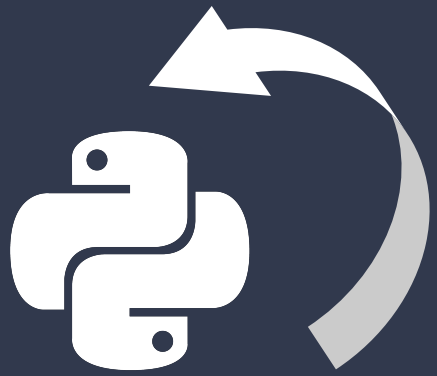
Explanation

Things to remove: Unnecessary else statements

```
if season == SUMMER:
    if moisture < DRY_SOIL_SUMMER:
        if forecast == 'no rain expected':
            return True
        return False
    return False
elif season == AUTUMN:
    if moisture < DRY_SOIL_AUTUMN_SPRING:
        return True
    return False
elif season == SPRING:
    if moisture < DRY_SOIL_AUTUMN_SPRING:
        return True
    return False
return False
```

CSCA08 Midterm Review Seminar



Nov 4th, 2025
18:00 – 20:00

Course Reps: Kevin, Zohair Syed, Krisha Patel, Nathanael Kevin Sitepu

PRESENTED BY

AMACSS
The Association of Mathematical and Computer Science Students

What would you like us to cover first?

Please fill out this poll to rank the topics you want us to review.

You can scan the QR code to the right or go to this link:
https://www.menti.com/aln5t7odibcj

Or you can go to menti.com and enter to code **8660 7727**

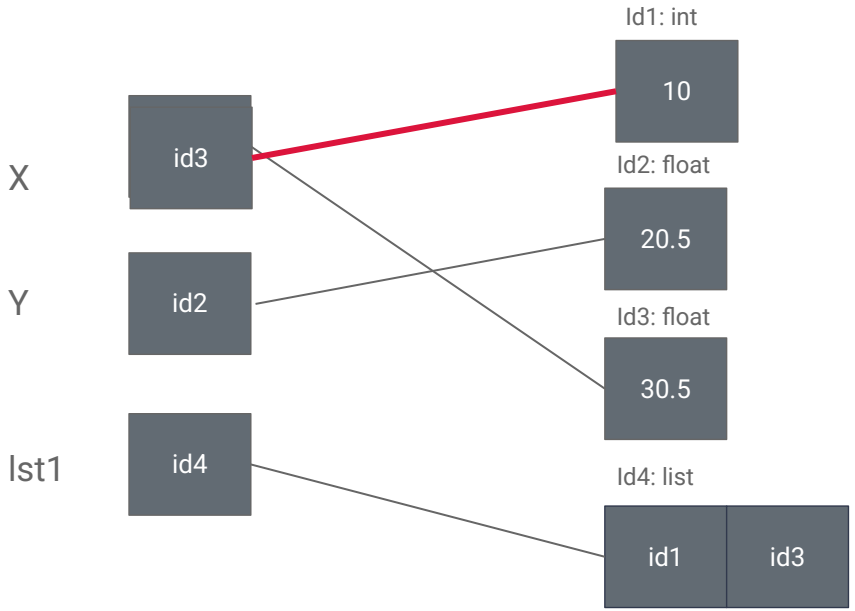


Part 1: Variables, Design Recipe & Conditional Statements

Memory Model

A way of representing current execution of code. Think of it as a big ‘box’ with lockers, each containing a value and memory address.

```
x = 10
y = 20.5
x = x + y
lst1 = [x, y]
```



Aliasing

When multiple variables refer to the same object in your memory.

In these example ‘b’ is our Alias

```
a = "Hello World"
b = a
```

What happens when we try to modify an alias(in this case ‘b’)?

- A new object gets created if we are dealing with Immutable data types.
- The same object gets modified dealing with mutable data types.

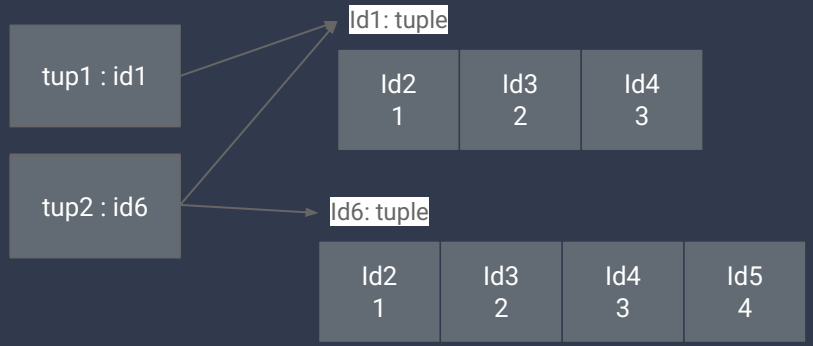
Some built-in methods might be an exception to this rule of thumb.

Refer to the helpsheet if you encounter an unfamiliar built-in method.

Aliasing in Immutable object

```
1  tup1 = (1,2,3)
2  tup2 = tup1
3  tup2 += (4,)
```

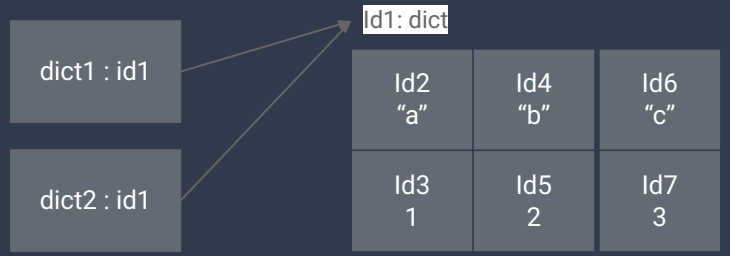
After executing line 3



Aliasing in Mutable object

```
1  dict1 = {"a": 1, "b" : 2}
2  dict2 = dict1
3  dict2["c"] = 3
```

After executing line 3



Try it yourself!

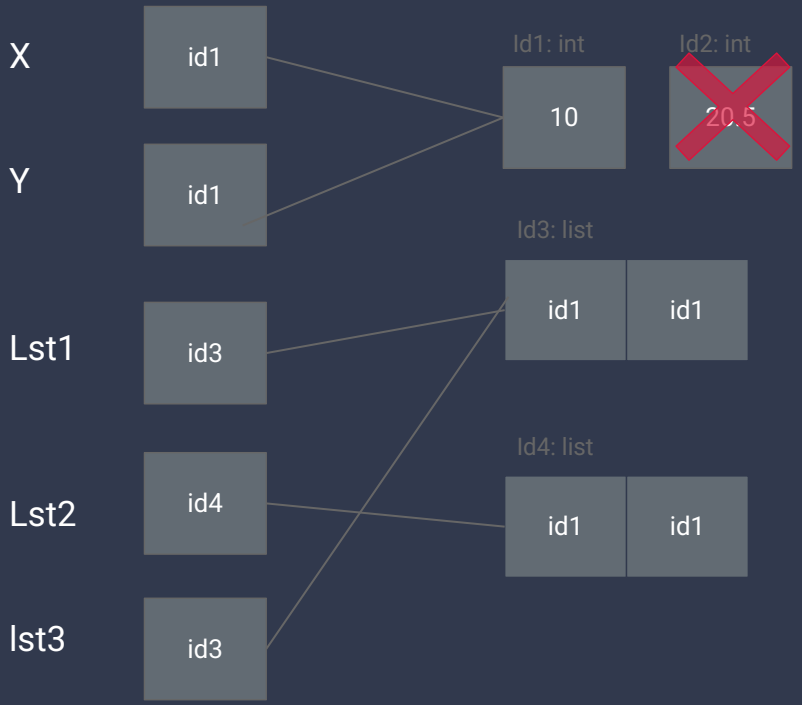
What is true about the following piece of code?

```
x = 10
y = 20.5
y = x
lst1 = [x, y]
lst2 = [x, y]
lst3 = lst1
```

a) lst1 and lst2 refer to the same ID in memory
b) lst3 and lst2 refer to the same ID in memory
c) lst2[1] and lst3[1] refer to the same ID in memory
d) x and y refer to same ID in memory

Try it yourself!

What is true about the following piece of code?



