OMIS 30: Intro to Programming (with Python)

Week 3, Class 1

Introduction to Programming

Instructor: Denis Vrdoljak

Dictionary (dict)

- A mutable unordered set of key:value pairs, with unique keys
  - Syntax: sound_dict = {“cat”: “meow”, “duck”: “quack”}
  - To access a given value, call the key:
    - >> sound_dict[“duck”]
    - Returns: “quack”
  - To assign a new key:value pair or reassign an existing key:
    - >> sound_dict[“cow”] = “moo”
    - >> print(sound_dict)
    - Returns: {“cat”: “meow”, “duck”: “quack”, “cow”: “moo”}

Dictionary Methods and Operations

- sound_dict = {“cat”: “meow”, “duck”: “quack”, “cow”: “moo”}
- .keys() - returns a list of the keys of the dictionary
  - sound_dict.keys() = dict_keys(['cat', 'duck', 'cow'])
- .values() - returns a list of the values of the dictionary
  - sound_dict.values() = dict_values(['meow', 'quack', 'moo'])
- .items() - returns a list of tuples of (key,value)
  - sound_dict.items() = dict_items([('cat', 'meow'), ('duck', 'quack'), ('cow', 'moo')])

Tuple

- An immutable ordered list with a known number of elements.
  - Syntax: x = (1,4,6)
  - Immutability refers to the inability to be changed after the original assignment.
  - Tuples, like all the primitive data types, are immutable.

Set

- An unordered collection of UNIQUE items.
  - Syntax: x = {4,1,6} or x = set((4,1,6))
    - If y = {4,4,6,1} -> y = {4,6,1} (the extra 4 is removed because its not unique item)
  - Cannot update an item only add or remove
    - set.add() -> adds that item to the set
      - x.add(7) -> {1, 4, 6, 7}
    - remove() -> removes that item from the set
      - x.remove(1) -> {6, 4, 7}

Basic Built-ins

- Introducing a few useful/common built in functions:
  - len() <- returns the length of that object
  - type() <- returns the type of that object

Advanced Built-ins

- **zip()**
  - Used to pair up the elements of two lists (or other iterable) based on shared index
    - odd = (1,3,5), even = (2,4,6)
    - >> print(list(zip(odd, even)))

      [(1,2),(3,4),(5,6)]
  - Can also be used with dictionaries:
    - students = [“Matt”, “Jane”, “Bob”], grades = [82, 97, 70]
    - >> print(dict(zip(names, grades)))

      {“Matt”:82, “Jane”:97, “Bob”:70}

IF Statements

- Creates a condition, which if True, executes the block of code. If False, it skips over it.
- Format:

  if boolean_condition:

      #code to execute

- The block of code is indented with a tab (or 4 spaces)
- The IF block ends when the indentation ends

Nested IF Statements

- IF statements can be nested

```
If my_animal == “mammal”:
    if my_animal_species == “dog:
                print(“It’s a dog!”)
print(“It’s a mammal”)
```

IF-ELSE

- ELSE blocks execute if the IF condition is False

```
if my_animal == “mammal”:
    if my_animal == “dog:
                print(“It’s a dog!”)
    else:
        print(“It’s a mammal, but not a dog”)
```

ELIF (else if)

- ELIF executes a second IF condition, if the first condition is False

```
if my_animal == “mammal”:
    if my_animal == “dog:
            print(“It’s a dog!”)
    elif my_animal == “cat”:
            print(“It’s a cat!”)
    else:
        print(“It’s a mammal, but not a dog or a cat”)
```

IF statement practice:

- What's the expected output?

```
x = [2,4,6,8]

if x:
    print(x)
else:
    print("No x to be found!")
```

IF statement practice:

- What's the expected output?

```
x = [2,4,6,8]

if x:
    print(x)
else:
    print("No x to be found!")
```

Output: [2,4,6,8]

IF statement practice:

- What's the expected output?

```
x =‘’

if x:
    print(x)
else:
    print(”no data (string) to be found!")
```

IF statement practice:

- What's the expected output?

```
x =‘’

if x:
    print(x)
else:
    print(”no data (string) to be found!")
```

Output: [2,4,6,8]

IF, ELIF, ELSE example

OPEN FILES
- test_3.py
- test_2.py
- test.py

test_3.py | test_2.py

```
1  #get user input for favorite number
2  fav = input('What is your favorite number: ')
3
4  #print different results based on value of number
5  if int(fav) <= 50:
6      print("That's a good number")
7  elif int(fav) < 1000:
8      print("Interesting choice")
9  else:
10     print("That's a high number!")
11
```

```
Bens-MacBook-Pro:~ benthompson$ python ~/Desktop/test_3.py
What is your favorite number: 50
That's a good number
Bens-MacBook-Pro:~ benthompson$ python ~/Desktop/test_3.py
What is your favorite number: 500
Interesting choice
Bens-MacBook-Pro:~ benthompson$ python ~/Desktop/test_3.py
What is your favorite number: 1001
That's a high number!
Bens-MacBook-Pro:~ benthompson$
```

WHILE loops

- Creates a condition and while it is True the block of code below is repeatedly executed. Once it is not true, the execution stops. (Checked in between loops.)
- Format:

  while boolean_condition:

      #code to execute

- The block of code is indented with a tab (or 4 spaces)
- The while block ends when the indentation ends

The format is the same a simple IF statement. It uses the same kinds of boolean conditions. But, a WHILE loop repeats until the boolean condition evaluates as False when it's next evaluated, while IF statements just execute once (if the boolean condition is true).

