Introduction to Programming in Python

Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Control Flow

Loops

- A loop is used whenever a bunch of statements needs to be executed more than once.
- Python has two kinds of loops: `while` loops and `for` loops.
- A `while` loop is used to run some code as long as some condition is true.
- A `for` loop is used to run some code a pre-specified number of times.

`While` loops

- Similar to `if`, but jumps back to the `while` statement after the `while` block has finished.

In [ ]:

```
x = -1 # why is this needed?
while x < 0 or x > 9:
    x = int(input("Enter a number between 0 and 9: "))
    if x < 0:
        print("You have entered a negative number.")
    elif x > 9:
        print("You have entered a number greater than 9.")
print("Thank you. You entered " + str(x) + ".")
```

- Alternative implementation:

In [ ]:

```
while True:
    x = int(input("Enter a number between 0 and 9: "))
    if x < 0:
        print("You have entered a negative number.")
    elif x > 9:
        print("You have entered a number greater than 9.")
    else:
        print("Thank you. You entered " + str(x) + ".")
        break  # exit innermost enclosing loop.
```

Remark: Like `if` blocks, `while` loops can have an `else` block. It is executed when the condition is (or becomes) false. The same is of course true if one just puts the code in the `else` block *after* the loop like in our first implementation, but the two approaches differ when there is a `break` statement. Here is our first implementation again, modified to use this:

In [ ]:

```
x = -1 # why is this needed?
while x < 0 or x > 9:
    x = int(input("Enter a number between 0 and 9: "))
    if x < 0:
        print("You have entered a negative number.")
    elif x > 9:
        print("You have entered a number greater than 9.")
else:
    print("Thank you. You entered " + str(x) + ".")
```

Exercise: Number Guessing Game

Implement the following game: The computer chooses a random number, and the player has to guess it. After guessing, the player receives feedback as to whether they guessed correctly, or too low, or too high. The game ends when the correct number has been guessed. Start with the skeleton below.

In [ ]:

```
import random # a standard library (i.e., built in) module. more on this later
lower = 0
upper = 100
to_guess = random.randint(lower, upper) # a function from that library. has to be prepended by th
current_guess = -1
while current_guess != to_guess:
    break
```

`for` loops

This is the other kind of loop supported by Python. It iterates over the elements of a sequence (e.g., a list):

In [ ]:
```
for letter in "Python":
    print(letter)
```

`letter` is called the loop variable. Every time the loop body is executed, it will in turn assume the value of each element of the sequence.

Range objects

`range` objects are often useful in conjunction with `for` loops, especially for numerical computations. They represent sequences of integers.

Syntax:

In [ ]:
```
myrange = range(1, 10, 2) # start, stop, step; c.f. slicing
print(myrange)
```

In [ ]:
```
type(myrange)
```

In [ ]:
```
list(myrange)
```

- As you can see, we have to convert the `range` to a `list` in order to see its contents (this shouldn't be done in practice).
- Reason: a range doesn't actually materialize its contents upon creation, only when it is used.
- This is called `lazy` computation.
- Advantage: `range`s can be huge, without consuming any memory.
- Usage in `for` loops:

In [ ]:

```
squares = []
for element in range(1, 11): # step is optional
    squares.append(element ** 2)
print(squares)
```

Quiz: What does the following compute?

In [ ]:

```
n = 5
result = 1
for i in range(1, n+1):
    result = result * i # or shorter: result *= 1
print(result)
```

`for` loops support `break` for early stopping, and an `else` clause which is executed when the loop terminates regularly (i.e., not via `break`):

In [ ]:
```
import random
haystack = random.sample(range(1, 11), 10)
print(haystack)
```

In [ ]:
```
needle = 6
counter = 0
for elem in haystack:
    if elem == needle:
        print(needle, "found at position " + str(counter) + ".")
        break
    counter += 1
else:
    print(needle, " not found.")
```

(we could just have used `haystack.index(needle)` of course).

Loops can be nested:

In [ ]:

```
for row in range(1, 6):
    for col in range(row):
        print("*", end="") # end="" suppresses the newline
    print()
```

Exercise

Complete the following code snippet so that it prints out the primes up to `N` (it currently prints out all numbers between 2 and `N`). Hints:

- `pass` is a syntactical placeholder. This is what you need to replace.
- The "modulo" operater `%` returns the remainder after division, i.e., `a % b == 0` is `True` if and only if `a` is divisible by `b`.
- use `break`.

In [ ]:

```
N = 100
for i in range(2, N+1):
    for j in range(2, i):
        pass
    else:
        pass
```

Note: this algorithm is not efficient. A more efficient algorithm is the Sieve of Eratosthenes.

Homework

Add some bells and whistles to the guessing game

1. The player only gets a limited number of guesses, say `n=10`. When reached, the game stops with a message given to the player.
2. The player can stop the game by entering a negative number. The computer then says "Bye."
3. Improved feedback: if the player guesses too high (say `to_guess` is 50 and the user guesses 75), then in the next round, the computer asks for a number between 0 and 75 in the next round. Analogously if the guess is too low.

More Homework

`while`

- Intermediate exercises 9a, 9b from https://holypython.com/
- Optional: Exercise 9 (hard) from https://www.w3resource.com/python-exercises/python-conditional-statements-and-loop-exercises.php

Note for the sample solution of Ex. 9: two variables can be assigned at the same time as follows:

In [ ]:
```
a, b = 1, 2
print(a, b)
```

`for`

- Exercises 1, 4, 6 from https://www.w3resource.com/python-exercises/python-conditional-statements-and-loop-exercises.php