Text-based-Python-adventures

This is a step-to-step notebook to make **text-based-Python-adventures**.

And! To work the following **programming concepts**:

- reading `input` from the user at the terminal
- organizing data in structures like `lists` and `dictionaries`
- using `if/else statements` to check different conditions
- encapsulating code with `functions`
- using `while` loops

We will use **Python scripts** and **run the code in the terminal**.

This notebook is more like a reference, you can run code here but it's recommended to work in a Python script.

We will start with a more **linear approach**. After that, we switch to a **non-linear approach**.

Using input()

```
In [ ]: input()
```

```
In [ ]: reply = input()
```

```
In [ ]: print(reply)
```

Linear approach

Copy the code from below and save it as a Python script.

You can give it any name, just make sure the file ends with `.py`.
For example: `adventures.py`

Note: choose where you prefer to work (on `breadcube` in Jupyter, or on your own computer)

In …
```
welcome_message = """
    You came back from the winter break and you
walk into the XPUB studio.
    Oh... Nobody is here apart from a bunch of
silent Mac computers...

    Before the break, Michael mentioned something
about renovations...
    You can't remember what it was again, but he
said something about
    the FIT people, and that they would install an
ultra-smart-sensor-thingy in the studio...
    and that it would respond to any XPUB student
saying... 'hello'...

    Would this be true...?

    Is it installed already...?
    """

print(welcome_message)
reply = input()

if "hello" in reply:
    print("hello ... hello ... ello ... lo ...
o ...")
else:
    print("This cave is special... it only echoes
the word 'hello'.")

message = """
    Suddenly the lights go off!

    What happened? Is there no electricity?

    You look around to find the exit door, but it's
very dark.
    You have something in your right hand... is it
a lamp?
    """
```

```python
print(message)
reply = input()

if "get lamp" in reply:
    print("You turn on the light, and wow! Did you see that?")
else:
    print("The cave is still damp... what is this thing in your right hand?")
```

Open a terminal, navigate to the folder where you saved the Python file, and run the script with:

`$ python3 adventures.py` or `python adventures.py` or `py adventures.py` (depending on your computer)

Try to play the game first, and have a look at the code afterwards.

Good practice is to annotate the code step-by-step, to see if you can follow what is happening.

Non-linear approach

Make another script and call it something else, for example `xpub.py`.

Copy the code from below and run the script to see what it does.

```
In [ ]: welcome_message = "You are in the XPUB studio."

        print(welcome_message)

        while True:
            reply = input()
```

Not a single reply (using if/else)

How to handle multiple possible replies? For this we can use **if/ else** statements.

We can use `if`, `elif` and `else` to respond to multiple possible replies:

```
message = "What do you want to do?"
print(message)
reply = input()

if "look" in reply:
    print("Return what happens when you 'look'.")
elif "bag" in reply:
    print("Return what happens when you open your bag.")
elif "get" in reply:
    print("Return what happens when you 'go'.")
elif "read" in reply:
    print("Return what happens when you 'read'.")
elif "exit" in reply:
    print("Return what happens when you want to exit the game.")
else:
    print("Return what happens when the action is not built into the game, for example by saying: 'Hmm, not sure how to do that?'")
```

Recurrent actions (using functions)

In programming it is very common to reuse the same lines of code in your script.

You can just copy and paste these lines, which is maybe the easiest way to do it! (And it's perfectly fine to do it in this way!)

But...

if you want to be lazy... :--), or if you want to repeat some programmatic behavior, you can also use a **function**.

A function in Python is written in the following way:

```
In [ ]: def functionName():
            message = "hello!"
            return message
```

The required elements are:

- the word `def`
- a `functionName`
- the `()`
- the `:`
- and a `return` statement

Once you have written a function, you can use it in this way:

```
In [ ]: functionName()
```

So when we want to write a function that returns a description of the room when you say `look`:

```
In …  def look():
          message = "There are so many Apple computers
      here, but they're all locked."
          return message
```

```
In [ ]: look()
```

Let's insert that into the code from above:

```
message = "What do you want to do?"
print(message)
reply = input()

if "look" in reply:
    print(look())
elif "bag" in reply:
    print("Return what happens when you open your
bag.")
elif "get" in reply:
    print("Return what happens when you 'go'.")
elif "read" in reply:
    print("Return what happens when you 'read'.")
elif "exit" in reply:
    print("Return what happens when you want to exit
the game.")
else:
    print("Return what happens when the action is
not built into the game, for example by saying:
'Hmm, not sure how to do that?'")
```

But! There is more...

Functions do not have to return static content... You can use a function to execute lines of code for you.

Think of it as *wrapping* code in a box, that you give a name.

