin = time.time()
list_comprehension(10**6)
end = time.time()

# Display time taken by for_loop()
print('Time taken -> list_comprehension:', round(end-begin, 2))
```

```
Time taken -> for_loop: 0.41
Time taken -> list_comprehension: 0.35
```

List Comprehension

Why do we care about writing code via List Comprehension if at the end we got the same answer? right?



In general, list comprehensions are faster than for-loops.

Let’s move to another subject…

How data-storage is managed while working on Python?

As Data Scientist, we care about writing efficient code that runs fast.

But also…

writing efficient code means writing a ***memory-efficient*** code

Memory Management in Python

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the Python memory manager. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

At the lowest level, a raw memory allocator ensures that there is enough room in the private heap for storing all Python-related data by interacting with the memory manager of the operating system. On top of the raw memory allocator, several object-specific allocators operate on the same heap and implement distinct memory management policies adapted to the peculiarities of every object type. For example, integer objects are managed differently within the heap than strings, tuples or dictionaries because integers imply different storage requirements and speed/space tradeoffs. The Python memory manager thus delegates some of the work to the object-specific allocators, but ensures that the latter operate within the bounds of the private heap.

Python.org

Memory Management in Python

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the Python memory manager. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

At the lowest level, a raw memory allocator ensures that there is enough room in the private heap for storing all Python-related data by interacting with the memory manager of the operating system. On top of the raw memory allocator, several object-specific allocators operate on the same heap and implement distinct memory management policies adapted to the peculiarities of every object type. For example, integer objects are managed differently within the heap than strings, tuples or dictionaries because integers imply different storage requirements and speed/space tradeoffs. The Python memory manager thus delegates some of the work to the object-specific allocators, but ensures that the latter operate within the bounds of the private heap.

Python.org

This topic is very technical…

Memory Management in Python

Key points:

- Garbage collection is a process in which the interpreter frees up the memory when not in use to make it available for other objects.
- Reference counting works by counting the number of times an object is referenced by other objects in the system. When references to an object are removed, the reference count for an object is decremented. When the reference count becomes zero, the object is deallocated.
- CPython is the default and most widely used implementation of the Python language. Since CPython is the reference implementation, all new rules and specifications of the Python language are first implemented by CPython.

Memory Management in Python

Key points:

- It's important to note that there are implementations other than CPython. IronPython compiles down to run on Microsoft's Common Language Runtime. Jython compiles down to Java bytecode to run on the Java Virtual Machine. PyPy claims to run faster for particular applications.
- Each object has its own object-specific memory allocator that knows how to get the memory to store that object. Each object also has an object-specific memory deallocator that "frees" the memory once it's no longer needed.
- The Global Interpreter Lock (GIL) performs a single global lock on the interpreter when a thread is interacting with shared resources. In other words, only one thread can write at a time.

If you want to get deep into memory management in Python:

https://docs.python.org/3/c-api/memory.html

https://www.honeybadger.io/blog/memory-management-in-python/

Memory Management in Python

How Python make efficient use of memory allocation?

Let’s remember the Phyton function **id()**

Python id() function returns the “identity” of the object. The identity of an object is an integer, which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same id() value.

In CPython implementation, this is the address of the object in memory.

DigitalOcean

Mem_Allocation.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

+ Code + Text

RAM Disk

Python ID

[2]

