Computer Systems Modeling and Verification
(USEEN1)

Structural Pattern Matching (tutorial)

This tutorial is taken from the first part of PEP 636 – Structural Pattern Matching: Tutorial.

As an example to motivate this tutorial, you will be writing a text adventure. That is a form of interactive fiction where the user enters text commands to interact with a fictional world and receives text descriptions of what happens. Commands will be simplified forms of natural language like `getsword`, `attack dragon`, `go north`, `enter shop` or `buy cheese`.

Matching sequences

Your main loop will need to get input from the user and split it into words, let's say a list of strings like this:

```
command = input("What␣are␣you␣doing␣next?␣")
# analyze the result of command.split()
```

The next step is to interpret the words. Most of our commands will have two words: an action and an object. So you may be tempted to do the following:

```
[action, obj] = command.split()
... # interpret action, obj
```

The problem with that line of code is that it's missing something: what if the user types more or fewer than 2 words? To prevent this problem you can either check the length of the list of words, or capture the `ValueError` that the statement above would raise.

You can use a matching statement instead:

```
match command.split():
    case [action, obj]:
        ... # interpret action, obj
```

The match statement evaluates the **"subject"** (the value after the `match` keyword), and checks it against the **pattern** (the code next to `case`). A pattern is able to do two different things:

- Verify that the subject has certain structure. In your case, the `[action, obj]` pattern matches any sequence of exactly two elements. This is called **matching**
- It will bind some names in the pattern to component elements of your subject. In this case, if the list has two elements, it will bind `action = subject[0]` and `obj = subject[1]`.

If there's a match, the statements inside the case block will be executed with the bound variables. If there's no match, nothing happens and the statement after `match` is executed next.

Note that, in a similar way to unpacking assignments, you can use either parenthesis, brackets, or just comma separation as synonyms. So you could write `case action, obj` or `case (action, obj)` with the same meaning. All forms will match any sequence (for example lists or tuples).

Matching multiple patterns

Even if most commands have the action/object form, you might want to have user commands of different lengths. For example, you might want to add single verbs with no object like `look` or `quit`. A match statement can (and is likely to) have more than one `case`:

```
match command.split():
    case [action]:
        ... # interpret single-verb action
    case [action, obj]:
        ... # interpret action, obj
```

The match statement will check patterns from top to bottom. If the pattern doesn't match the subject, the next pattern will be tried. However, once the *first* matching pattern is found, the body of that case is executed, and all further cases are ignored. This is similar to the way that an `if/elif/elif/...` statement works.

Matching specific values

Your code still needs to look at the specific actions and conditionally execute different logic depending on the specific action (e.g., `quit`, `attack`, or `buy`). You could do that using a chain of `if/elif/elif/...`, or using a dictionary of functions, but here we'll leverage pattern matching to solve that task. Instead of a variable, you can use literal values in patterns (like `"quit"`, `42`, or `None`). This allows you to write:

```
match command.split():
    case ["quit"]:
        print("Goodbye!")
        quit_game()
    case ["look"]:
        current_room.describe()
    case ["get", obj]:
        character.get(obj, current_room)
    case ["go", direction]:
        current_room = current_room.neighbor(direction)
    # The rest of your commands go here
```

A pattern like `["get", obj]` will match only 2-element sequences that have a first element equal to `"get"`. It will also bind `obj = subject[1]`.

As you can see in the `go` case, we also can use different variable names in different patterns.

Literal values are compared with the `==` operator except for the constants `True`, `False` and `None` which are compared with the `is` operator.

Matching multiple values

A player may be able to drop multiple items by using a series of commands`drop key`, `drop sword`, `drop cheese`. This interface might be cumbersome, and you might like to allow dropping multiple items in a single command, like `drop key sword cheese`. In this case you don't know beforehand how many words will be in the command, but you can use extended unpacking in patterns in the same way that they are allowed in assignments:

```
match command.split():
    case ["drop", *objects]:
        for obj in objects:
            character.drop(obj, current_room)
    # The rest of your commands go here
```

This will match any sequences having "drop" as its first elements. All remaining elements will be captured in a `list` object which will be bound to the `objects` variable.

This syntax has similar restrictions as sequence unpacking: you can not have more than one starred name in a pattern.

Adding a wildcard

You may want to print an error message saying that the command wasn't recognized when all the patterns fail. You could use the feature we just learned and write `case [*ignored_words]` as your last pattern. There's however a much simpler way:

```
match command.split():
    case ["quit"]: ... # Code omitted for brevity
    case ["go", direction]: ...
    case ["drop", *objects]: ...
    ... # Other cases
    case _:
        print(f"Sorry, I couldn't understand {command!r}")
```

This special pattern which is written `_` (and called wildcard) always matches but it doesn't bind any variables.

Note that this will match any object, not just sequences. As such, it only makes sense to have it by itself as the last pattern (to prevent errors, Python will stop you from using it before).