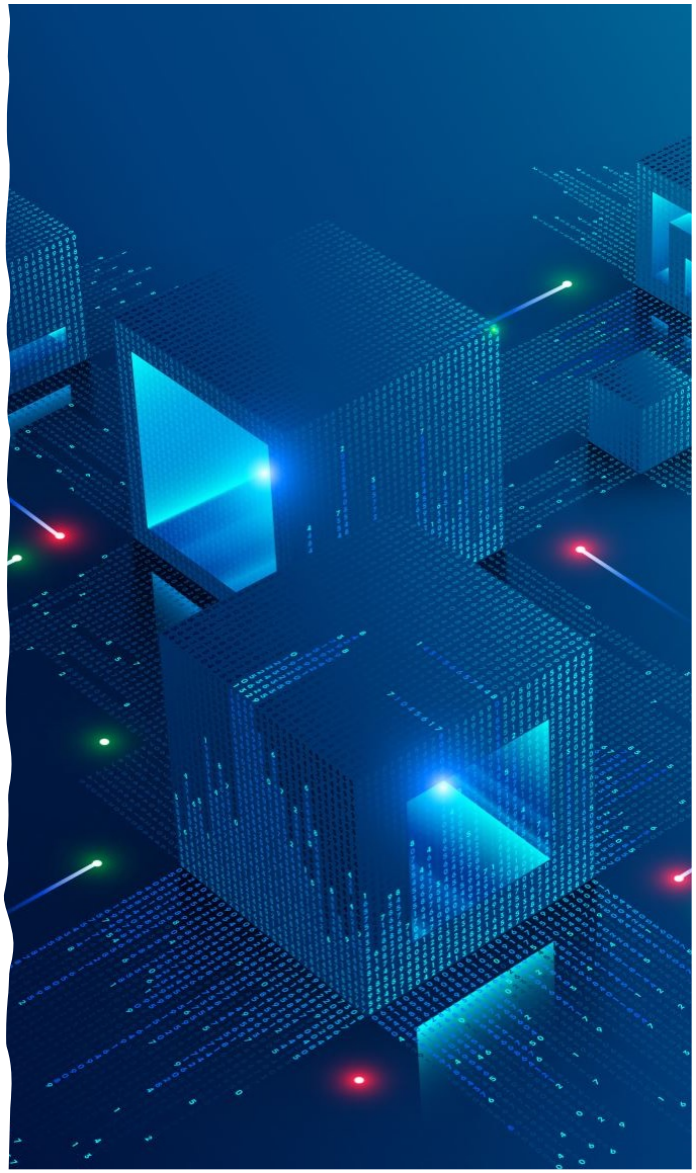
List Comprehension, Memory Management, and Lambda Expression



Solve this…

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

```
[ ]  # Two artless approaches

     # Option 1

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

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

     print(squared)

     # Option 2

     squared_2 = []

     for x in numbers:
       squared_2.append(x * x)

     print(squared_2)
```

```
     # With List Comprehension

     numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
     squares = [x ** 2 for x in numbers]

     print(squares)

     # alternatively

     squares = [x**2 for x in range(1, 11)]
     print(squares)
```

List Comprehensions

- Offer a **concise way** to create new lists from existing iterables
- Work with **any iterable**, including lists, strings, and tuples
- Syntax includes:
  - A **New List**
  - An **expression**
  - A **for clause**
  - Optionally, additional **for or if clauses**

Why Use Them?