For example: we can write a bit of code that uses `random.choice()` and let a function return a random reply from a list.

```
In [ … from random import choice

       def randomDefaultReply():
           replies = [
               "Hmm, not sure how to do that?",
               "There is so much you can do, but this
       ain't one.",
               "Eh?"
           ]
           return choice(replies)
```

And let's use it in the same code again:

```
In [… message = "What do you want to do?"
      print(message)
      reply = input()

      if "look" in reply:
          print(look())
      elif "bag" in reply:
          print("Return what happens when you open your
      bag.")
      elif "get" in reply:
          print("Return what happens when you 'go'.")
      elif "read" in reply:
          print("Return what happens when you 'read'.")
      elif "exit" in reply:
          print("Return what happens when you want to
      exit the game.")
      else:
          print(randomDefaultReply())
```

In the same way, we can make a function for each *action* (`look` , `openBag` , `get` , `read`).

The `exit` action is a special one. Because we use a `while` loop, you can use the special word `break` to stop the game:

In [ ]:
```
while True:
    message = "What do you want to do?"
    print(message)
    reply = input()

    if "exit" in reply:
        break
```

Making and storing objects (using variables and lists)

To work with objects and rooms, we need to define them and store information about them.

To do this, we can use **variables** and **lists**.

For example, let's say that we call things that you can interact with *objects*.

When the game starts, there are 3 objects in the `room` (the XPUB studio) and 1 already in your `bag` (the lamp).

In [ ]:
```
room = ["breadcube", "cup", "book"]
bag = ["lamp"]
```

To **add** an object into your `bag` we can use `.append()`:

In [ ]:
```
bag.append("breadcube")
```

In [ ]:
```
bag
```

To **remove** an object from your `bag` we can use `.remove()`:

In [ ]:
```
bag.remove("breadcube")
```

In [ ]:
```
bag
```

With these variables, we can make functions for the actions `look`, `openBag` and `get`:

In [...

```python
def look():
    message = f"""
    There are so many Apple computers here, but they're all locked.
    What else is here? { room }
    """
    return message

def openBag():
    message = f"""
    Your bag currently holds: { bag }
    """
    return message

def get(obj):
    if obj in room:
        room.remove(obj)
        bag.append(obj)
        print(f"[DEBUG] Currently in the room: { room }")
        print(f"[DEBUG] Currently in the bag: { bag }")
        message = f"Added { obj } to your bag!"
    else:
        message = f"There is no { obj } here."

    return message
```

Store information about objects (using dictionaries)

If we want to store information about the objects, we can use a **dictionary**.

Dictionaries are used to store data in `key : value` pairs.

`keys` and `values` can be **strings** or **numbers**. But they can also be a **list**, **tuple** or **dictionary** (dictionary in a dictionary). It really depends on what data you want to store!

In [ ]:
```
candy = {
    "type" : "drop",
    "shapes" : ["round", "square", "triangle"],
    "left" : 10,
    "brands" : {
        "haribo" : "not so nice",
        "klene" : "not too bad",
        "oldtimers" : "jummy"
    }
}
```

You can read the information from the dictionary like this:

In [ ]:
```
candy["type"]
```

Back to the text-adventure game...

It would be useful to store **information** about each of the objects, so let's make a dictionary and store it there.

Let's say that the objects are important for the game. They hold hidden **clues** that the player can `read`:

I…
```
clues = {
    "lamp" : "There is a small note scribled on the bottom of the lamp: 'check breadcube!' it says. Hmm...",
    "breadcube" : "You turn breadcube on... luckily you remember the root password by heart. The welcome message shows up on the screen: FIT LIKES COFFEE!",
    "cup" : "The cup is empty, someone else seemed to have poored coffee to FIT already... Hmm... You take the cup. Underneath it is a book.",
    "book" : "The book is called 'The hidden secrets of the ultra-smart-sensor-device for the XPUB studio'. Hmm..."
}
```

We can make the code return these clues to the player.

So for example, when you type `read lamp`, the clue is printed on the screen.

We can write it like this:

In [ ]:
```
clues["lamp"]
```

But to connect the object that is mentioned in the reply to the dictionary, we need to *parse* the reply, so we know what is the `action` and what is the `object`.

```
In [ ]: message = "What do you want to do?"
        print(message)
        reply = input()

        command = reply.split()
        print(f"[DEBUG] Your command: { command }")

        action = command[0]
        obj = command[1]

        print(f"[DEBUG] Your action: { action }")
        print(f"[DEBUG] Your object: { obj }")
```

The object is stored as a variable now (`obj`), which we can use to read the clue from the dictionary:

```
In [ ]: if "read" in action:
            print(clues[obj])
```

And like the other actions, you can store this as a function called `read` :

```
In [ ]: def read(obj):
            if obj in clues:
                message = clues[obj]
            else:
                message = f"You can't read { obj }"

            return message
```