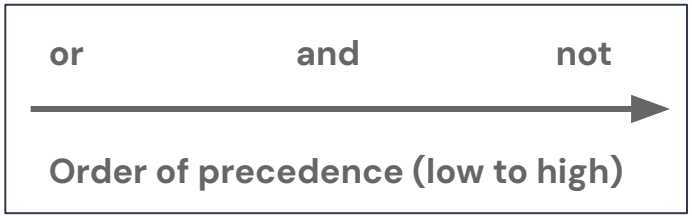

```
x = 10
y = 20.5
y = x
lst1 = [x, y]
lst2 = [x, y]
lst3 = lst1
```

a) lst1 and lst2 refer to the same ID in memory
b) lst3 and lst2 refer to the same ID in memory
c) lst2[1] and lst3[1] refer to the same ID in memory
d) x and y refer to same ID in memory

Boolean & Comparison Operators

Booleans hold a 'True' or 'False' value in Python (remember to capitalize!).
We can use operators, which evaluate to a boolean. This is useful in our *conditional statements*

Comparison Operators	***<, >, ==, !=, >=, <=***
Logical operators	keywords ***and, or, not***




```
x = 10

x > 100              # False
8 < x < 20           # True

not x >= 10          # not (True) -> False
x == 10 and x > 9    # True and True -> True
x > 10 or x < 11     # False or True -> True
```

If Statements & Conditionals

```
if condition:
    print("condition is true")
elif condition2:
    print("condition is false, condition2 is true")
else:
    print("condition and condition2 are false")
```

- **If**: checks the *first* condition. If it's True, the code inside runs.
- **elif (else if)**: checks the *next* condition *only if* the previous ones were False.
- **else**: runs only if *all* previous conditions were False.

- We often use the ***if, elif, else*** sequence when amongst many conditions, we only want one statement/block to execute
- It is best practice to reduce the #of if/elif/else blocks as we can for readability!

Proper usage of if, elif and else

Poor usage

```
def func(x:int) -> bool:
    if x == 10:
        return True
    else:
        return False
```

Improper use of if and else.

As we return early, we do not need the else statement at the end! ***We can write it in a more concise, and proper way.***

Better usage

```
def func(x:int) -> bool:
    if x == 10:
        return True
    return False
```

This is a better way of using conditionals. Note, when x == 10, we return True and *exit* from function.

When x != 10, we return ***False,*** behaving the same as the block of code on the left.

Proper usage of if, elif and else

Optimal solution!

There's even a better way to write our function without if!

- We want to return **TRUE** iff x == 10
  - Note that the statement ***x == 10*** evaluates to a **TRUE** or **FALSE** value
  - As we are returning a boolean, we can simply return that statement
- It is best practice to reduce your conditional statements

```
def func(x:int) -> bool:
    return x == 10
```

Example

Which pieces of code are equivalent? (i.e. produce same output for all values of x)

A) (1) and (4) B) (3) and (4) C) (1) and (2) D) None

1

```
if x == 100 and x >= 99:
    print("GPA is high")
    print("GPA is high - 1")
else:
    print("GPA is something")
```

2

```
if x == 100:
    print("GPA is high")
    if x >= 99:
        print("GPA is high - 1")
    else:
        print("GPA is something")
```

3

```
if x == 100:
    print("GPA is high")
elif x >= 99:
    print("GPA is high - 1")
else:
    print("GPA is something")
```

4

```
if x == 100:
    print("GPA is high")
if x >= 99:
    print("GPA is high - 1")
elif x < 99:
    print("GPA is something")
```

Solution

Which pieces of code are equivalent?

A) (1) and (4) B) (3) and (4) C) (1) and (2) D) None

1

```
if x == 100 and x >= 99:
    print("GPA is high")
    print("GPA is high - 1")
else:
    print("GPA is something")
```

2

```
if x == 100:
    print("GPA is high")
    if x >= 99:
        print("GPA is high - 1")
    else:
        print("GPA is something")
```

X	1	2
101	“GPA is something”	N/A
100	“GPA is high” “GPA is high -1”	“GPA is high” “GPA is high -1”
99	“GPA is high” “GPA is high -1”	“GPA is high” “GPA is high -1”
50	“GPA is something”	N/A

Solution

Which pieces of code are equivalent?

A) (1) and (4) B) (3) and (4) C) (1) and (2) D) None

3

```
if x == 100:
    print("GPA is high")
elif x >= 99:
    print("GPA is high - 1")
else:
    print("GPA is something")
```

4

```
if x == 100:
    print("GPA is high")
if x >= 99:
    print("GPA is high - 1")
elif x < 99:
    print("GPA is something")
```

X	3	4
101	“GPA is something”	N/A
100	“GPA is high”	“GPA is high” “GPA is high -1”
99	“GPA is high”	“GPA is high” “GPA is high -1”
50	“GPA is something”	“GPA is something”

Solution

Answer: None are equivalent!

X	1	2	3	4
102	“GPA is something”	N/A	“GPA is something”	N/A
100	“GPA is high” “GPA is high -1”	“GPA is high” “GPA is high -1”	“GPA is high”	“GPA is high” “GPA is high -1”
99	“GPA is high” “GPA is high -1”	“GPA is high” “GPA is high -1”	“GPA is high”	“GPA is high” “GPA is high -1”
50	“GPA is something”	N/A	“GPA is something”	“GPA is something”

Strings and String Operations

Are **IMMUTABLE!!!!!** Please remember this; don't be doing str[5] = 'a' !!!!!
Strings can be treated as a ***list*** of characters. This means you can access a character at a specific index (zero-indexed).

Operator	Description
str[i]	Access char at position i
len(x)	Length of string
x + y	Concatenate
x*y (y is integer)	Concat y copies of x
sub in **str**	Checks if **sub** is a substring in **str**

```
x = "hel"
y = "lo"
print(x[2]) # prints "l"
len(x)      # length of x is 3

x = x + y # "hello"
x*=3      # "hellohellohello"
"hel" in x #True
```

String Slicing

An important property is that you can slice strings...
Say I have a string **x="Hello world!"**

-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
H	e	l	l	o		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

I can cut it using this...



String Slicing

s[start:stop:stride]

- **Start and stop are the starting and stopping indices (with ‘stop’ being EXCLUSIVE)**

- **Stride** refers to the step value used in slicing strings
  - **POSITIVE STRIDE:** default values of start and stop are (start = 0, stop: len(s))
  - **NEGATIVE STRIDE:** default values of start and stop are (start = -1, stop: -(1+len(s))

- **Negative stride…**
  - When stride is negative, you reverse the direction of ‘iteration’. You start from the end of the string!

****Think about it: what if stride is zero?**

String Slicing

An important property is that you can slice strings…
Say I have a string **x=”Hello World!”**

X =

-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
H	e	l	l	o		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

x[0:5] # “Hello”
x[0:6] # “Hello “

x[-5:] # “orld!”
x[0:-1] # “Hello World”

x[::-1] # “!dlorW olleH”
x[::-2] # “!lo leH” #every two char

x[0:5:2] # “Hlo”
x[10:4:-1] # “dlroW ”

String Slicing

Let's breakdown **x[10:4:-1] # "dlroW"**

X =

-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
H	e	l	l	o		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

Last index (4) exclusive

" World" -> Reverse it -> "dlorW "

Try it yourself!

Given this string, produce the outputs with the given slices...

-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
A	M	A	C	S	S		2	K	2	5	!
0	1	2	3	4	5	6	7	8	9	10	11

This one
Is a bit
interesting!

x[0:len(x)]
x[0:]

x[0:200]

x[-9:-8]
X[-9:-9]

x[::2]
x[-1:-6:-1]
x[3:6:0]

Try it yourself!

Given this string, produce the outputs with the given slices...

-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
A	M	A	C	S	S		2	K	2	5	!
0	1	2	3	4	5	6	7	8	9	10	11

This one
Is a bit
interesting!

x[0:len(x)] # “AMACSS 2K25!”
x[0:] # “AMACSS 2K25! “

x[0:200] # “AMACSS 2K25!”

x[-9:-8] # “C”
X[-9:-9] # “”

x[::2] # “AAS K5”
x[-1:-6:-1] # “!52K2”
X[3:6:0] VALUE ERROR!! (zero stride is not allowed!!!!)

Design Recipe

```
def function_name(parameter1_name: parameter_type1,
                  parameter2_name: parameter_type2) -> return_type:
    """
    # Description

    # Docstring Testing
    >>> function_name(a, b)
    # anticipated correct value for that function call

    >>> function_name(c, d)
    # anticipated correct value for that function call

    """

    # Function Body

    return #something with the val of return type
```

Ensure to describe..

- What is returned
- Preconditions
- What the function does
    - Desc. should include parameter names
    - Be detailed, yet concise

Tip: Try to start your description with 'Return...' and then continue your description. (Doesn't always apply)

Design Recipe

```
def function_name(parameter1_name: parameter_type1,
                  parameter2_name: parameter_type2) -> return_type:
    """
    # Description

    # Docstring Testing
    >>> function_name(a, b)
    # anticipated correct value for that function call

    >>> function_name(c, d)
    # anticipated correct value for that function call

    """

    # Function Body

    return #something with the val of return type
```

DO NOT...

