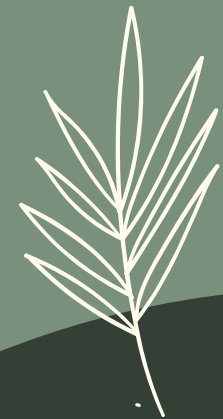


Exam 2 Review Session

Strings



Strings

Understanding Python Strings:

In Python, strings are sequences of characters. Think of them as a line of letters, numbers, or symbols connected in order. You can create a string by enclosing characters in quotes. For example, "Hello, World!" is a string.

String Indexing:

Every character in a string has a unique position called an index. In Python, indexing starts at 0. So, in the string "Python", the character P is at index 0, y is at index 1, and so on. You can use these indexes to access individual characters (**string[index]**)

| | | | | | |
|----|----|----|----|----|----|
| -6 | -5 | -4 | -3 | -2 | -1 |
| f | o | o | b | a | r |
| 0 | 1 | 2 | 3 | 4 | 5 |

Substrings

Slicing allows you to obtain a portion of the string, known as a substring. Use the syntax **string[start:stop:step]** to get characters from the "start" index up to but not including the "stop" index. For example, "Python"[1:4:1] results in "yth"

Functions

- * "hi" in my_string - contains
- * str.lower()
- * str.upper()
- * str.startswith("hello")
- * str.endswith("world")
- * Note: return new value

- * str.replace(str1, str2)
- * str.strip()

- str.count(str1) - returns total # of occurrences
- str.find(str1) - returns lowest index

Strings Questions

Practice Question 1:

Given the string `s = "Programming in Python"`, what will `print(s[3:14:2])` output?

- A) "gamgni"
- B) "rgamg y"
- C) "rgm ni"
- D) "Programming"

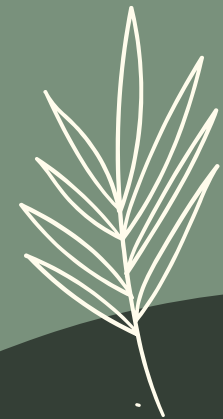
Practice Question 2:

Consider the string `phrase = "Data Science"`. What does `print(phrase[::-1])` display?

- A) "ecneicS ataD"
- B) "Data Science"
- C) An error message
- D) "ecneicS ataD "

Answer: A) "ecneicS ataD"

Lists





Lists

Understanding Python Lists:

Lists are versatile and can hold a mixed collection of items (strings, numbers, or even other lists). They are created with square brackets, like ['apple', 'banana', 'cherry']. Lists are mutable, allowing modification

Modifying Lists:

You can add, remove, or change items in a list. Use methods like `.append(item)` to add, `.remove(item)` to remove, and `list[index] = new_value` to change an item.

List Operations

Lists support operations like concatenation (+), repetition (*), and membership testing (in). For example, `['a', 'b'] + ['c', 'd']` results in `['a', 'b', 'c', 'd']`.

- Adding:
 - `my_list.append('watermelon')`
 - `my_list.insert(1, 'orange')` - insert the item at the specified index
- Remove:
 - `my_list.remove('apple')`
- Index
 - `my_list.index('banana')` - gives you the index location
- Other
 - `my_list.sort()`

List Questions

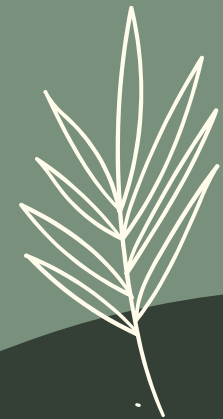
Consider the list `numbers = [1][2][3][4]`. After executing `numbers.insert(3, 'four')`, what does `numbers` look like?

- A) `[1, 2, 3, 'four', 4, 5]`
- B) `[1, 2, 'four', 3, 4, 5]`
- C) `[1, 2, 3, 4, 'four', 5]`
- D) An error message

Given `fruits = ["apple", "banana", "cherry"]`, what will `fruits.append(["dragonfruit", "elderberry"])` result in?

- A) `["apple", "banana", "cherry", ["dragonfruit", "elderberry"]]`
- B) `["apple", "banana", "cherry", "dragonfruit", "elderberry"]`
- C) An error message
- D) `["apple", "banana", "cherry", "dragonfruit"]`

Tuples



Tuples

Understanding Python Tuples:

Tuples are ordered collections of items, similar to lists. However, unlike lists, tuples are immutable, meaning once created, their content cannot be changed. Create a tuple by enclosing items in parentheses, like (1, 2, 3)

Accessing Tuples:

Access elements in a tuple using their index, just like strings. For instance, in the tuple ('a', 'b', 'c') the element 'a' has an index of 0. Use **tuple[index]** to access a specific element.

