AI Boot Camp

Complex Programming Decisions

Module 2 Day 3

Class Objectives

By the end of class, you will be able to:

1. Refactor an existing `if-elif-else` statement into a match case structure.
2. Create and manipulate Python dictionaries and effectively iterate through them using `items()`, `keys()`, and `values()`.
3. Demonstrate the ability to iterate through complex nested data structures, such as lists within lists and lists of dictionaries.
4. Extract specific information from nested dictionaries.
5. Use list comprehension and list operations to process data and perform calculations.

Activity:

Rock Paper Scissors Warm Up

Write a simple game of Rock, Paper, Scissors to refresh your memory on if-else statements, while loops, and membership, logical, and comparison operators.

Suggested Time:

15 Minutes

Instructor **Demonstration**

Match Case

Match case

1. They are a kind of conditional that checks which of a list of particular cases has been met.

2. Alternative to some `if-elif-else` statements.

   Reduces the complexity of the statement—e.g., rather than having to write the same variable multiple times in each if/elif line, you can use it in a single match line and then include a case line for each value you want to match.

3. Of course it is possible that the input isn't accounted for in any of the specified cases. Here, underscore is used as a "catchall" final line if you want the statement to do something when no matches are found.

4. The match statement is completed and exited upon the first match. If the `break` keyword is used inside any of the cases, that will break from a loop when the match statement is inside a loop.

Activity:
Refactor Rock Paper Scissors

Refactor the if-elif-else statements from the Rock, Paper, Scissors code into match statements.

Suggested Time:
10 Minutes

Time’s up!

Let’s review

Questions?

Instructor **Demonstration**

Dictionaries

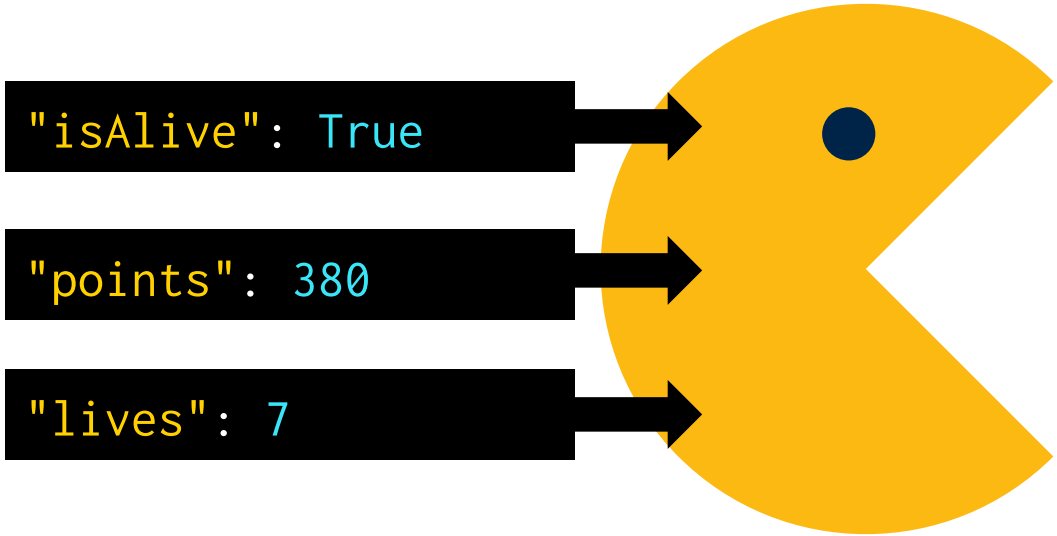
Another commonly used data type in Python is **the dictionary or `dict`.**

A **dictionary** is an object that stores a collection of data.

Dictionaries

Like lists and tuples, dictionaries can contain multiple values and data types. However, unlike lists and tuples, dictionaries store data in key-value pairs.

The key in a dictionary is a string that can be referenced to collect an associated value.