- Describe how the function works
- Be too long with your description.

Example

TDLR: This function takes a string, replaces an instance of a given letter with ":D"

```
def replace_letter(replace_letter: str, word: str) -> str:
    """
        Return the modified version of the string _word_ with all instances of the
        letter _replace_letter_ replaced with the string ":D"

        Precondition: len(replaceLetter) = 1

        >>> replace_letter("h", "XDXDXD")
        "XDXDXD"
        >>> replace_letter("3", "")
        ""
        >>> replace_letter("e", "heello")
        "h:D:Dllo"
    """
```

Example - Docstrings

```
def replace_letter(replace_letter: str, word: str) -> str:
    """
        Return the modified version of the string _word_ with all instances of the
        letter _replace_letter_ replaced with the string ":D"

        Precondition: len(replaceLetter) = 1

        >>> replace_letter("h", "XDXDXD")
        "XDXDXD"
        >>> replace_letter("3", "")
        ""
        >>> replace_letter("e", "heello")
        "h:D:Dllo"
    """
```

When letter is not in string

String is empty

1 or more instance of letter in string

Your docstring tests should...

- Have around 3~4 (or how many the question permits you to have)
- Cover all cases; regular cases, edge cases
- DO NOT BE REPETITIVE WITH CASES!!

Try it yourself!

Problem Statement: Osmows UTSC is offering a discount!

UTSC-students (denoted as 'UTSC') can get a ***discount $3 off***, while other UofT students (denoted 'UofT') can get a ***discount of $1 off.*** A non UofT student (denoted as 'N/A') gets no discount.

The following function below applies the discount to the price based on the student status ***(with the possible statuses being "UTSC", "UofT" and "N/A")*** and returns True **IFF** the discounted price is $5 or less, and the student goes to UofT.

```
def osmows_discount(price: FILL_IN, student_type: FILL_IN) -> FILL_IN:
    """
        Description

        Preconditions?

        >>> test 1

        >>> test 2

        >>> test 3

    """
```

Sample Soln

```
def osmows_discount(price: float, student_type: str) -> bool:
    """
        Return True iff the student, _student_type_ is from UofT
        (i.e. student_type == "UofT" or student_type == "UTSC") AND
        the given price, _price_ is less than or equal to 5 dollars after applying
        a discount, based on the type of student.

        Preconditions:
            student_type = "UofT" OR "UTSC" OR "N/A"
            price >= 0

        >>> osmows_discount(44, "N/A")
            False

        >>> osmows_discount(55, "UTSC")
            False

        >>> osmows_discount(6, "UofT")
            True

    """
```

Part 2: Lists, List and String Methods, Loops

Lists

A data type that can store a **collection of elements** under one variable

```
# A list of integers
a = [1, 2, 3, 4]

# Elements of a list can be of different types
b = ["one", 2, "three", 4]

# An example of list-nesting
c = [[1], [1, 2], [1, 2, 3]]
```

```
# Modifying a list:
a[1] = 99
print(a) # [1, 99, 3, 4]

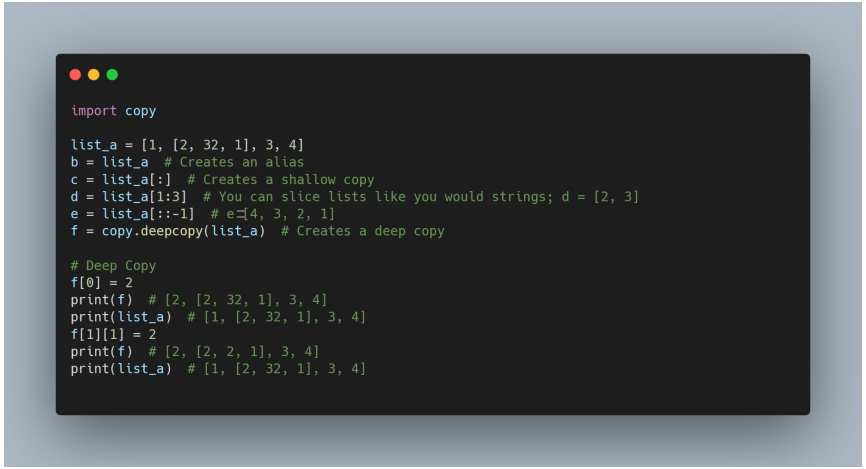
# An example of aliasing
fake_b = b
fake_b[0] = "one million"
print(b) # ["one million", 2, "three", 4]

# Accessing elements in a nested list
print(c[0][0]) # 1
print(c[2][2]) # 3
print(c[0][3]) # IndexError; c[0] is a list with only one element
```

Lists - Continued

Shallow Copy: if b is a shallow copy of a, a and b are different objects in memory, but their elements refer to the **same** objects in memory.

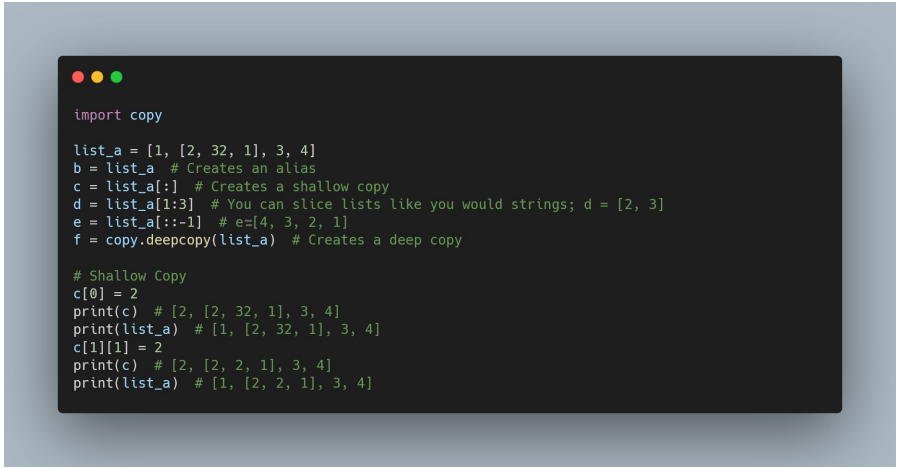
Deep Copy: if b is a deep copy of a, and b are different objects in memory, and their elements refer to **different** objects in memory.



```python
import copy

list_a = [1, [2, 32, 1], 3, 4]
b = list_a  # Creates an alias
c = list_a[:]  # Creates a shallow copy
d = list_a[1:3]  # You can slice lists like you would strings; d = [2, 3]
e = list_a[::-1]  # e=[4, 3, 2, 1]
f = copy.deepcopy(list_a)  # Creates a deep copy

# Deep Copy
f[0] = 2
print(f)  # [2, [2, 32, 1], 3, 4]
print(list_a)  # [1, [2, 32, 1], 3, 4]
f[1][1] = 2
print(f)  # [2, [2, 2, 1], 3, 4]
print(list_a)  # [1, [2, 32, 1], 3, 4]
```