```
# The identity/address of 4 integers/objects

a = 10
b = 10
c = 11
d = 12

print(id(a))
print(id(b))
print(id(c))
print(id(d))
```

Mem_Allocation.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

+ Code + Text

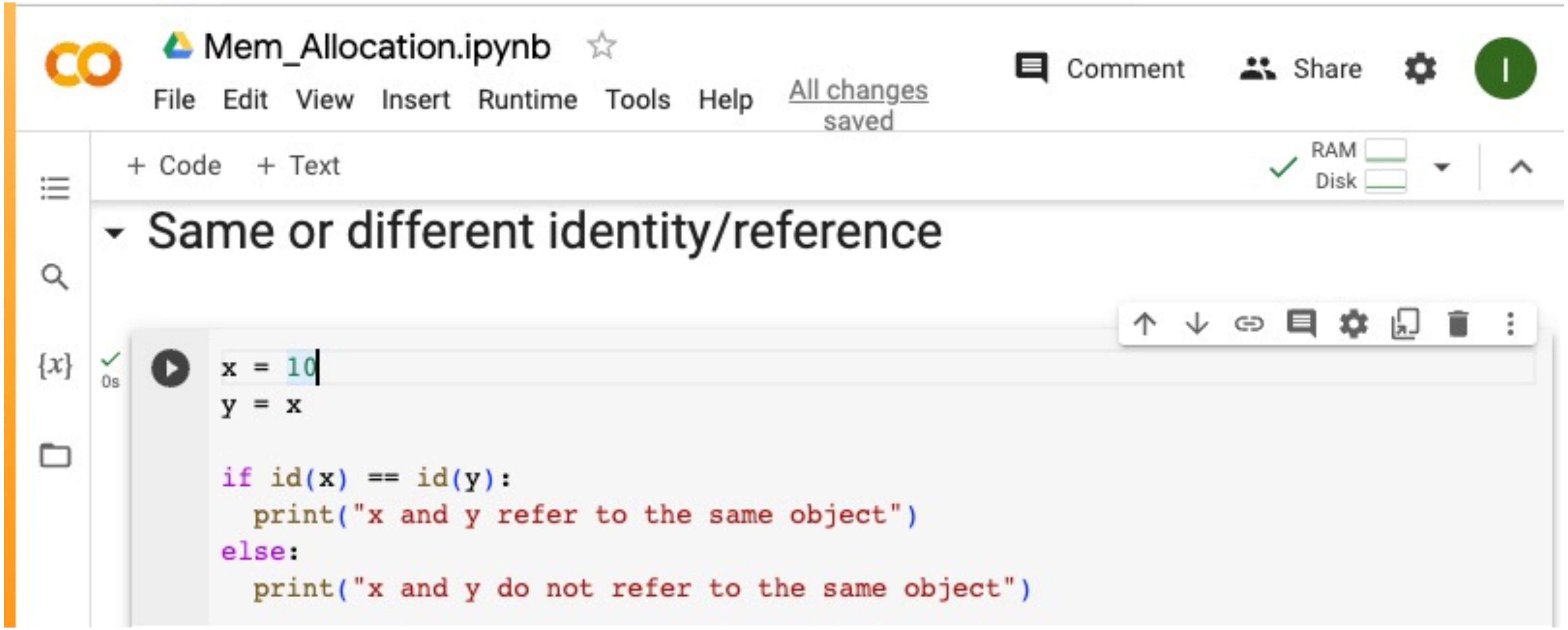
RAM Disk

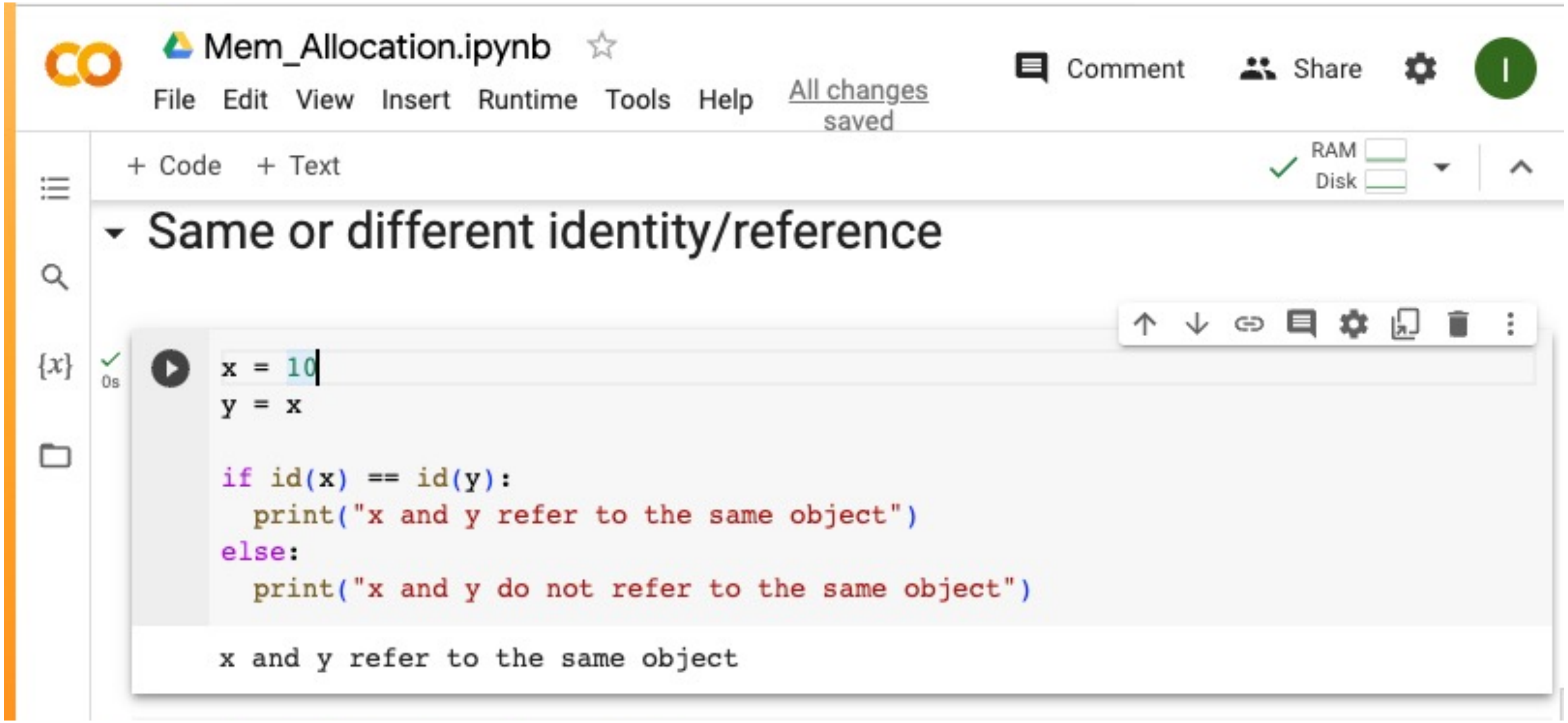
Python ID

[2]