- Provide a **compact alternative** to standard for loops
- Help you **write cleaner, shorter code**
- Ideal for simple transformations or filtering
- Often faster than loops

List Comprehension

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

newList = [*expression* **for** *clause*]

newList_with_conditional = [*expression* **for** *clause* **if** *conditional*]

clause = *element* **in** *oldList*

This mimics set buidler notation (if you've ever heard of that)

squared = [x ** 2 for x in numbers]

is the same as

squared = $\{x^2 \mid 1 \leq x \leq 10\}$

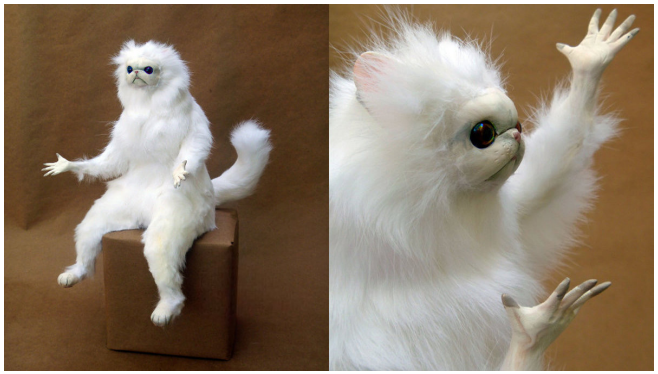
```
# With List Comprehension

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squares = [x ** 2 for x in numbers]

print(squares)

# alternatively

squares = [x**2 for x in range(1, 11)]
print(squares)
```

Let’s explore more sophisticated constructions in Google Colab

- Conditional Statements
- Additional Syntax
- Dealing with Tuples
- Nested Loops
- Time Analysis

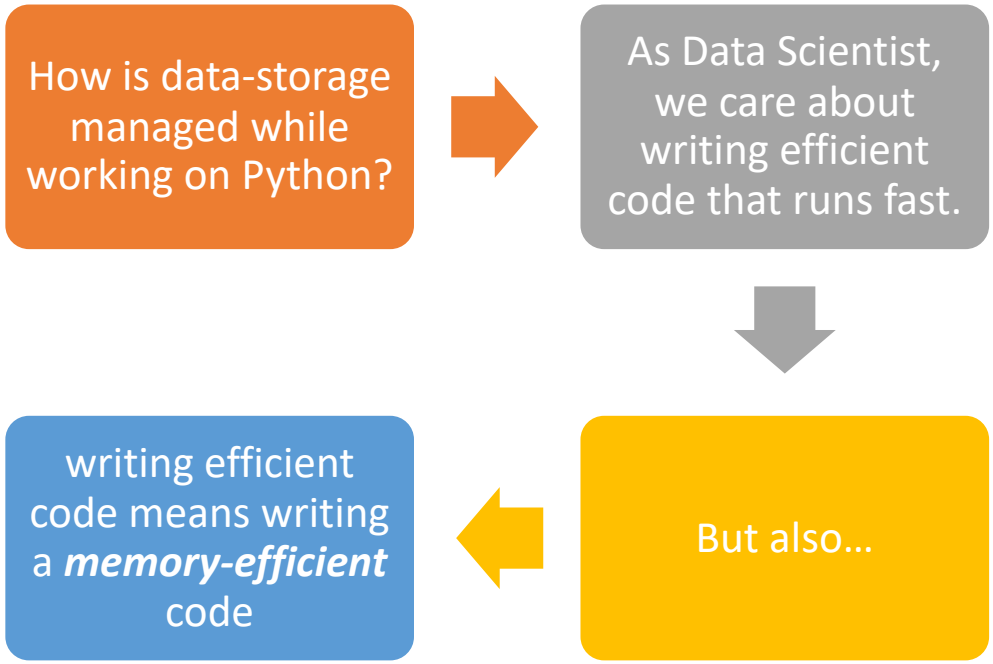
List Comprehension

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…



Memory Management for Python

- CPython
  - Reference implementation of Python
  - All new language features start here
  - Other implementations exist for certain use cases
- Private Heap:
  - All Python objects/data reside in a private heap
  - Managed internally by the Python Memory Manager
- Memory Allocation Layers:
  - Raw allocator interacts with OS to reserve heap memory
  - Object-specific allocators manage memory for types like int, str, dict, etc.
- Garbage Collection:
  - Frees memory no longer in use
  - Object is deallocated when ref count = 0
- Per-Object Memory Handling:
  - Each object has its own allocator & deallocator
- Global Interpreter Lock (GIL):
  - Only one thread can execute Python bytecode at a time when accessing shared data