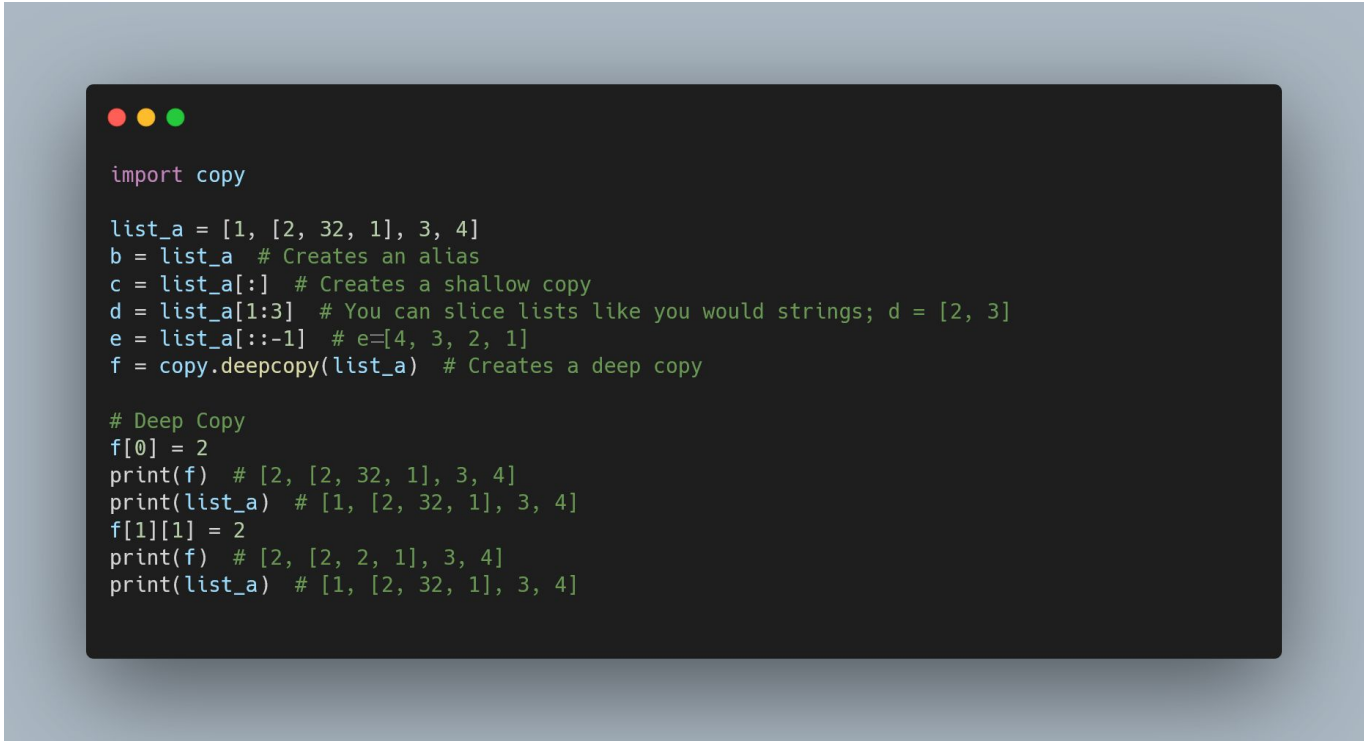


```python
import copy

list_a = [1, [2, 32, 1], 3, 4]
b = list_a  # Creates an alias
c = list_a[:]  # Creates a shallow copy
d = list_a[1:3]  # You can slice lists like you would strings; d = [2, 3]
e = list_a[::-1]  # e=[4, 3, 2, 1]
f = copy.deepcopy(list_a)  # Creates a deep copy

# Shallow Copy
c[0] = 2
print(c)  # [2, [2, 32, 1], 3, 4]
print(list_a)  # [1, [2, 32, 1], 3, 4]
c[1][1] = 2
print(c)  # [2, [2, 2, 1], 3, 4]
print(list_a)  # [1, [2, 2, 1], 3, 4]
```

Lists - Deep Copy Example

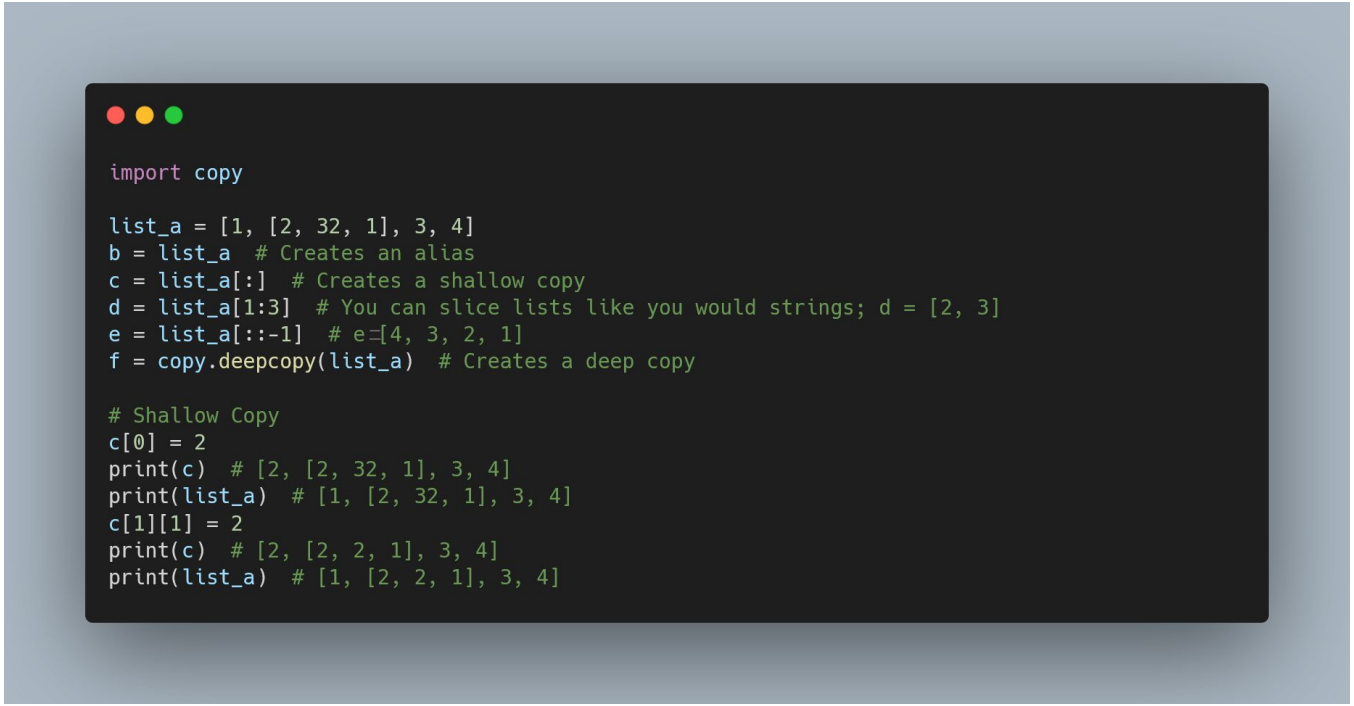


```python
import copy

list_a = [1, [2, 32, 1], 3, 4]
b = list_a  # Creates an alias
c = list_a[:]  # Creates a shallow copy
d = list_a[1:3]  # You can slice lists like you would strings; d = [2, 3]
e = list_a[::-1]  # e=[4, 3, 2, 1]
f = copy.deepcopy(list_a)  # Creates a deep copy

# Deep Copy
f[0] = 2
print(f)  # [2, [2, 32, 1], 3, 4]
print(list_a)  # [1, [2, 32, 1], 3, 4]
f[1][1] = 2
print(f)  # [2, [2, 2, 1], 3, 4]
print(list_a)  # [1, [2, 32, 1], 3, 4]
```

Lists - Shallow Copy Example



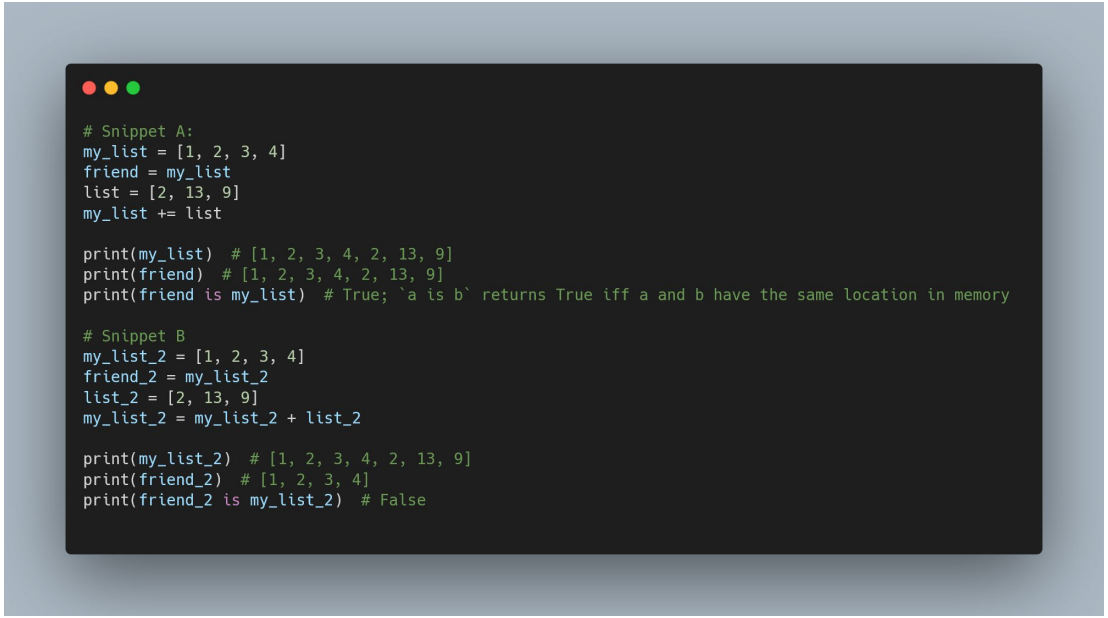
```python
import copy

list_a = [1, [2, 32, 1], 3, 4]
b = list_a  # Creates an alias
c = list_a[:]  # Creates a shallow copy
d = list_a[1:3]  # You can slice lists like you would strings; d = [2, 3]
e = list_a[::-1]  # e=[4, 3, 2, 1]
f = copy.deepcopy(list_a)  # Creates a deep copy

# Shallow Copy
c[0] = 2
print(c)  # [2, [2, 32, 1], 3, 4]
print(list_a)  # [1, [2, 32, 1], 3, 4]
c[1][1] = 2
print(c)  # [2, [2, 2, 1], 3, 4]
print(list_a)  # [1, [2, 2, 1], 3, 4]
```

