2/15/2023

Mathematics Association of Nairobi University
isaak@students.uonbi.ac.ke

Booleans, Comparison Operators, and Logical Operators

The bool data type holds one of the values `True` or `False`, which are often encoded as `1` or `0`, respectively.

There are 6 comparison operators that are common to see in order to obtain a `bool` value:

Comparison Operators

Comparison Operators

Symbol Use Case	Bool	Operation
5 < 3	False	Less Than
5 > 3	True	Greater Than
3 <= 3	True	Less Than or Equal To
3 >= 5	False	Greater Than or Equal To
3 == 5	False	Equal To
3 != 5	True	Not Equal To

And there are three logical operators you need to be familiar with:

Logical Use	Bool	Operation
5 < 3 `and` 5 == 5	False	`and` - Evaluates if all provided statements are True
5 < 3 `or` 5 == 5	True	`or` - Evaluates if at least one of many statements is True
`not` 5 < 3	True	`not` - Flips the Bool Value

Quiz 3: Which is denser, Nairobi or Mombasa?

Try comparison operators in this quiz! This code calculates the population densities of `Nairobi` and `Mombasa`.

Write code to compare these densities. Is the population of `Mombasa` more dense than that of `Nairobi`? Print `True` if it is and `False` if not.

Take populations and area to be:

nairobi_population = 864816

nairobi_area = 231.89

mombasa_population = 6453682

mombasa_area = 486.5

> > Write code that prints `True` if Nairobi is denser than Mombasa, and `False` otherwise

Data Structures

You will learn to make use of new data structures, which group and order data in different ways, to help you solve problems.

Types of Data Structures:

```
Lists, Tuples, Sets, Dictionaries, Compound Data Structures
```

Lists

Lists are used to store multiple items in a single variable.

Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are `Tuple`, `Set`, and `Dictionary`, all with different qualities and usage.

Lists are created using square brackets:

In [17]:
```
mylist = ["apple", "banana", "cherry", "apple", "cherry"]
print(mylist)
```

```
['apple', 'banana', 'cherry', 'apple', 'cherry']
```

List items are:

1. ordered:

- means that the items have a defined order, and that order will not change. If you add new items to a list, the new items will be placed at the end of the list.

1. changeable:

- meaning that we can change, add, and remove items in a list after it has been created

1. allow duplicate values.

In [19]:
```
friends = ["Albert"]
friends.append("Oscar")
friends.append("Angela")
friends.insert(0, "Kevin")
print(friends)
```

```
['Kevin', 'Albert', 'Oscar', 'Angela']
```

Slice and Dice with Lists

You saw that we can pull more than one value from a list at a time by using `slicing`. When using slicing, it is important to remember that the lower index is `inclusive` and the upper index is `exclusive`.

Therefore, this:

In [23]:
```
list_of_random_things = [1, 3.4, 'a string', True]
list_of_random_things[1:2]
```

Out[23]:
```
[3.4]
```

will only return `3.4` in a list. Notice this is still different than just indexing a single element, because you get a list back with this indexing. The colon tells us to go from the starting value on the left of the colon up to, but not including, the element on the right.

If you know that you want to start at the beginning, of the list you can also leave out this value.

In [24]:
```
list_of_random_things[:2]
```

Out[24]:
```
[1, 3.4]
```

or to return all of the elements to the end of the list, we can leave off a final element.

In [25]:
```
list_of_random_things[1:]
```

Out[25]:
```
[3.4, 'a string', True]
```

Membership Operators

You saw that we can also use `in` and `not in` to return a `bool` of whether an element exists within our list, or if one string is a substring of another.

`in` evaluates if an element exists within our list `not in` evaluates if an element does not exist within our list

In [27]:
```
'this' in 'this is a string'
```

Out[27]:
```
True
```

In [28]:
```
'isa' in 'this is a string'
```

Out[28]:
```
False
```

In [29]:
```
5 in [1, 2, 3, 4, 6]
```

Out[29]:
```
False
```

Quiz 3 List Indexing

Use list indexing to determine how many days are in a particular month based on the integer variable month, and store that value in the integer variable num_days. For example, if `month` is `8`, `num_days` should be set to 31, since the eighth month, August, has 31 days.

Remember to account for zero-based indexing!

In [34]:
```
month = 8
days_in_month = [31,28,31,30,31,30,31,31,30,31,30,31]

# use list indexing to determine the number of days in month
num_days = days_in_month[month - 1]

print(num_days)
```

```
31
```

Tuples

A tuple is another useful container. It's a data type for immutable ordered sequences of elements. They are often used to store related pieces of information. Consider this example involving latitude and longitude:

In [35]:
```
location = (13.4125, 103.866667)
print("Latitude:", location[0])
print("Longitude:", location[1])
```

```
Latitude: 13.4125
Longitude: 103.866667
```

Tuples are similar to lists in that they store an ordered collection of objects which can be accessed by their indices. Unlike lists, however, tuples are immutable - you can't add and remove items from tuples, or sort them in place.

Tuples can also be used to assign multiple variables in a compact way.