```
# The identity/address of 4 integers/objects

a = 10
b = 10
c = 11
d = 12

print(id(a))
print(id(b))
print(id(c))
print(id(d))
```

```
133584008266256
133584008266256
133584008266288
133584008266320
```


Mem_Allocation.ipynb
File Edit View Insert Runtime Tools Help
All changes saved
Comment
Share
+ Code + Text
RAM
Disk
Same or different identity/reference
0s
x = 10
y = x
if id(x) == id(y):
print("x and y refer to the same object")
else:
print("x and y do not refer to the same object")


Mem_Allocation.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment
Share
+ Code + Text
RAM
Disk
Same or different identity/reference
x = 10
y = x

if id(x) == id(y):
  print("x and y refer to the same object")
else:
  print("x and y do not refer to the same object")
x and y refer to the same object
0s

Mem_Allocation.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

+ Code + Text

RAM Disk

Same or different identity/reference

```
x = 10
y = x

if id(x) == id(y):
  print("x and y refer to the same object")
else:
  print("x and y do not refer to the same object")
```

```
x and y refer to the same object
```

```
# Modifying the value of 'x'
x = 10
y = x
x += 1

if id(x) == id(y):
  print("x and y refer to the same object")
else:
  print("x and y do not refer to the same object")
```

Mem_Allocation.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Same or different identity/reference

```python
x = 10
y = x

if id(x) == id(y):
  print("x and y refer to the same object")
else:
  print("x and y do not refer to the same object")
```

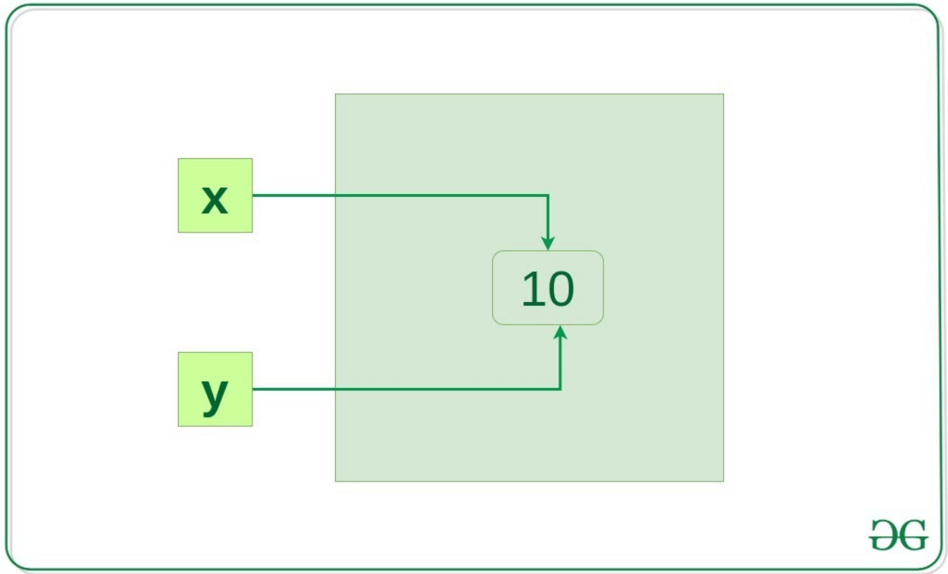
```
x and y refer to the same object
```