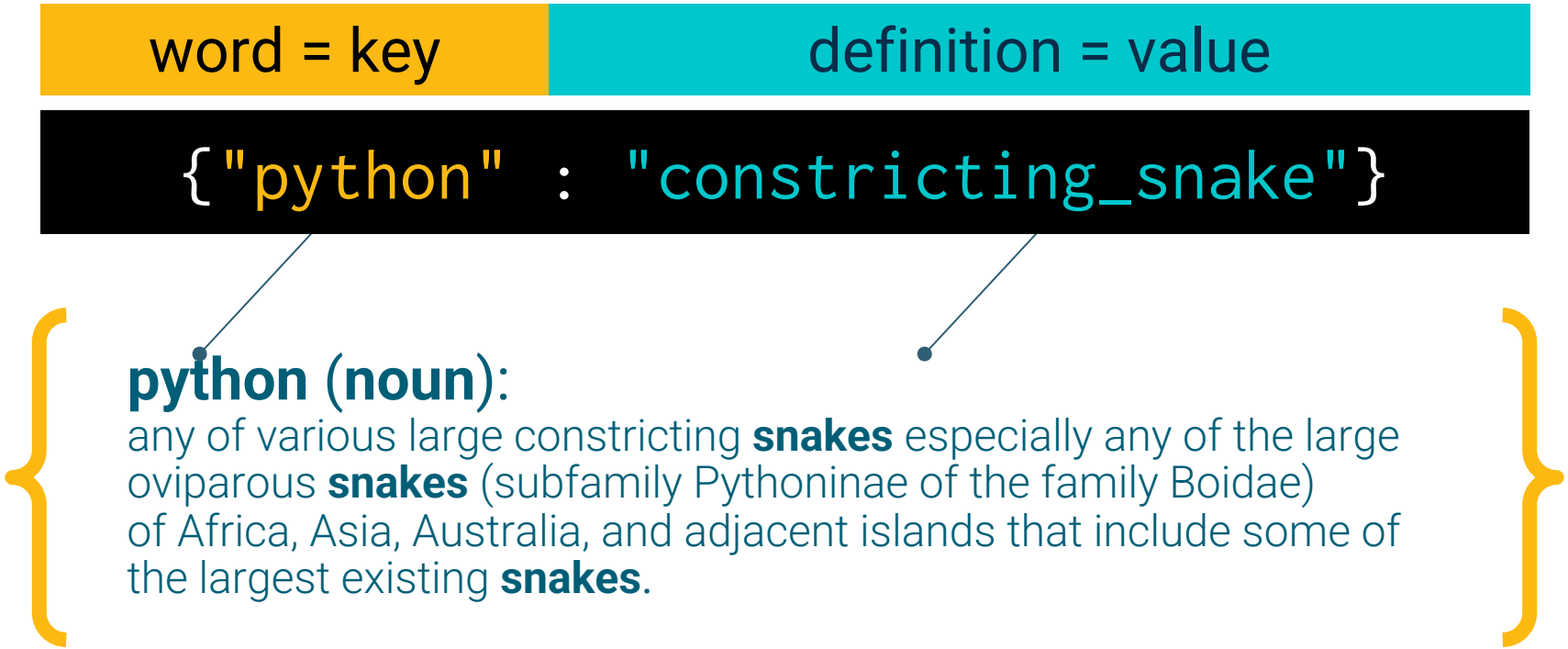



```
pacman = {
    "isAlive": True,
    "points": 380,
    "lives": 7

}
```

Dictionaries

To use the example of a physical dictionary, the words in the dictionary would be considered the keys, and the definitions of those words would be the values.



Dictionaries

Keys

Keys are immutable objects, like integers, floating-point decimals, or strings.

Keys cannot be lists or any other type of mutable object.

Values

Values in a dictionary, as captured in the following image, can be objects of any type:

- Integers
- Floating-point decimals
- Strings, Booleans
- `datetime` values
- Lists

Dictionaries

To initialize or create an empty dictionary, we use the syntax `actors = {}`.

```
# Create a dictionary to hold the actor's names.
actors = {}
```

You can also create a dictionary with the built-in Python `dict()` function, or `actors = dict()`.

```
# Create a dictionary using the built-in function.
actors = dict()
```

Dictionaries

Items can be added to dictionaries at declaration by creating a key, following it with a colon, and then placing the desired value after the colon.

To reference a value within a dictionary, we simply call the dictionary and follow it up with a pair of brackets containing the key for the desired value.

```
# A dictionary of an actor.
actors = {"name": "Tom Cruise"}
print(f'{actors["name"]}')
```

Dictionaries

Values can also be added to dictionaries by placing the key within single or double quotation marks inside brackets, and then assigning the key a value; then, values can be changed or overwritten by assigning the key a new value.

```
# Add an actor to the dictionary with the key "name"
# and the value "Denzel Washington".
actors["name"] = "Denzel Washington"
```

Dictionaries

Dictionaries can hold multiple pieces of information by following up each key-value pairing with a comma and then another key-value pair.

```
# A list of actors
actors_list = [
    "Tom Cruise",
    "Angelina Jolie",
    "Kristen Stewart",
    "Denzel Washington"]

# Overwrite the value, "Tom Cruise", with the list of actors.
actors["name"] = actors_list
```

Dictionaries

Items in a list in a dictionary can be accessed by calling the key and then using indexing to access the item, as in the following image.

```
# Print the first actor
print(f'{actors["name"][0]}')
```

You only need a basic understanding of this for now; when you get into APIs, you will get a lot more practice!