Memory Management for Python (a company analogy)

- CPython (Company Headquarters):
  - Reference implementation of Python
  - All new language features start here
  - Other implementations exist for certain use cases
- Private Heap:
  - All Python objects/data reside in a private heap (a company's private warehouse)
  - Managed internally by the Python Memory Manager (architect overseeing project)
- Memory Allocation Layers:
  - Raw allocator interacts with OS to reserve heap memory (bricks for warehouse)
  - Object-specific allocators manage memory for types like int, str, dict, etc. (rooms in warehouse)
- Garbage Collection:
  - Frees memory no longer in use (cleanup crew)
  - Object is deallocated when ref count = 0 (tool/material sign-out sheet)
- Per-Object Memory Handling:
  - Each object has its own allocator & deallocator (type-specific storage bins in each room)
- Global Interpreter Lock (GIL):
  - Only one thread can execute Python bytecode at a time when accessing shared data (single checkout line, safe but slow)

Memory Management for Python

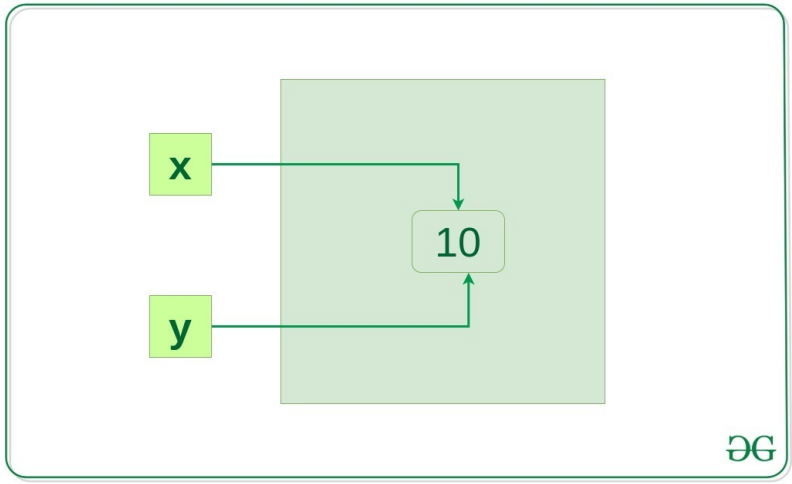
```python
# The identity of 4 objects
import math
a = 10
b = 10
c = 11.49274
d = math.pi

print('a:', id(a))
print('b:', id(b))
print('c:', id(c))
print('d:', id(d))
```

```
a: 10758024
b: 10758024
c: 136767239146928
d: 136767667016624
```

id(*object*) = memory address (identity) of an object

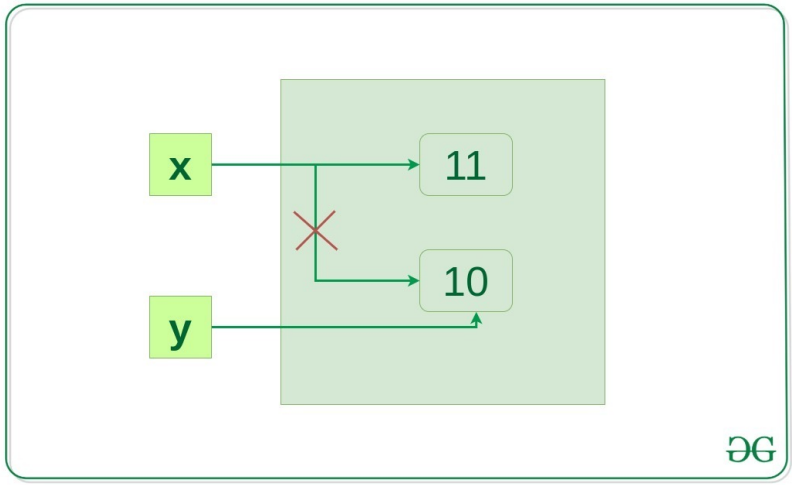
```python
x = 10
y = x

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
```



```python
# Integers

x = 10
y = x
print('x:', id(x))
print('y:', id(y))
x += 1
print('x:', id(x))
```

```
x: 10758024
y: 10758024
x: 10758056
```

```python
# Strings
print('Strings: ')
s1 = 'ABC'
s2 = 'ABC'
s3 = s1
print('s1:', id(s1))
print('s2:', id(s2))
print('s3:', id(s3))
```

```
Strings:
s1: 10155328
s2: 10155328
s3: 10155328
```