```python
# Modifying the value of 'x'
x = 10
y = x
x += 1

if id(x) == id(y):
  print("x and y refer to the same object")
else:
  print("x and y do not refer to the same object")
```

```
x and y do not refer to the same object
```

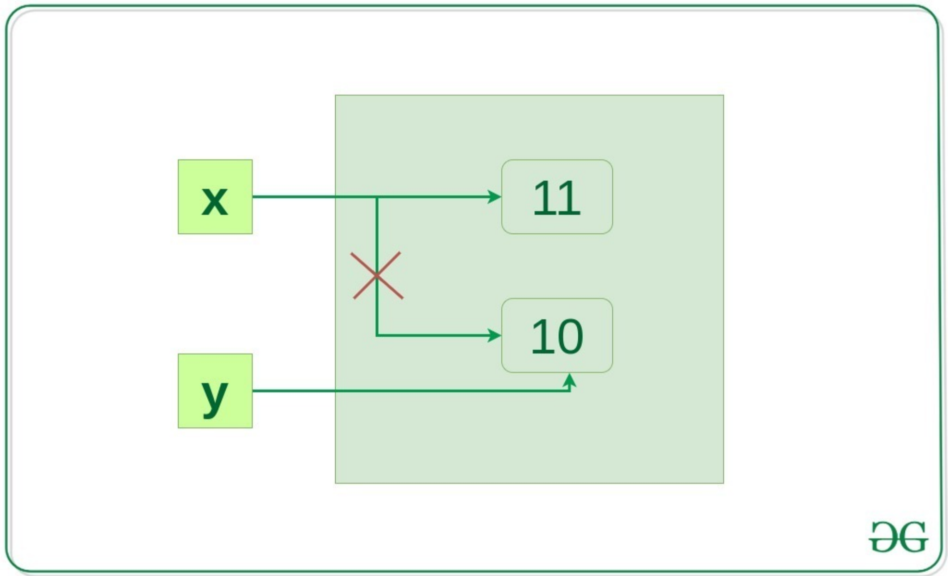
```python
x = 10
y = x

if id(x) == id(y):
  print("x and y refer to the same object")
else:
  print("x and y do not refer to the same object")
```

```
x and y refer to the same object
```




```python
# Modifying the value of 'x'
x = 10
y = x
x += 1

if id(x) == id(y):
  print("x and y refer to the same object")
else:
  print("x and y do not refer to the same object")
```

```
x and y do not refer to the same object
```



Mem_Allocation.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

+ Code + Text RAM Disk

Behaviour with String and Tuples

```
# Do we get the same behaviour when using string and tuples?

# Strings
print('Strings: ')
s1 = 'ABC'
s2 = 'ABC'
print(id(s1))
print(id(s2))

# Tuples
print('Tuples: ')
tu1 = ['A', 'B']
tu2 = ['A', 'B']
tu3 = tu1
print(id(tu1))
print(id(tu2))
print(id(tu3))
tu4 = tu3
tu4.append('C')
print(id(tu4))
tu4[0] = 'D'
print(id(tu4))
```

Mem_Allocation.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

+ Code + Text

RAM Disk

Behaviour with String and Tuples

```python
# Do we get the same behaviour when using string and tuples?

# Strings
print('Strings: ')
s1 = 'ABC'
s2 = 'ABC'
print(id(s1))
print(id(s2))

# Tuples
print('Tuples: ')
tu1 = ['A', 'B']
tu2 = ['A', 'B']
tu3 = tu1
print(id(tu1))
print(id(tu2))
print(id(tu3))
tu4 = tu3
tu4.append('C')
print(id(tu4))
tu4[0] = 'D'
print(id(tu4))
```

```
Strings:
133584007093552
133584007093552
Tuples:
133583817816832
133583228411712
133583817816832
133583817816832
133583817816832
```

Summary

- Code collaboration in real-time among data scientists is crucial to speed-up results and to be part of an efficient teamwork.
- Techniques, such as list comprehension, help us to write faster code.
- Understanding memory allocation and management help us to write efficient code.

Stay Safe & Healthy