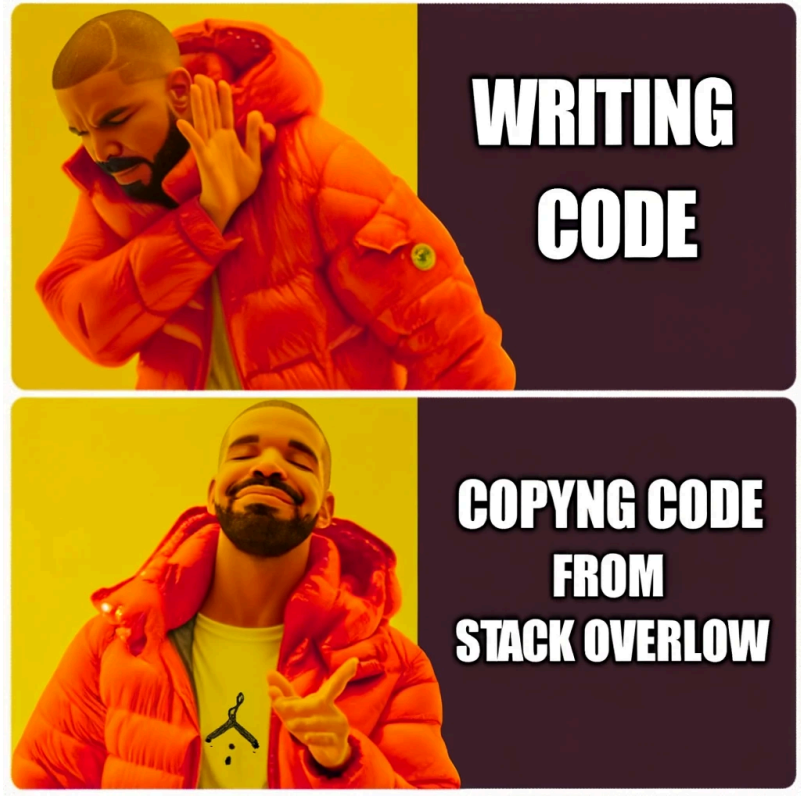
Discussion 2

DSC 20, Spring 2024

Please come up and take a worksheet (if you have an ipad, then the worksheet is already posted in the discussion tab of the calendar). Take 15 minutes to complete the questions to the best of your ability, we will go over answers afterwards.

Meme of the week



Courtesy of DALL-E

Agenda

- imports
- escape character / following style
- **Content**
    - loops
    - Mutability
    - lists, tuples, sets
    - in-place vs not
    - Indexing
    - List Methods
    - String Methods
    - Dictionaries

About Imports

"import" is another special keyword in python that has a unique function - it makes code from one body available in another. This can be something like a function from a different file or a whole package (ex. Pandas)

- We "ban" import in this class because many packages are too powerful (and out of the scope of this course)
- When an import is necessary, we will explicitly import the package/module for you in the starter

Packages are powerful tools that you'll be using nonstop after this class, but for now they are a whole different beast that you will have to deal with later.

In [34]:
```
import numpy as np
def find_ordinal_winner_np(votes, names):
    scores = np.array(votes).sum(axis=0)
    return names[np.where(scores==min(scores))[0][0]],names[np.where(sc

votes = [[2,1,3,4],[3,2,1,4],[2,1,4,3]]
names = ['tayven','gwen','linh','brandon']
winner, loser = find_ordinal_winner_np(votes,names)
print(winner)
print(loser)
```

```
gwen
brandon
```

Escape Character / Following Style

- \ is the escape "operator"
- bypasses the next character for some functionality

In [13]:
```
# instead of n being printed, using \ changed it to "newline character'
print('a \nb')
```

```
a
b
```

If I have a line of code that is way too long, I can use the escape character to retain functionality while reducing making it more readable

In [28]:
```
import math
x,y,z = 1,2,3
round(abs(((x ** 2) + (2 * y) - (3 * z)) / (math.sqrt(x) + math.exp(y)
```

Out[28]:
```
0.12
```

In [29]:
```
round(abs(((x ** 2) + (2 * y) - (3 * z)) / \
          (math.sqrt(x) + math.exp(y) * math.log(z)) * \
          (math.sin(x) + math.cos(y) + math.tan(z))), 2)
```

Out[29]:
```
0.12
```

doctest output lengths

In [1]:
```
def some_function(x):
    """
    >>> some_function('marina langlois')
    'marina langloismarina langloismarina langloismarina langloismarina
    """
    return x*5
```

In [ ]:
```
def some_function(x):
    """
    >>> some_function('marina langlois')
    'marina langloismarina langloismarina \
    langloismarina langloismarina langlois'
    """
    return x*5
```

In [ ]:
```
# The correct one!
def some_function(x):
    """
    >>> some_function('marina langlois')
    'marina langloismarina langloismarina \
langloismarina langloismarina langlois'
    """
    return x*5
```

Content

Loops

Loops are used to **repeat computations** many times.