Immutability

Once a tuple is created, its elements cannot be altered. This characteristic makes tuples a secure choice for storing data that should not change throughout the execution of your program.

```
my_tuple = ("S", "R", "I", "S", "H", "T", "I")
```

Sciencetech Easy

Index number



Negative Index number

- Index
 - `my_tuple.index('Yellow')` - gives you the index location
- Other
 - `my_tuple.count(22)` - how many times that item appears

Tuples Questions

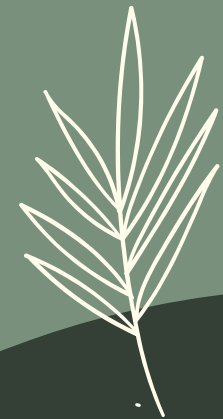
Given the tuple `t = (10, 20, 30, 40, 50)`, what is the result of `t[1:4]`?

- A) `(20, 30, 40)`
- B) `(10, 20, 30)`
- C) `(20, 30, 40, 50)`
- D) An error message

If `colors = ("red", "green", "blue")`, which of the following operations is invalid?

- A) `print(colors[2])`
- B) `colors += ("yellow",)`
- C) `colors[1] = "purple"`
- D) `print(colors.index("green"))`

Dictionaries





Dictionaries

Understanding Python Dictionaries:

Dictionaries in Python are collections of key-value pairs. They allow you to store and retrieve data by a unique key. Unlike lists, dictionaries are unordered, which means the data can be accessed by the key rather than the position. Dictionaries are created using curly braces {} with pairs separated by commas, and key-value pairs are connected by colons. For example, `my_dict = {'name': 'John', 'age': 30}`

Modifying Dictionaries:

Dictionaries are mutable, meaning you can add new key-value pairs or change the value of existing keys. To add a new entry, use `dictionary[new_key] = new_value`. To update an existing entry, use the same syntax with an existing key. For example, `my_dict['age'] = 31` updates John's age

Methods:

Python dictionaries come with several useful methods for manipulation and access. Some of these include `.keys()` to get a list of all keys, `.values()` for all values, and `.items()` for all key-value pairs. The `.pop(key)` method can remove an item by key.



Dictionaries

d[key]

Return the **value** of *d* with key *key*. Raises a **KeyError** if *key* is not in the dictionary.

len(d)

Return the **number of items** in the dictionary.

key in d

Return **True** if *d* has a key *key*, else **False**.

list(d)

Return a **list of all the keys** in the dictionary.

del d[key]

Remove *d[key]* from *d*. Raises a **KeyError** if *key* is not in the dictionary.

d.popitem()

Remove and return a (**key**, **value**) pair from the dictionary. Pairs are returned in LIFO (most recent) order.

d.pop(key[, default]) # *default is optional*

If *key* is in the dictionary, **remove it and return its value**, else return *default*. If *default* is not given and *key* is not in the dictionary, a **KeyError** is raised.

d.get(key[, default]) # *default is optional*

Return the value for *key* if *key* is in the dictionary, else *default*. If *default* is not given, it defaults to **None**, so that this method never raises a **KeyError**.





Dictionaries Questions



Given the dictionary `d = {'key1': 10, 'key2': 20}`, what code would you use to add a new key-value pair `'key3': 30` and then retrieve the value for `'key2'`?

Consider the dictionary `grades = {'Alice': 88, 'Bob': 76, 'Charlie': 90}`. Write a Python expression that removes Bob's grade and then prints the remaining dictionary.







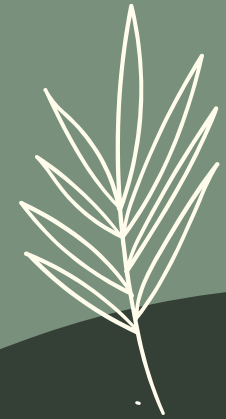
Dictionaries Answers

```
d['key3'] = 30  
print(d['key2'])
```

```
grades.pop('Bob')  
print(grades)
```



Nested Data Structures





Nested Data Structures

Understanding Nested Structures:

Nested data structures refer to collections that contain other collections, such as lists of lists, dictionaries within dictionaries, or any combination thereof. They are useful for representing complex data relationships.

Navigating:

To access elements within nested structures, chain the indices or keys according to the level of nesting. For a list of lists, `my_list[1]` accesses the second element of the first list. For a dictionary containing lists, `my_dict['key']` accesses the first element of the list associated with 'key'.

Modifying:

You can modify elements in nested structures by accessing the target element and assigning a new value. For example, to change an element in a nested list, use `my_list[1][2] = new_value`. For nested dictionaries, `my_dict['outer_key']['inner_key'] = new_value` changes the value associated with 'inner_key' inside an outer dictionary.



Looping through structures:

List:

Loop through a list by the items -

```
for my_item in my_list:  
    Do stuff with my_item
```

Loop through a list by index -

```
for i in range(len(my_list)):  
    my_item = my_list[i]  
    Do stuff with my_item
```

Dict:

Loop through a dictionary -

```
for my_key in my_dictionary:  
    Do stuff with my_key  
    Do stuff with my_dictionary[my_key]
```

Can also loop through -

- d.keys()
- d.values()
- d.items()

```
for key, val in my_dictionary.items():  
    # do stuff with key  
    # do stuff with val  
    ...
```





Nested Structure Questions



Given the nested list `nested_list = [[1][2][3], [4][5][6], [7][8]]`, how would you access the number 5?

