CS50's Introduction to Programming with Python

cs50.harvard.edu/python/2022/notes/3

Lecture 3

Exceptions

- Exceptions are things that go wrong within our coding.
- In our text editor, type `code hello.py` to create a new file. Type as follows (with the intentional errors included):

  ```
  print("hello, world)
  ```

  Notice that we intentionally left out a quotation mark.

- Running `python hello.py` in our terminal window, an error is outputted. The compiler states that it is a "syntax error."" Syntax errors are those that require you to double-check that you typed in your code correction.

- You can learn more in Python's documentation of Errors and Exceptions.

Runtime Errors

- Runtime errors refer to those created by unexpected behavior within your code. For example, perhaps you intended for a user to input a number, but they input a character instead. Your program may throw an error because of this unexpected input from the user.
- In your terminal window, run `code number.py`. Code as follows in your text editor:

  ```
  x = int(input("What's x? "))
  print(f"x is {x}")
  ```

  Notice that by including the `f`, we tell Python to interpolate what is in the curly braces as the value of `x`. Further, testing out your code, you can imagine how one could easily type in a string or a character instead of a number. Even still, a user could type nothing at all – simply hitting the enter key.

- As programmers, we should be defensive to ensure that our users are entering what we expected. We might consider "corner cases" such as `-1`, `0`, or `cat`.

- If we run this program and type in "cat", we'll suddenly see `ValueError: invalid literal for int() with base 10: 'cat'` Essentially, the Python interpreter does not like that we passed "cat" to the `print` function.

- An effective strategy to fix this potential error would be to create “error handling” to ensure the user behaves as we intend.
- You can learn more in Python’s documentation of Errors and Exceptions.

`try`

- In Python `try` and `except` are ways of testing out user input before something goes wrong. Modify your code as follows:

```
try:
    x = int(input("What's x?"))
    print(f"x is {x}")
except ValueError:
    print("x is not an integer")
```

  Notice how, running this code, inputting `50` will be accepted. However, typing in `cat` will produce an error visible to the user, instructing them why their input was not accepted.

- This is still not the best way to implement this code. Notice that we are trying to do two lines of code. For best practice, we should only `try` the fewest lines of code possible that we are concerned could fail. Adjust your code as follows:

```
try:
    x = int(input("What's x?"))
except ValueError:
    print("x is not an integer")

print(f"x is {x}")
```

  Notice that while this accomplishes our goal of trying as few lines as possible, we now face a new error! We face a `NameError` where `x is not defined`. Look at this code and consider: Why is x not defined in some cases?

- Indeed, if you examine the order of operations in `x = int(input("What's x?"))`, working right to left, it could take an incorrectly inputted character and attempt to assign it as an integer. If this fails, the assignment of the value of `x` never occurs. Therefore, there is no x to print on our final line of code.

`else`

- It turns out that there is another way to implement `try` that could catch errors of this nature.

- Adjust your code as follows:

```
try:
    x = int(input("What's x?"))
except ValueError:
    print("x is not an integer")
else:
    print(f"x is {x}")
```

  Notice that if no exception occurs, it will then run the block of code within `else`. Running `python number.py` and supplying `50`, you'll notice that the result will be printed. Trying again, this time supplying `cat`, you'll notice that the program now catches the error.

- Considering improving our code, notice that we are being a bit rude to our user. If our user does not cooperate, we currently simply end our program. Consider how we can use a loop to prompt the user for `x` and if they don't prompt again! Improve your code as follows:

```
while True:
    try:
        x = int(input("What's x?"))
    except ValueError:
        print("x is not an integer")
    else:
        break

print(f"x is {x}")
```

  Notice that `while True` will loop forever. If the user succeeds in supplying the correct input, we can break from the loop and then print the output. Now, a user that inputs something incorrectly will be asked for input again.

Creating a Function to Get an Integer

- Surely, there are many times that we would want to get an integer from our user. Modify your code as follows:

```
def main():
    x = get_int()
    print(f"x is {x}")


def get_int():
    while True:
        try:
            x = int(input("What's x?"))
        except ValueError:
            print("x is not an integer")
        else:
            break
    return x


main()
```

  Notice that we are manifesting many great properties. First, we have abstracted away the ability to get an integer. Now, this whole program boils down to the first three lines of the program.

- Even still, we can improve this program. Consider what else you could do to improve this program. Modify your code as follows:

```
def main():
    x = get_int()
    print(f"x is {x}")


def get_int():
    while True:
        try:
            x = int(input("What's x?"))
        except ValueError:
            print("x is not an integer")
        else:
            return x


main()
```

  Notice that `return` will not only break you out of a loop, but it will also return a value.

- Some people may argue you could do the following:

```
def main():
    x = get_int()
    print(f"x is {x}")


def get_int():
    while True:
        try:
            return int(input("What's x?"))
        except ValueError:
            print("x is not an integer")


main()
```

  Notice this does the same thing as the previous iteration of our code, simply with fewer lines.

pass

- We can make it such that our code does not warn our user, but simply re-asks them our prompting question by modifying our code as follows:

```
def main():
    x = get_int()
    print(f"x is {x}")


def get_int():
    while True:
        try:
            return int(input("What's x?"))
        except ValueError:
            pass


main()
```

  Notice that our code will still function but will not repeatedly inform the user of their error. In some cases, you'll want to be very clear to the user what error is being produced. Other times, you might decide that you simply want to ask them for input again.

- One final refinement that could improve the implementation of this `get_int` function. Right now, notice that we are relying currently upon the honor system that the `x` is in both the `main` and `get_int` functions. We probably want to pass in a prompt that the user sees when asked for input. Modify your code as follows.

```
def main():
    x = get_int("What's x? ")
    print(f"x is {x}")


def get_int(prompt):
    while True:
        try:
            return int(input(prompt))
        except ValueError:
            pass


main()
```

- You can learn more in Python's documentation of `pass`.

Summing Up

Errors are inevitable in your code. However, you have the opportunity to use what was learned today to help prevent these errors. In this lecture, you learned about…

- Exceptions
- Value Errors
- Runtime Errors
- `try`
- `else`
- `pass`