- Two types of loops:
    - While loop: Uses logical conditions, useful for when the number of iterations is unknown (as long as a condition is true, code will run).
    - For loop : Uses an iterable object (ex. list), usually for when the number repetition is known.

In [46]:
```
nums = [1,2,3,4]
i=0
while i < len(nums):
    nums[i] = nums[i]*2
    i+=1
nums
```

Out[46]:
```
[2, 4, 6, 8]
```

In [48]:
```
nums = [1,2,3,4]
for i in range(len(nums)):
    nums[i]*=2
nums
```

Out[48]:
```
[2, 4, 6, 8]
```

Lists, Tuples, Sets

List

- Mutable vector of values
- Can store any data type, multiple types at a time
- Elements are acccessed via indexing

Tuple

- **Immutable** vector of values
- all else equal to list

Set

- Mutable vector of values
- Only stores unique elements (removes duplicates)
- otherwise, list like behavior

In [3]:
```
to_5_list = [1,2,3,4,5]
to_5_tup = (1,2,3,4,5)
to_5_set = {1,2,3,4,5}
```

Mutability

- Object is mutable if it can be directly changed after it is created
- If it can't, it is immutable
- Lists are **mutable**
- strings, tuples, and numbers are **immutable**

In [1]:
```
test_str = 'DSC20'
test_str[-2] = '3'
test_str
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent
call last)
Cell In[1], line 2
      1 test_str = 'DSC20'
----> 2 test_str[-2] = '3'
      3 test_str

TypeError: 'str' object does not support item assignment
```

In [2]:
```
test_lst = ['D','S','C','2','0']
test_lst[-2] = '3'
test_lst
```

Out[2]:
```
['D', 'S', 'C', '3', '0']
```

Indexing/Slicing

Indexing/slicing refers to accessing specific element(s) from an iterable object. Two of the most common cases for this are lists and strings. Indexing results in a copy (unless reasssigned)!

- iterable[start:stop:skip] (start:inclusive, stop: NOT inclusive)
- not every section needs to be specified (can just use start or stop or skip)
- sub indexes can be applied (ex. lst[0][0] -> takes the first element of the first element)
- Trying to access an index that doesn't exist in the list will result in an error

In [49]:
```
lst = list(range(2,13))
print("original list: " + str(lst))
```

```
original list: [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

In [50]:
```
print("reversed list: " + str(lst[::-1]))
```

```
reversed list: [12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2]
```

In [51]:
```
print("the 2nd to 4th element: " + str(lst[2:4]))
```

```
the 2nd to 4th element: [4, 5]
```

In [52]:
```
print("every third element from the 1st to 10th element: " + str(lst[1:
```

```
every third element from the 1st to 10th element: [3, 6, 9]
```

In [53]:
```
print("If I try to slice outside of range: " + str(lst[-100:100]))
```

```
If I try to slice outside of range: [2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12]
```

in-place operations

Before we start exploring functions, it's important to understand what in-place operations are.

Definition: an operation is in-place if the result occurs directly on the original object, rather than a copy. Many in-place functions return None for an output.

What does this actually mean? The result of a not in-place function is a copy, the original object is not modified and the result has to be assigned to a variable to be retained. In-place functions modify the actual object passed in.

In [54]:
```
lst = [1,2,3,4,5]
print('This result is temporary - unless I reassign \
lst to it, lst is not modified: ' + str(lst + [6]))
print("lst's current state: " + str(lst))
lst_new = lst + [6] # reassigning the result retains the output
print("the reassigned lst -> lst_new=" + str(lst_new))
```

```
This result is temporary - unless I reassign lst to it, lst is n
ot modified: [1, 2, 3, 4, 5, 6]
lst's current state: [1, 2, 3, 4, 5]
the reassigned lst -> lst_new=[1, 2, 3, 4, 5, 6]
```

In [55]:
```
lst = [1,2,3,4,5] # compare that to .append
print("lst's current state: " + str(lst))
print("The result of .append() is: " + str(lst.append(6))) # .append()
print("but we can see that the original lst is modified: " + str(lst))
```