Dictionaries can also contain **other dictionaries.**

Dictionaries

To access the values inside nested dictionaries, simply add another key to the reference.

```
# A dictionary can contain multiple pairs of information
actress = {"name": "Angelina Jolie", "genre": "Action", "nationality": "United States"}

#----------------------------------------------------------------

# A dictionary can contain multiple types of information
another_actor = {"name": "Sylvester Stallone", "age": 62, "married": True,"best movies"
print(f'{another_actor["name"]} was in {another_actor["best movies"][0]}

#----------------------------------------------------------------

# A dictionary can even contain another dictionary
film = {"title": "Interstellar",
        "revenues": {"United States": 360, "China": 250, "United Kingdom": 73}}
print(f'{film["title"]} made {film["revenues"]["United States"]}'" in the US.")
#----------------------------------------------------------------
```

Dictionary functions

1. `keys()` returns a list of the key names.
2. `values()` returns a list of the values in whatever data type or data structure they're stored as.
3. `items()` returns a list of tuples in the format [(key, value)].

Activity:
Understanding Dictionaries

Practice creating dictionaries and accessing information from them, both directly and iterating through them.

Suggested Time:
10 Minutes

Time’s up!

Let’s review

Questions?

Instructor **Demonstration**

Iterating Nested Data Structures

Nested data structures

Just like conditionals within conditionals and loops within loops, you can nest data structures into one another, whether that be a list of lists, a tuple of tuples, a dictionary of dictionaries, or any combination you can think of.

List of lists:

```
[
      ["James", "Noor", "Ayoka"],
      [65, 72, 80],
      ["Washington", "Kuala Lumpur", "Abuja"]
]
```

List of dictionaries:

```
[
      {
            "name": "James",
            "score": 65,
            "city": "Washington"
      },
      {
            "name": "Noor",
            "score": 72,
            "city": "Kuala Lumpur"
      },
      {
            "name": "Ayoka",
            "score": 80,
            "city": "Abuja"
      }
]
```

Iterating through nested data structures

1. Iterating through a list of lists or any nested data structure is a simple matter of indexing or calling the desired element through the relevant storage layers.

2. If we want to call the second list from our previous list of lists, we call it as we would any other element:

   - `list_of_lists[1]`

3. If we want only the first element of the second list, we can just specify that as well.

   - `list_of_lists[1][0]`

4. The same principle applies to dictionaries and calling values using the appropriate keys.

Activity:
Exploring the Nest

Practice accessing information from nested data structures containing information about birds.

Suggested Time:
15 Minutes

Time’s up!

Let’s review

Break

15 mins

Instructor **Demonstration**

Nested Dictionaries

Activity:

Printing a Menu

Navigate a food truck menu stored in a Python dictionary to print out different sections based on user selection.

Suggested Time:

20 Minutes

Time’s up!

Let’s review

Instructor **Demonstration**

List Comprehension

List comprehensions

List comprehensions are a powerful method for populating lists.

Example:

```
for i in range(5)
        squares[i] = i*i
```

vs

```
squares = [i * i for i in range(5)]
```

Every list comprehension in Python includes three elements:

- The **expression** instructs how the values should be created. In the example, the expression is `i * i`.
- The **member** is the object or value in the list or iterable. In the example, the member value is `i`.
- The **iterable** is an object that can return its elements one at a time. In the example, the iterable is `range(5).`

Activity:

Guest List

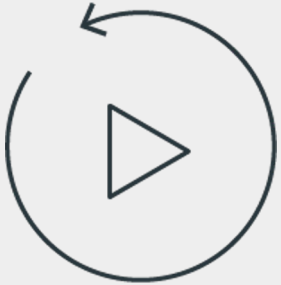
Extract data from dictionaries and use list comprehensions to determine the number of guests attending an event.

Suggested Time:

10 Minutes

Time’s up!

Let’s review



Let’s **recap**

Recap

Today you learned about:

1. Refactoring an existing `if-elif-else` statement into a match statement.
2. Creating and manipulating Python dictionaries and effectively iterating through them using `items()`, `keys()`, and `values()`.
3. Demonstrating the ability to iterate through complex nested data structures, such as lists within lists and lists of dictionaries.
4. Extracting specific information from nested dictionaries.
5. Using list comprehension and list operations to process data and perform calculations.

Challenge

Food Truck Order System

Questions?

The End