Lists - Continued

Some common stuff that trips a lot of people up:



```
# Snippet A:
my_list = [1, 2, 3, 4]
friend = my_list
list = [2, 13, 9]
my_list += list

print(my_list)  # [1, 2, 3, 4, 2, 13, 9]
print(friend)  # [1, 2, 3, 4, 2, 13, 9]
print(friend is my_list)  # True; `a is b` returns True iff a and b have the same location in memory

# Snippet B
my_list_2 = [1, 2, 3, 4]
friend_2 = my_list_2
list_2 = [2, 13, 9]
my_list_2 = my_list_2 + list_2

print(my_list_2)  # [1, 2, 3, 4, 2, 13, 9]
print(friend_2)  # [1, 2, 3, 4]
print(friend_2 is my_list_2)  # False
```

Snippet A:

`+=` **modifies** the list. Since friend is an alias, we can see the change being reflected.

Snippet B:

`my_list_2 + list_2` **creates a new list in memory**. Then, the assignment operator assigns this new list to `my_list_2`. `friend_2` still points to the old object in memory.

Lists (Example From Last Slide)

```python
# Snippet A:
my_list = [1, 2, 3, 4]
friend = my_list
list = [2, 13, 9]
my_list += list

print(my_list)  # [1, 2, 3, 4, 2, 13, 9]
print(friend)  # [1, 2, 3, 4, 2, 13, 9]
print(friend is my_list)  # True; `a is b` returns True iff a and b have the same location in memory

# Snippet B
my_list_2 = [1, 2, 3, 4]
friend_2 = my_list_2
list_2 = [2, 13, 9]
my_list_2 = my_list_2 + list_2

print(my_list_2)  # [1, 2, 3, 4, 2, 13, 9]
print(friend_2)  # [1, 2, 3, 4]
print(friend_2 is my_list_2)  # False
```

Reading Documentation

One of the important parts of reading Python documentation is being able to **understand the function signatures**.

```
func_name(param_1: type_1)
func_name(param_0: type_0, param_1: type_1[, param_2: type_2])
```

Sometimes, you'll get the same function defined with different parameters. This lets you call the function with different arguments. This concept is called **function overloading**. You'll learn more about in CSCA48!

Bracketed `[ ]` parameters are **optional.**

List and String Methods

Some common string methods can be found at:
https://docs.python.org/3/library/stdtypes.html#string-methods

Some common list methods can be found at:
https://docs.python.org/3/tutorial/datastructures.html

On last year's (Fall 2024) exams, we were **given a cheat sheet containing documentation** for various functions and methods.



List and String Methods - Example

1) Past Midterm Question

Consider the following code:

```
text = "The quick brown fox jumps over the lazy dog"
result = text.replace('e', '3').replace('a', '@').replace('i', '_')
```

What will the value of result be after these replacements?

A. `"Th3 qu_ck brown fox jumps ov3r th3 l@zy dog"`
B. `"Th3 qu1ck brown fox jumps ov3r th3 l@zy dog"`
C. `"Th3 quick brown fox jumps over the l@zy dog"`
D. `"Th3 qu_ck brown f0x jumps ov3r th3 l@zy dog"`
E. `Error`

List and String Methods - Example

1) Past Midterm Question - Answer

```
A.  "Th3 qu_ck brown fox jumps ov3r th3 l@zy dog"
B.  "Th3 qu1ck brown fox jumps ov3r th3 l@zy dog"
C.  "Th3 quick brown fox jumps over the l@zy dog"
D.  "Th3 qu_ck brown f0x jumps ov3r th3 l@zy dog"
E.  Error
```

Explanation

```
>>> "The quick brown fox jumps over the lazy dog".replace('e', '3')
'Th3 quick brown fox jumps ov3r th3 lazy dog'
>>> 'Th3 quick brown fox jumps ov3r th3 lazy dog'.replace('a', '@')
'Th3 quick brown fox jumps ov3r th3 l@zy dog'
>>> 'Th3 quick brown fox jumps ov3r th3 l@zy dog'.replace('i', '_')
'Th3 qu_ck brown fox jumps ov3r th3 l@zy dog'
```

List and String Methods - Example

2) Past Midterm Question

Consider the following code.

```
my_list = [1, 2, 3, 2, 4, 2]
my_list.remove(2)
my_list.reverse()
print(my_list)
```

What is the output in python shell after the code is executed?

A. `[4, 2, 3, 2, 1]`
B. `[4, 3, 2, 1]`
C. `[1, 3, 4, 2, 2]`
D. `[2, 4, 2, 3, 1]`
E. `None of the above`

List and String Methods - Example

2) Past Midterm Question - Answer

A. `[4, 2, 3, 2, 1]`
B. `[4, 3, 2, 1]`
C. `[1, 3, 4, 2, 2]`
D. `[2, 4, 2, 3, 1]`
E. None of the above

Explanation

```
>>> my_list = [1, 2, 3, 2, 4, 2]
>>> my_list.remove(2)
>>> my_list
[1, 3, 2, 4, 2]
>>> my_list.reverse()
>>> my_list
[2, 4, 2, 3, 1]
```

Loops

There are two main types of loops in Python: **while** loops and **for** loops

Some Terminology:

Loop body:

- Code that gets **executed during the loop**
- In Python, you must **indent** to indicate the loop body

```
# while loop example

while True:
  print("Please Help!")
```

```
# for-loop example
# prints "The sum is: 105"

sum = 0

for i in range(15):
  sum += i

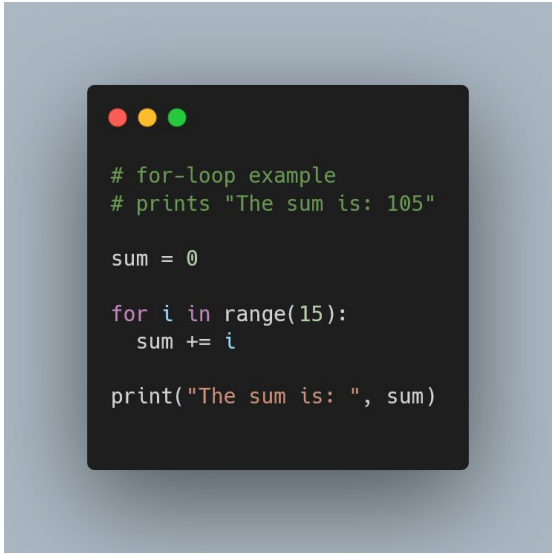
print("The sum is: ", sum)
```

For Loops

For loops let you iterate through each element in an **iterable object** using the `for … in` syntax.

The variable `i` denotes the element corresponding to the current iteration. The **scope** of `i` is limited to **only within the loop body**.

Iterable objects are objects with a finite number of elements; you can count the each element within an iterable object: lists, strings, and tuples are some examples.



```
# for-loop example
# prints "The sum is: 105"

sum = 0

for i in range(15):
  sum += i

print("The sum is: ", sum)
```

What's range?