```
lst's current state: [1, 2, 3, 4, 5]
The result of .append() is: None
but we can see that the original lst is modified: [1, 2, 3, 4,
5, 6]
```

List Methods

- list.pop(index) -> removes the element at index. in-place operation; returns the value removed
- list.append(element) -> adds the element to the end of the list. in-place operation
- list.sort() -> sorts the list (as name implies). Can specify ascending or descending

In [13]:
```
lst = [5,4,3,2,1,0,0]
print("result of pop first element: " + str(lst.pop(0)))
lst.append(19)
print("after appending 19: "+ str(lst))
lst.sort()
print("after sorting list: " + str(lst))
```

```
result of pop first element: 5
after appending 19: [4, 3, 2, 1, 0, 0, 19]
after sorting list: [0, 0, 1, 2, 3, 4, 19]
```

Checkpoint

Assume the following code has been ran:

In [5]:
```
lst = [10, [12,13,11,10], (2,3,10,4), [4,6, 10, [10,11]], 1]
```

Which statement will extract 10 from the list? Select any that apply.

A. lst[-1]

B. lst[1][-1]

C. lst[2][3]

D. lst[-2][-1][0]

Checkpoint Solution

In [6]:
```
print(lst[-1])
print(lst[1][-1])
print(lst[2][3])
print(lst[-2][-1][0])
```

```
1
10
4
10
```

String Methods

- string.split(separator) -> splits a string into a list of elements on each separator
- "[string]".join(iterable) -> returns the iterable "joined" with the [string] in between each element
- string.lower()/string.upper() -> lowers/uppers all elements in a string
- string.strip() -> removes whitespace from beginning and end of the string

In [14]:
```
string = "Professor Marina Langlois"
print(string.split())
print('-'.join(string.split()))
print(string.upper())
```

```
['Professor', 'Marina', 'Langlois']
Professor-Marina-Langlois
PROFESSOR MARINA LANGLOIS
```

Checkpoint

Assume the following code has been ran:

In [4]:
```
statement = "Marina Langlois is the best DSC20 professor ever."
temp = ' '.join(statement.split())
```

Is the following statement True or False? temp is equal to statement

A. True

B. False

Checkpoint Solution

In [5]:
```
temp
```

Out[5]:
```
'Marina Langlois is the best DSC20 professor ever.'
```

In [6]:
```
statement
```

Out[6]:
```
'Marina Langlois is the best DSC20 professor ever.'
```

In [7]:
```
temp == statement
```

Out[7]:
```
True
```

Dictionaries

- Mutable storage of key, value pairs
- Can store any data type, multiple at a time
- Elements are acccessed via keys
- keys must be **hashable** and **unique**

methods

- accessing keys (as a list) -> dict.keys()
- accessing values (as a list) -> dict.values()
- accessing key,value pairs (as a list of tuples) -> dict.items()

note: hashablility correlates to the stability of the data - essentially, **data that can't change is hashable** (int, str, tuple, etc.) while **data that can change is not hashable (list, dictionary). Basically, it's all about mutability!**

In [6]:
```
temp = {}
```

In [7]:
```
temp['marina langlois'] = 1
```

In [5]:
```
hash('marina langlois')
```

Out[5]:
```
2102949250031902339
```

In [6]:
```
temp[[1,2,3]] = 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent
call last)
Cell In[6], line 1
----> 1 temp[[1,2,3]] = 2

TypeError: unhashable type: 'list'
```

In [2]:
```
hash([1,2,3])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent
call last)
Cell In[2], line 1
----> 1 hash([1,2,3])
```

```
TypeError: unhashable type: 'list'
```

In [16]:
```
list(temp.keys())[0]
```

Out[16]:
```
'marina langlois'
```

Checkpoint

Assume the following code has been ran:

In [8]:
```
dct = {"nikki":"nikki", ('a','b','c'):[1,2,3], 'max':'nikki'}
```

Which of the following statements will extract "nikki" from the dictionary? Select any that apply.

A. dct.items()[0][1]

B. dct.keys[0]

C. list(dct.values())[-1]

D. dct['max']

Checkpoint Solution

In [9]:
```
print(list(dct.values())[-1])
print(dct['max'])
```

```
nikki
nikki
```

In [10]:
```
dct.items()[0][1]
```

```
---------------------------------------------------------------------
-----------
TypeError                                 Traceback (most recent
call last)
Cell In[10], line 1
----> 1 dct.items()[0][1]

TypeError: 'dict_items' object is not subscriptable
```

In [11]:
```
dct.keys()[0]
```

```
---------------------------------------------------------------------
-----------
TypeError                                 Traceback (most recent
call last)
Cell In[11], line 1
----> 1 dct.keys()[0]
```

```
TypeError: 'dict_keys' object is not subscriptable
```

In [15]:
```
dct.keys()
```

Out[15]:
```
dict_keys(['nikki', ('a', 'b', 'c'), 'max'])
```

Thanks for coming!