Lecture 4

January 25, 2023

1 Loops: "for" v.s. "while"

- "For" loops are used to iterate over a sequence of items.The general syntax of a for loop is:

```
numbers = [1, 2, 3, 4, 5]
for number in numbers:
    print(number)
```

- "While" loops are used to repeat an action while a certain condition is true. The general syntax of a while loop is:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

Problem:

1. Use a **while** loop to print the numbers from 1 to 10.
2. Use a **while** loop to print out the first three even numbers from a given list.

Hint:

```
def three_first_even(a):
    # code here
```

Expected outcome:

```
a=[10,3,6,7,20, 7, 2, 0, 11]
three_first_even(a)
# output: 10, 6, 20
```

[ ]:
```
# Solution for problem 1:
# Use a while loop to print the numbers from $1$ to $10$.
count = 1
while count <= 10:
    print(count)
    count += 1
```

[ ]:
```
# Solution to problem 2:
# Use a while loop to print out the first three even numbers from a given list.

```

```
def three_first_even(a):
    count = 0
    ind = 0
    while count<3 and ind< len(a):
        if a[ind]%2 ==0:
            print(a[ind])
            count += 1
        ind += 1
a=[10,3,0,2,7, 7, 20, 11]
three_first_even(a)
```

2 Control flow statements

In Python, the **break**, **continue**, and **pass** statements are used to control the flow of execution within loops.

2.1 The "break" statement

- The **break** statement is used to exit a loop early, before the loop condition is met.
- When the interpreter encounters a **break** statement, it immediately exits the loop and continues with the next statement after the loop.

[ ]:
```
numbers = [1, 2, 3, 4, 5]
for number in numbers:
    if number == 4:
        break
    print(number)
```

2.2 The "continue" statement

- The **continue** statement is used to skip the current iteration of a loop and move on to the next iteration.
- When the interpreter encounters a **continue** statement, it immediately jumps to the next iteration of the loop and continues with the next iteration.

[ ]:
```
# Example for continue
numbers = [1, 2, 3, 4, 5]
for number in numbers:
    if number == 4:
        continue
    print(number)
```

2.3 The "pass" statement

The **pass** statement is used as a **placeholder** in the code. It does not do anything and is often used when a statement is required syntactically, but no action needs to be performed.

[ ]:
```
#Example for pass
numbers = [1, 2, 3, 4, 5]
for number in numbers:
    if number == 4:
        pass
    else:
        print(number)
```

3 The "input" function

- The **input()** function is used to read input from the user.
- It takes an **optional string argument** which is used as a **prompt** to the user.
- The **input()** function waits for the user to type in input and press the enter key.
- The **input()** is returned as a string, so if you want to use the input as a number, you need to convert it using appropriate type casting.

[ ]:
```
name = input("Your name:")
print ("My name is", name)
```

[ ]:
```
def square():
    x = input('Enter your number:')
    x = int(x)
    return x**2
square()
```

Practice problem: Write down a function to check that whether a number given by the user is even or odd

[ ]:
```
def even_odd():
    x = input("Give me your number:")
    x = int(x)
    if x%2 == 0:
        print('Even')
    else:
        print("Odd")
even_odd()
```

3.1 Error Handling: "Try-Except"

Try-except statements are used to handle errors in a program.

The general syntax of a **try-except** statement is:

```
try:
    # code that might raise an error
except ErrorType:
    # code to handle the error
```

The **try** block lets you test a block of code for errors.

The **except** block lets you handle the error.

[ ]:
```
try:
    x = 5/0
except ZeroDivisionError:
    print("hi there, we cannot divide by zero")
```

[ ]:
```
## Raising Custom Errors
def divide(x, y):
    if y == 0:
        raise ValueError("hi there, we cannot divide by zero")
    return x / y
divide(3,0)
```

3.2 Handling Errors: more!

The **try** block lets you test a block of code for errors.

The **except** block lets you handle the error.

The **else** block lets you execute code when there is no error.

The **finally** block lets you execute code, regardless of the result of the try- and except blocks

[ ]:
```
## Using try-except-else-finally blocks
try:
    x = 5
    y = 0
    result = x / y
except ZeroDivisionError:
    print("Cannot divide by zero")
else:
    print(result)
finally:
    print("This block will always be executed.")
```

Problem: 1. Use a try-except block to handle a TypeError when trying to add a string to a number. 2. Raise a custom ValueError if a variable is assigned a negative value. 3. Use a try-except-else block to handle a FileNotFoundError when trying to open a file.

[ ]:
```
try:
    a = 6
    b = 'hello'
    a + b
except TypeError:
    print('We cannot add a string to an integer')
```

[ ]:
```
try:
    a = 6
    b = 'hello'
    a + b
except TypeError:
    print('We cannot add a string to an integer')
```

[ ]:
```
try:
    num = 5
    string = "Hello"
    result = num + string
except TypeError:
    raise TypeError("Cannot add a string and a number")
```

4 Check out ticket

Problem: Write a program to let a user repeatedly input two numbers a, b.

- Return the quotient $\frac{a}{b}$ if possible. Otherwise, raise the error message "Watch out: b==0!!!" and exit the program.
- Exit the program if the user inputs "I'm done!".