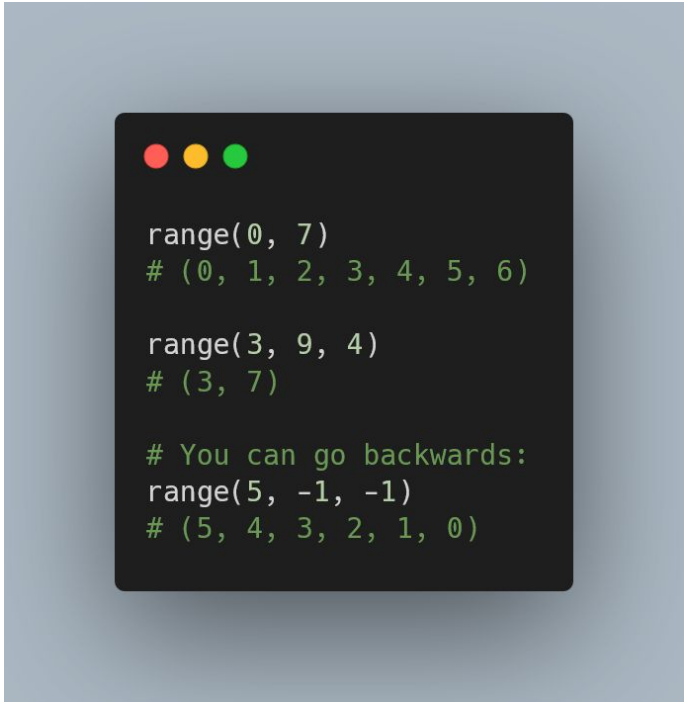
The `range` function lets you create a **range object**, which is an **immutable sequence** of numbers.

It has the following function signatures:

```
range(stop)
range(start, stop[, step])
```

The sequence **starts at** `start`, **increments by** `step`, and **ends just before** `stop`.

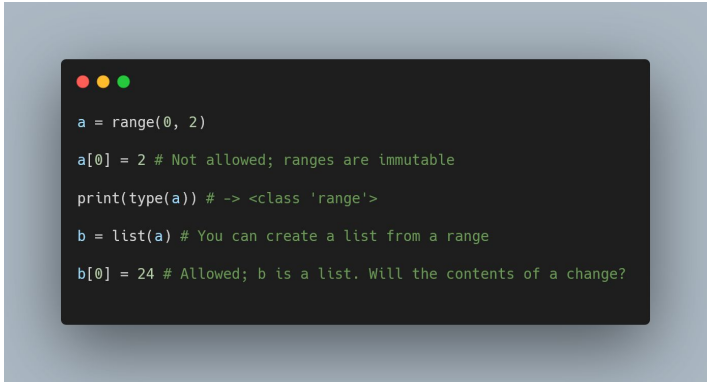
If `start` is omitted, the sequence starts at 0. If `step` is omitted, the sequence increments by 1.



`range` - Continued

Gotcha:

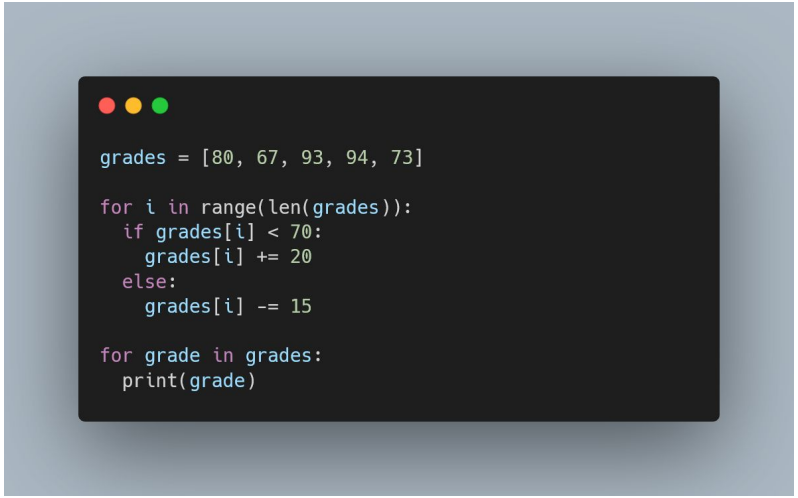
- A `range` is **not a list**
  - Like mentioned before, a range is an iterable object. **A list is an iterable object, but not all iterable objects are lists**



For Loops - Examples

What is the expected output of each program?

A)

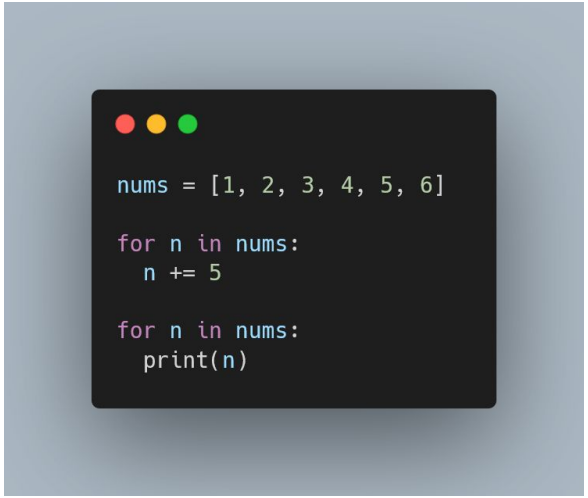


```
grades = [80, 67, 93, 94, 73]

for i in range(len(grades)):
  if grades[i] < 70:
    grades[i] += 20
  else:
    grades[i] -= 15

for grade in grades:
  print(grade)
```

B)

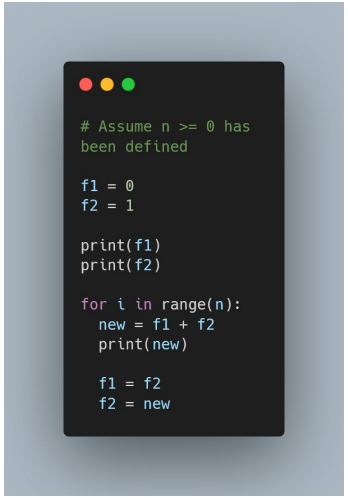


```
nums = [1, 2, 3, 4, 5, 6]

for n in nums:
  n += 5

for n in nums:
  print(n)
```

C)



```
# Assume n >= 0 has
been defined

f1 = 0
f2 = 1

print(f1)
print(f2)

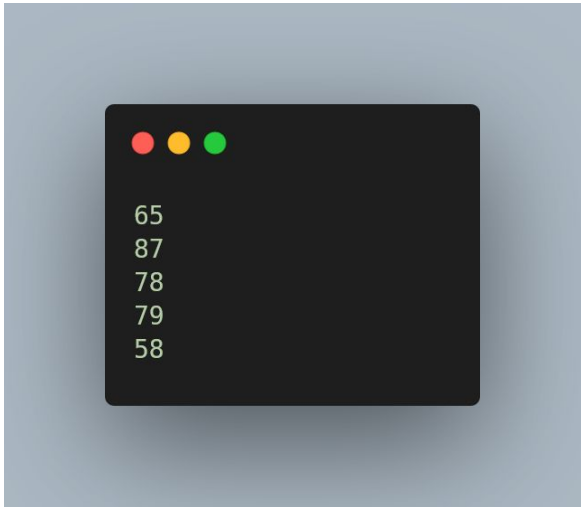
for i in range(n):
  new = f1 + f2
  print(new)

  f1 = f2
  f2 = new
```

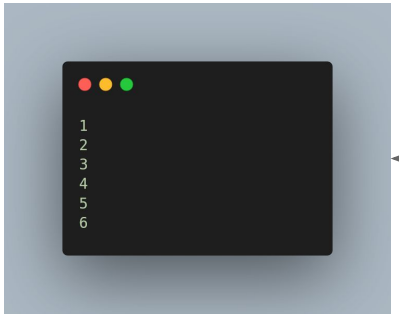
For Loops - Examples

Answers:

A)



B)



C)

The first n + 2 fibonacci numbers

The `for … in` syntax **creates a copy of each element**; the original array doesn't get modified.

If you are trying to mutate the list, use `range` to generate a sequence of indices, and use it to **index into the list**; see example A).