WHILE loops

Example:

```
>>          Index = 0
        while Index <10:
            #code to do
            print(Index)
            #increment index
            Index += 1
```

Output: (0,1,2,3,4,5,6,7,8,9)

WHILE loop teaching example

What if we want to check if every element in "x" is even?

```
x = [4,5,6,6.5,7]
i = 0
# i is commonly used for the index of an object
# while loops require explicitly instantiating the counter

while i < len(x):
    if x[i] % 2 == 0:
                                        # % is called the modulus operator
                                        # % yields the remainder from the division of the first argument
by the second
        print(x[i], "is even")
    else:
        print(x[i], "is not even")

    i += 1
```

WHILE loop teaching example

```
x = [4,5,6,6.5,7]
i = 0

while i < len(x):
    if x[i] % 2 == 0:
        print(x[i], "is even")
    else:
        print(x[i], "is not even")

    i += 1
```

Output:

4 is even

5 is not even

6 is even

6.5 is not even

7 is not even

WHILE loops - Break

```
1  x = [0,1,2,3,4,5]
2  while True:
3      x.pop()
4      print(x)
```

```
[0, 1, 2, 3, 4]
[0, 1, 2, 3]
[0, 1, 2]
[0, 1]
[0]
[]

---------------------------------------------
IndexError                                Trace
<ipython-input-70-bbc18fe3391b> in <module>()
      1 x = [0,1,2,3,4,5]
      2 while True:
----> 3     x.pop()
      4     print(x)

IndexError: pop from empty list
```

Without a stopping condition, the loop throws an error. Use **break** to gracefully stop a loop.

```
1  x = [0,1,2,3,4,5]
2  while True:
3      x.pop()
4      print(x)
5      if len(x) < 2:
6          break
```

```
[0, 1, 2, 3, 4]
[0, 1, 2, 3]
[0, 1, 2]
[0, 1]
[0]
```

WHILE loops - Continue

Using “continue” passes the control back to the top of the loop without exiting the loop, allowing certain conditions to not engage the rest of the loop.

```
1  x = [0,1,2,3,4,5]
2  while True:
3      x.pop()
4      if x[-1] == 2 or x[-1] == 4:
5          continue
6      print(x)
7      if len(x) < 2:
8          break
```

```
[0, 1, 2, 3]
[0, 1]
[0]
```

Nested WHILE loops:

Can insert while loops inside each other!

As more loops are ‘nested’ inside of each other, need to be more cognisant of accidentally making an infinite loop (a loop that never reaches the conditions to terminate).

Nested WHILE example (find the prime numbers):

```
i = 2
while i < 100:
    j = 2
    while j <= i/j:
        if i%j == 0:
            break
        j += 1
    if j > i/j:
        print(i, "is prime")
    i += 1

print("Complete!")
```

Read this carefully, and try to figure out what's going on here. We didn't go through this one in detail in class, but going through this one is good practice to better understand while loops (even if you get a little lost in the algorithm for identifying primes).

IF STATEMENTS REVIEW

IF Statements

- Creates a condition, which if True, executes the block of code. If False, it skips over it.
- Format:

  if boolean_condition:

      #code to execute

- The block of code is indented with a tab (or 4 spaces)
- The IF block ends when the indentation en

IF Statements

- Creates a condition, which if True, executes the block of code. If False, it skips over it.

```
if my_animal == “mammal”:
    print(“It’s a mammal”)
```

- The block of code is indented with a tab (or 4 spaces)
- The IF block ends when the indentation en

Nested IF Statements

- IF statements can be nested

```
If my_animal == “mammal”:
    if my_animal == “dog:
                print(“It’s a dog!”)
print(“It’s a mammal”)
```

IF-ELSE

- ELSE blocks execute if the IF condition is False

```
if my_animal == “mammal”:
    if my_animal == “dog:
                print(“It’s a dog!”)
    else:
        print(“It’s a mammal, but not a dog”)
```

ELIF (else if)

- ELIF executes a second IF condition, if the first condition is False

```
if my_animal == “mammal”:
    if my_animal == “dog:
            print(“It’s a dog!”)
    elif my_animal == “cat”:
            print(“It’s a cat!”)
    else:
        print(“It’s a mammal, but not a dog or a cat”)
```