Consider a nested dictionary `nested_dict = {'first': {'a': 1}, 'second': {'b': 2}}`. Write a Python expression to add a new key-value pair `'c': 3` to the dictionary associated with the key `'second'`.







Nested Data Structures Answers

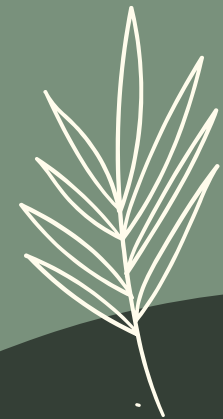


```
print(nested_list[1][1])
```

```
nested_dict['second']['c'] = 3  
print(nested_dict)
```



Regex





Regular Expressions

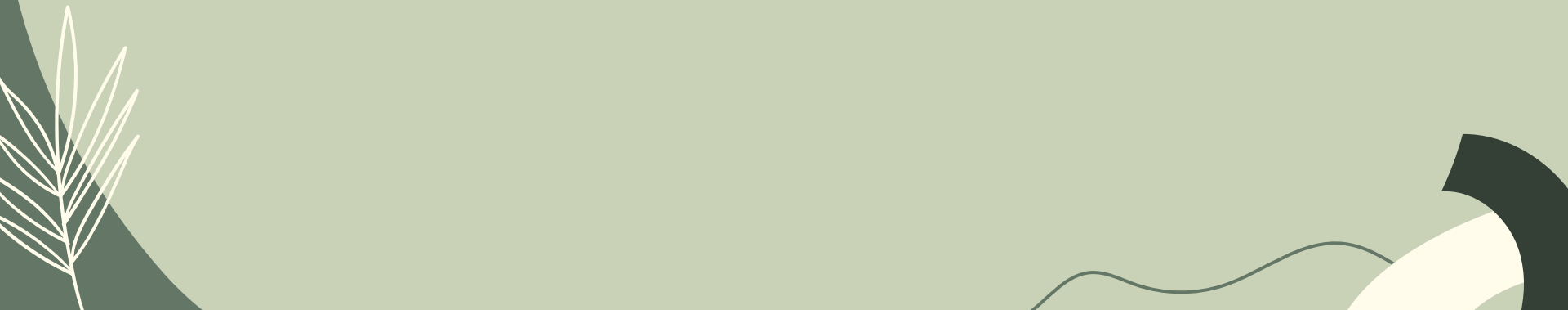
The Basics:

Regular expressions, or RegEx, are a powerful tool for pattern matching and text processing. They allow you to define a search pattern to locate specific sequences of characters within strings. RegEx is widely used for searching and manipulating text, such as in coding, log file analysis, and data validation.

Tips for RegEx:

Practice! The more you practice the more familiar you will be with the instructions:

<https://regex.sketchengine.eu/index.html>



- `[]` = brackets, matches a single character contained within
- `[x-z]` = hyphen, denotes a range of characters
- `a.b` = dot, matches any single character in that location
- `[^abc]` = caret, matches a single character that is not a, b, or c
- `c?` = preceding character (c) 0 or 1 times
- `c+` = preceding character (c) 1 or MORE times
- `c*` = preceding character (c) 0 or MORE times
- `a|b` = vertical bar, means or

- `^` = matches the **beginning** of a string
- `$` = matches the **end** of a string
- `{m}` = exactly **m copies** of the previous RE
- `{m,n}` = from **m to n copies** of the previous RE
- `\b` = word **boundary** matching RE at beginning or end of a **word**
- `\d` = matches any **decimal digit**
- `\D` = matches any character that is **not a decimal digit**
- `\s` = matches any **whitespace** character
- `\S` = matches any character that is **not a whitespace** character
- `\w` = matches any **word** character
- `\W` = matches any character that is **not a word** character





RegEx Questions

Construct a regex pattern that matches any string starting with 'b' and ending with 'at', such as "bat", "boat", or "brat".


Create a regex pattern that matches any string containing only lowercase letters and the underscore character.

Write a regex pattern that matches strings that have exactly three digits followed by any number of alphabetic characters.





Final Tips:

- ❑ Look over old quiz questions, especially if you lost points.
 - ❑ Time yourself, a lot of students struggle with pacing. Aim for two and a half minutes max per question
 - ❑ Practice!
 - ❑ Past exams + practice worksheet: [Exam 2 | CS 1112 S24 \(quole0812.github.io\)](https://quole0812.github.io)
 - ❑ Regex Practice: <https://regex.sketchengine.eu/index.html>
 - ❑ <https://pynative.com/python-exercises-with-solutions/>
 - ❑ Feel free to ask conceptual question on piazza!
- 
- 