While Loops

While loops let you run code so long as a **certain condition is True.**

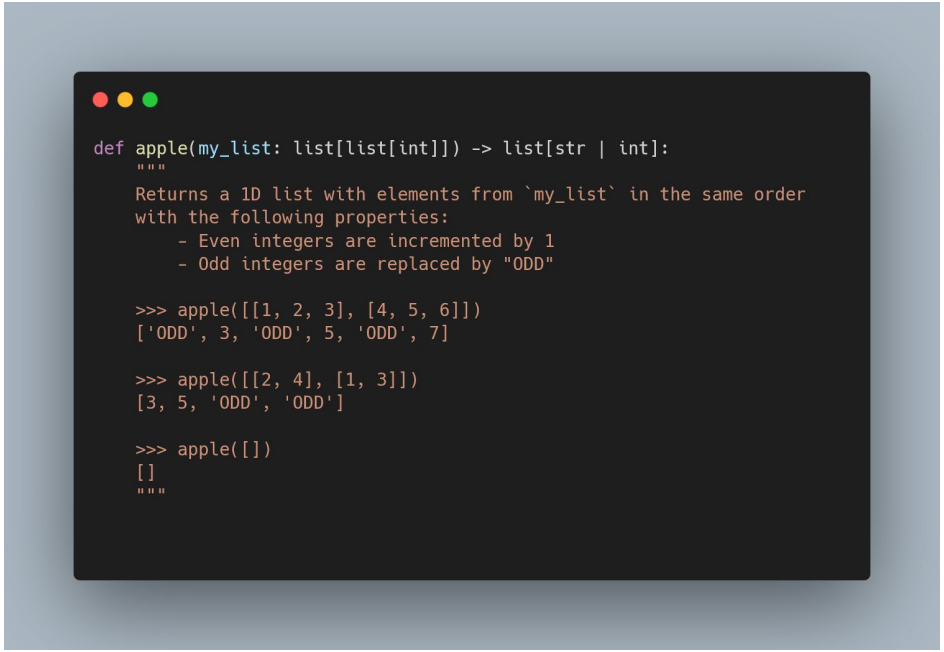
When the condition is **False, the loop terminates**; it no longer runs.



This loop will never terminate. Can you see why?

While Loops - Example

Given the following signature and docstring, implement the function. You **must use** a for loop and a while loop.



```
def apple(my_list: list[list[int]]) -> list[str | int]:
    """
    Returns a 1D list with elements from `my_list` in the same order
    with the following properties:
        - Even integers are incremented by 1
        - Odd integers are replaced by "ODD"

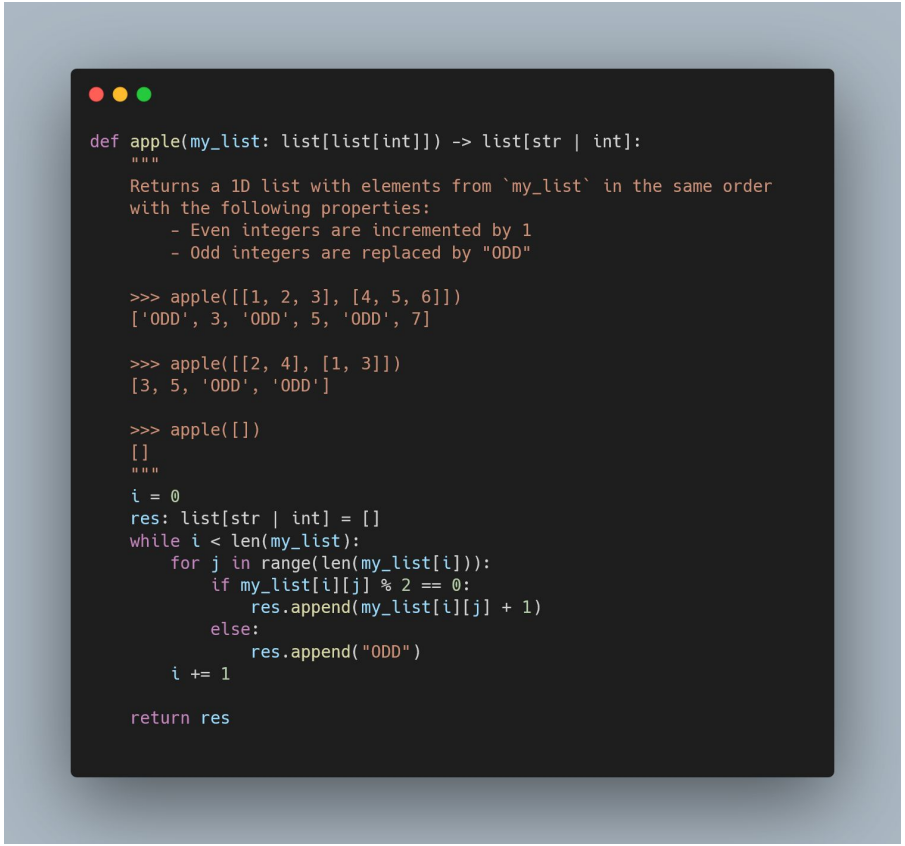
    >>> apple([[1, 2, 3], [4, 5, 6]])
    ['ODD', 3, 'ODD', 5, 'ODD', 7]

    >>> apple([[2, 4], [1, 3]])
    [3, 5, 'ODD', 'ODD']

    >>> apple([])
    []
    """
```

While Loops - Example

Answer:



```
def apple(my_list: list[list[int]]) -> list[str | int]:
    """
    Returns a 1D list with elements from `my_list` in the same order
    with the following properties:
        - Even integers are incremented by 1
        - Odd integers are replaced by "ODD"

    >>> apple([[1, 2, 3], [4, 5, 6]])
    ['ODD', 3, 'ODD', 5, 'ODD', 7]

    >>> apple([[2, 4], [1, 3]])
    [3, 5, 'ODD', 'ODD']

    >>> apple([])
    []
    """
    i = 0
    res: list[str | int] = []
    while i < len(my_list):
        for j in range(len(my_list[i])):
            if my_list[i][j] % 2 == 0:
                res.append(my_list[i][j] + 1)
            else:
                res.append("ODD")
        i += 1

    return res
```

Part 3: Understanding Tuples and Dictionaries

Introducing Tuples

An n-tuple is a grouped collection of n items, similar to a list, but:

- They are **immutable**
- Use `my_tuple = (1, 2, 3, 4)` syntax

Quick question: How do you sort the elements of a tuple?

A: Convert to a list first, then you can sort using the aforementioned list methods!

```
my_tuple = (4, 1, 3, 5, 2)




print(my_tuple)

# >>> (1, 2, 3, 4, 5)
```

Introducing Dictionaries

A dictionary is a mapping of **key-value pairs**, with many helpful built-in methods

```
d = {
    'a': ["apple", "apricot", "avocado"],
    'b': 1,
    'c': (5, 3, 4, 1, 2),
    'd': 2,
}

print(d['a'])


print(d.keys())
```

```
d["c"] = "New changed value"
print(d["c"])


d["e"] = "New added value"
print(d["e"])
```

Common Dictionary Techniques

- Iterate through values with .values()

```
groceries_to_price = {
    'apple': 0.5,
    'grapes': 2.0,
    'milk': 2.0,
    'bread': 2.5
}


for prices in groceries_to_price.values():
    print(prices)

# Output:
# 0.5
# 2.0
# 1.5
# 2.5
```

- Counting frequencies

```
groceries_to_price = {
    'apple': 0.5,
    'grapes': 2.0,
    'milk': 2.0,
    'bread': 2.5
}

price_freq = {}

for price in groceries_to_price.values():
    if price not in price_freq:
        price_freq[price] = 1
    else:
        price_freq[price] += 1
```

```
price_freq = {
    0.5: 1,
    2.5: 1,
    2.0: 2
}
```

Dictionary Question

Hints:

1. The keys (students) of *grades* aren't needed, use .values() to loop over sub-dicts

2. We can use a similar counting approach as the last slide

```
def subject_averages(grades: dict) -> dict:
    """
    Given a dictionary mapping each student to their grades in
    various subjects, return a new dictionary mapping each
    subject to its average grade across all students.

    >>> grades = {
    ...     "Alice": {"math": 90, "bio": 80},
    ...     "Bob":  {"math": 70, "bio": 85, "chem": 75},
    ...     "Cara": {"math": 95, "chem": 100}
    ... }
    >>> subject_averages(grades)
    {'math': 85.0, 'bio': 82.5, 'chem': 87.5}
    """

    pass
```

Solution

```python
grades = {
    "Alice": {
        "math": 90,
        "bio": 80
        },
    "Bob": {
        "math": 70,
        "bio": 85,
        "chem": 75
        },
    "Cara": {
        "math": 95,
        "chem": 100
        }
}
```

```python
def subject_averages(grades: dict) -> dict:
    course_to_avg = {} # {course: grade}
    course_freq = {} # {course: count}

    # First we sum up the total grades per class
    for dicts in grades.values():
        for course, grade in dicts.items():
            if course not in course_to_avg:
                course_to_avg[course] = 0
                course_freq[course] = 0

            course_to_avg[course] += grade
            course_freq[course] += 1

    # Now we can divide by the frequency to get the average
    for course in course_to_avg:
        course_to_avg[course] /= course_freq[course]

    return course_to_avg
```

Reminders

- Dictionaries are unordered collections. **Do not try to index them**
- '+=' and '+' methods aren't defined for Dictionaries(i.e you cannot add two dictionaries)
- There are only two methods defined on tuples:
  count() : Counts the number of occurrences of a particular element.
  index() : Looks for the first occurence of an element. Throws an error if the element is not in the tuple.
- A single element tuple is defined a little differently:
  >>> a = (1,)
  >>> b = (1)
  'a' is a tuple and 'b' is **not a tuple.** Then what is the type of 'b' here?