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

```
Tuples:
tu1 136767239330112
tu2 136767239330048
tu3 136767239330112
tu4 136767239330112
tu4 136767239330112
```

Lambda Functions

- A **lambda function** is a small, anonymous function in Python.
- Created using the `lambda` keyword.
- lambda expression syntax: `lambda parameters: expression`
- Returns the result of the expression automatically (no return needed).
- **Promotes abstraction**, key to programming— you focus on *what* the function does, not *how* it works internally.

Lambda Functions

Advantages

- **Concise**: Define simple functions in a single line.
- **Anonymous**: No need to name them if used once.
- **Flexible**: Pass directly into higher-order functions.
- **Readability**: Useful when function logic is short and obvious.

Lambda Functions

The following terms may be used interchangeably depending on the programming language type and culture:

- Anonymous functions
- Lambda functions
- Lambda expressions
- Lambda abstractions
- Lambda form
- Function literals

Functional Style vs Lambda Expression

```
# Functional Style
def quadratic(x):
  return(x**2 + x + 1)

quad1 = quadratic(3)
print(quad1)

# Lambda Function where quad2 is the expression
quad2 = lambda x: x**2 + x + 1
print(quad2(3))

# Lambda Function where quad3 is the output
quad3 = (lambda x: x**2 + x + 1)(3)
print(quad3)
```

Keyword: lambda

Parameters(s): x

Expression: x**2 + x + 1

How to Use Underscore in Notebooks

```
[23] # How to use _ in notebooks to store previous expression

     2 + 5
```

```
7
```

Use an underscore if you want to take this and manupulate further

```
[24] _*9
```

```
63
```

```
[28] (lambda x: x**2 + x + 1)(3)
```

```
13
```

```
[25] lambda x: x**2 + x + 1
```

```
<function __main__.<lambda>(x)>
```

```
[26] _(3)
```

```
13
```

Syntax to evaluate directly:

```
(lambda function)(input)
```

Convert a String using a lambda expression

```
] str1 = 'You miss 100% of the shots you don’t take. — "Wayne Gretzky" — Michael Scott'

  upper = lambda string: string.upper()
  print(upper(str1))

  title = lambda string: string.title()
  print(title(str1))

  split = lambda string: string.split(sep=" ")
  print(split(str1))
```

```
  YOU MISS 100% OF THE SHOTS YOU DON’T TAKE. — "WAYNE GRETZKY" — MICHAEL SCOTT
  You Miss 100% Of The Shots You Don’T Take. — "Wayne Gretzky" — Michael Scott
  ['You', 'miss', '100%', 'of', 'the', 'shots', 'you', 'don’t', 'take.', '—', '"Wayne', 'Gretzky"', '—', 'Michael', 'Scott']
```

Returning a Lambda in a Function

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))

# What are the printed results?
```

Multi-parameter Lambda Expressions

```
[45] # The function-based approach
     def add_two( x, y ):
       return( x + y )

     add_two( 2, 3 )
```

```
[44] # The lambda approaches
     add_two = (lambda x, y: x + y)(2, 3)
     print(add_two)

     add_two = lambda x, y: x + y
     print(add_two(2, 3))
```

Higher Order Functions

- A **higher-order function** is a function that **takes other functions as arguments** or **returns a function** as a result.

Lambda expressions with higher-order functions