Part 4: Past Midterm Questions

Past Midterm Question

Question 1: [14 marks] Smart Sprinkler System Control Based on Season

In a smart garden system, the sprinkler is controlled based on the current **soil moisture level**, the **weather forecast**, and the **season**.

The sprinkler system should be turned on or off depending on the following conditions:

- In *summer*, the sprinkler turns on if the soil moisture is below 25% and it is not expected to rain.
- In *autumn or spring*, the sprinkler turns on if the soil moisture is below 15%, regardless of the weather.

The function should return **True** if the sprinkler is on, and **False** otherwise.

You are given the following constants, which ***you must use in your implementation***:

```
DRY_SOIL_SUMMER = 25.0  # Soil moisture threshold in summer
DRY_SOIL_AUTUMN_SPRING = 15.0  # Soil moisture threshold in autumn and spring
RAIN_EXPECTED = 'rain expected'
NO_RAIN = 'no rain expected'
SUMMER = 'summer'
AUTUMN = 'autumn'
SPRING = 'spring'
```

Assume the function works for valid `moisture, forecast and season`.
Meaning, `moisture` is a non-negative float, `forecast` is either `'rain expected'` or `'no rain expected'` and season is one of `'summer', 'autumn'`, or `'spring'`.

Part a [6 marks] Complete the documentation for the function

```
def is_sprinkler_on(moisture: float, forecast: str, season: str) -> bool:
    """ Return

    Preconditions:
```

Explanation

Things to Remember for Docstrings:

- Description
- Preconditions
- Tests

Description

- What does the function return?
- Explain how the return value changes according to the arguments
- Include parameter names

The function should return **True** if the sprinkler is on, and **False** otherwise.

Return True when the sprinkler should be turned on and False when the sprinkler should be turned off.

Explanation

Things to Remember for Docstrings:

- Description
- Preconditions
- Tests

Description

- What does the function return?
- Explain how the return value changes according to the arguments
- Include parameter names

In a smart garden system, the sprinkler is controlled based on the current **soil moisture level**, the **weather forecast**, and the **season**.

The sprinkler should be turned on or off based on 'moisture' that represents soil moisture level, 'forecast' that represents weather forecast and 'season'.

Explanation

Things to Remember for Docstrings:

- Description
- Preconditions
- Tests

Preconditions

- Explain any assumption that should be made to use the function
- Not binding

Assume the function works for valid moisture, forecast and season. Meaning, moisture is a non-negative float, forecast is either 'rain expected' or 'no rain expected' and season is one of 'summer', 'autumn', or 'spring'

moisture >= 0.0

The `forecast` is either RAIN_EXPECTED or NO_RAIN.

The `season` is one of SUMMER, AUTUMN or SPRING.

Explanation

Things to Remember for Docstrings:

- Description
- Preconditions
- Tests

Tests

- Follow assumption/precondition
- Test using different relevant arguments

```
>>> is_sprinkler_on(20.0, NO_RAIN, SUMMER)
True
>>> is_sprinkler_on(20.0, NO_RAIN, SUMMER)
True
>>> is_sprinkler_on(20.0, NO_RAIN, SUMMER)
True
>>> is_sprinkler_on(23.0, NO_RAIN, SPRING)
False
```

Soln

```
def is_sprinkler_on(moisture: float, forecast: str, season: str) -> bool:
    """ Return True when the sprinkler should be turned on and False when the
    sprinkler should be turned off. The sprinkler should be turned on or off
    based on 'moisture' that represents soil moisture level, 'forecast' that
    represents weather forecast and 'season'.

      Preconditions: moisture >= 0.0
     The `forecast` is either RAIN_EXPECTED or NO_RAIN.
     The `season` is one of SUMMER, AUTUMN or SPRING.

    >>> is_sprinkler_on(20.0, NO_RAIN, SUMMER)
    True
    >>> is_sprinkler_on(10.0, RAIN_EXPECTED, AUTUMN)
    True
    >>> is_sprinkler_on(20.0, NO_RAIN, AUTUMN)
    False
    >>> is_sprinkler_on(10.0, NO_RAIN, SPRING)
    True

"""
```

Past Midterm Question

In the function, `is_sprinkler_on()`, the provided implementation is correct (the function returns correct values), *but it is **not** written in good programming style*. Your task is to ***improve the code***. Recall that good code should be: Short, simple, and clear, avoids repeated computations, use good variable names, and use appropriate constants instead of hard-coded values.

Reminder of bad code to improve

```
def is_sprinkler_on(moisture: float, forecast: str, season: str) -> bool:
    """ [missing documentation]

    """
    if season == SUMMER:
        if moisture < DRY_SOIL_SUMMER:
            if forecast == 'no rain expected':
                return True
            else:
                return False
        else:
            return False
    elif season == AUTUMN:
        if moisture < DRY_SOIL_AUTUMN_SPRING:
            return True
        else:
            return False
    elif season == SPRING:
        if moisture < DRY_SOIL_AUTUMN_SPRING:
            return True
        else:
            return False
    else:
        return False
```

Explanation

Things to remove: Nested if statements

```
if (season == SUMMER and
        moisture < DRY_SOIL_SUMMER and
        forecast == 'no rain expected'):
    return True
elif (season == AUTUMN and
        moisture < DRY_SOIL_AUTUMN_SPRING):
    return True
elif (season == SPRING and
        moisture < DRY_SOIL_AUTUMN_SPRING):
    return True
return False
```

Explanation

Things to remove: Similar conditioning

```
if (season == SUMMER and
        moisture < DRY_SOIL_SUMMER and
        forecast == 'no rain expected'):
    return True
elif ((season == AUTUMN or season == SPRING) and
        moisture < DRY_SOIL_AUTUMN_SPRING):
    return True
return False
```

Explanation

Return Type & Question Specific

```
return ((season == SUMMER and
         moisture < DRY_SOIL_SUMMER and forecast == NO_RAIN) or
        ((season == AUTUMN or season == SPRING) and
         moisture < DRY_SOIL_AUTUMN_SPRING)
        )
```

Past Midterm

Complete the following function according to its docstring description. The function is designed to ask for a valid username (presumably for a user setting up a new account of some sort) until one is given, or the user reaches the maximum guesses allowed. You **must use a while-loop in your solution**. You may assume that **max_attempts** is at least 1.
Complete the function body. Do not write doctests or preconditions for the function.

```
def get_username(max_attempts):
    """
    Prompt the user for a valid username until a valid one is given or
    they have entered max_attempts invalid usernames.
    A valid username is one that is at least length 4 and does not contain
    any spaces. Each time the user gives an invalid username, tell them that
    it was invalid and prompt again if they have not reached max_attempts
    tries. If the user gives max_attempts invalid usernames, return the empty
    string; otherwise, return the valid username they gave.
    """
```

Explanation

Use a while loop, initiate variables, set return

```
attempts = 0
username = ""
while attempts < max_attempts:
    #Function Logic
return username
```

Explanation

Build function logic according to question

```
attempts = 0
username = ""
while attempts < max_attempts:
    username = input("Enter your username: ")
return username
```

Explanation

Build function logic according to question

```
attempts = 0
username = ""
while attempts < max_attempts:
    username = input("Enter your username: ")
    if len(username) >= 4 and " " not in username:
        break
return username
```

Explanation

Build function logic according to question

```
attempts = 0
username = ""
while attempts < max_attempts:
    username = input("Enter your username: ")
    if len(username) >= 4 and " " not in username:
        break
    username = ""
    print("Invalid username!")
    attempts += 1
return username
```

Solution

```
def get_username(max_attempts):
    """
    Prompt the user for a valid username until a valid one is given or
    they have entered max_attempts invalid usernames.
    A valid username is one that is at least length 4 and does not contain
    any spaces. Each time the user gives an invalid username, tell them that
    it was invalid and prompt again if they have not reached max_attempts
    tries. If the user gives max_attempts invalid usernames, return the empty
    string; otherwise, return the valid username they gave.

    """

    attempts = 0
    while attempts < max_attempts:
        username = input("Enter your username: ")
        if len(username) >= 4 and " " not in username:
            return username
        else:
            print("Invalid username.")
            attempts += 1
    return ""
```

Feedback Form!

Please fill out the feedback form below:



Acknowledgments

We would like to thank Professor Purva R. Gawde for providing us lecture notes to use as a basis for the review seminar.