```
0] # For high_ord_func, we make one of the parameters x and the other func
   # x will be a number and func will be another lambda expression
   high_ord_func = lambda x, func: x + func(x)

0] # Call the high_ord_func with x = 2 and func = another lambda
   high_ord_func(2, lambda x: x * x)

   # What will the result be?
```

If Statements in Lambda Expression

```
limit_alcohol = lambda age: 'Yes, be careful...' if age >= 21 else 'No, please wait...'

print( limit_alcohol(19) )

print( limit_alcohol(26) )
```

```
No, please wait...
Yes, be careful...
```

Some Syntax Limitations

```
# Instead of
(lambda x: a = 3 + x)(4)

# Try
a = (lambda x: 3 + x)(4)
```

```
# Instead of
(lambda y: return y == 3)(3)

# Try
(lambda y: y == 3)(3)
```

Practice

```
# arguments can be entered positionally or as keywords
ans = (lambda x, y, z: x + y + z)(1, 2, 3)

ans = (lambda x, y, z=3: x + y + z)(y=1, x=2)

ans = (lambda x, y, z=3: x + y + z)(1, 2, 4)

# collects all arguments
ans = (lambda *args: sum(args))(1,2,3)

# forced keyword args
ans = (lambda **kwargs: sum(kwargs.values()))(x=1, y=2, z=3)

# forced keyword args for y and z
ans = (lambda x, *, y=1, z=2: x + y + z)(1, y=2, z=3)

ans = (lambda x, *, y=1, z=2: x + y + z)(4)

ans = (lambda x, *, y=1, z=2: x + y + z)(4, y=3)

ans = (lambda x, *, y=1, z=2: x + y + z)(x=4, z=3)
```

Modify the sorting execution/behaviour

```
[78] # List of IDs
     ids = ['id1', 'id2', 'id100', 'id30', 'id3', 'id22']
```

```
[82] # Lexicographic Sort
     sorted(ids)
```

```
[84] # Extracting the number from each id

     (lambda x: int(x[2:]))(ids[3])
```

```
[83] # Sorting based on the integer value

     sorted(ids, key = lambda x: int(x[2:]) )
```

Some Examples

Find the second largest number in each list from a multi-list

```
] multiList = [ [3, 2, 1, 4], [1, 15, 7, 64], [13, 6, 9, 12, 7] ]
```

```
multiList = [ [3, 2, 1, 4], [1, 15, 7, 64], [13, 6, 9, 12, 7] ]
```

```
# Version 1: Two Lambda Functions + List Comprehension

# Sort each sublist with list comprehension (slower)
# Sorts all at once
sortList = lambda x: [sorted(i) for i in x]

# Get the second largest element
secondLargest = lambda x, f : [y[-2] for y in f(x)]
secondLargest(multiList, sortList)
```

```
multiList = [ [3, 2, 1, 4], [1, 15, 7, 64], [13, 6, 9, 12, 7] ]
```

```
# Version 2: Two Lambda Functions + List Comprehension + Generator Expression

# Sort each sublist with generator expression (faster)
# Sorts 1 at a time, on demand
sortList = lambda x: (sorted(i) for i in x)

# Get the second largest element
secondLargest = lambda x, f : [y[-2] for y in f(x)]
secondLargest(multiList, sortList)
```

```
multiList = [ [3, 2, 1, 4], [1, 15, 7, 64], [13, 6, 9, 12, 7] ]
```

```
# Version 3: One Lambda Function + List Comprehension

# Sort each list and extracting
sortList = [sorted(lis) for lis in multiList]

# Get the second largest number
secondLargest = lambda x: [y[-2] for y in x]
secondLargest(sortList)
```

```
multiList = [ [3, 2, 1, 4], [1, 15, 7, 64], [13, 6, 9, 12, 7] ]


# Version 4: List Comprehension

# Sort each list using only list comprehension
[sorted(lis)[-2] for lis in multiList]
```