In [36]:
```
dimensions = 52, 40, 100
length, width, height = dimensions
print("The dimensions are {} x {} x {}".format(length, width, height))
```

```
The dimensions are 52 x 40 x 100
```

Sets

A set is a data type for mutable unordered collections of unique elements. One application of a set is to quickly remove duplicates from a list.

In [37]:
```
numbers = [1, 2, 6, 3, 1, 1, 6]
unique_nums = set(numbers)
print(unique_nums)
```

```
{1, 2, 3, 6}
```

Sets support the `in` operator the same as lists do. You can `add` elements to sets using the add method, and remove elements using the `pop` method, similar to lists. Although, when you pop an element from a set, a random element is removed. Remember that sets, unlike lists, are unordered so there is no "last element"

In [38]:
```
fruit = {"apple", "banana", "orange", "grapefruit"}  # define a set

print("watermelon" in fruit)  # check for element
```

```
fruit.add("watermelon")  # add an element
print(fruit)

print(fruit.pop())  # remove a random element
print(fruit)
```

```
False
{'banana', 'watermelon', 'orange', 'apple', 'grapefruit'}
banana
{'watermelon', 'orange', 'apple', 'grapefruit'}
```

In [39]:
```
fruits = {"apple", "banana", "orange", "grapefruit,","apple"}
fruits
```

Out[39]:
```
{'apple', 'banana', 'grapefruit,', 'orange'}
```

Other operations you can perform with sets include those of mathematical sets. Methods like union, intersection, and difference are easy to perform with sets, and are much faster than such operators with other containers.

Dictionaries

A `dictionary` is a mutable data type that stores mappings of unique `keys` to `values`. Here's a dictionary that stores elements and their atomic numbers.

In [40]:
```
elements = {"hydrogen": 1, "helium": 2, "carbon": 6}
```

In general, dictionaries look like key-value pairs, separated by commas: `{key1:value1, key2:value2, key3:value3, key4:value4, ...}`

In [41]:
```
random_dict = {"abc": 1, 5: "hello"}
```

We can look up values in the dictionary using square brackets " `[]` " around the key, like :

`dict_name[key]`.

For example, in our random dictionary above, the value for `random_dict["abc"]` is 1, and the value for `random_dict[5]` is "hello".

In our elements dictionary above, we could print out the atomic number mapped to helium like this:

In [43]:
```
print(elements["helium"])
```

```
2
```

In [46]:
```
random_dict[5]
```

Out[46]:
```
'hello'
```

Control Flow

`Control flow` describes the order in which your lines of code are run. This order is usually different than the sequence in which the lines of code appear! Execution can flow from one place in the code to another

we'll learn about several tools in Python we can use to affect our code's control flow:

- Conditional Statements
- Boolean Expressions
- For and While Loops
- Break and Continue

Indentation

Some other languages use braces to show where blocks of code begin and end. In Python we use indentation to enclose blocks of code. For example, `if` statements use indentation to tell Python what code is inside and outside of different clauses.

In Python, indents conventionally come in multiples of four spaces. Be strict about following this convention, because changing the indentation can completely change the meaning of the code. If you are working on a team of Python programmers, it's important that everyone follows the same indentation convention!

Spaces or Tabs?

The Python Style Guide (https://peps.python.org/pep-0008/#tabs-or-spaces) recommends using 4 spaces to indent, rather than using a tab. Whichever you use, be aware that "Python 3 disallows mixing the use of tabs and spaces for indentation."

If Statement

An if statement is a conditional statement that runs or skips code based on whether a condition is true or false. Here's a simple example.

In [42]:
```
phone_balance = 20
bank_balance = 1000

if phone_balance < 20:
    phone_balance += 50
    bank_balance -= 50

print("Your phone_balance is: ", phone_balance)
print("Your bank_balance is: ", bank_balance)
```

```
Your phone_balance is:  20
Your bank_balance is:  1000
```

In [35]:
```
number = 145

if number % 2 == 0: # == is called comparison operators
    print("Number " + str(number) + " is even.")
else:
    print("Number " + str(number) + " is odd.")
```

```
Number 145 is odd.
```

To get inputs from the user we use `input()` function

In [28]:
```
#Example of how to get input from users
name = input('Please Enter your Name: ')
print("Hello " + name + " and Welcome to Data Analysis Class")
```

```
Please Enter your Name: John Doe
Hello John Doe and Welcome to Data Analysis Class
```

In [47]:
```
# A simple program for checking whether a number is odd or even
name = input('Please Enter your Name: ')
number = input("Please Enter Your Number: ")
number = int(number)

if number % 2 == 0:
    print("Hello " + name + "Your Number: " + str(number) + " is even.")
else:
    print("Hello " + name + " Your Number " + str(number) + " is odd.")
```

```
Please Enter your Name: Jane Doe
Please Enter Your Number: 13
Hello Jane Doe Your Number 13 is odd.
```

If, Elif, Else

In addition to the `if` clause, there are two other optional clauses often used with an `if` statement.

1. `if`: An if statement must always start with an if clause, which contains the first condition that is checked. If this evaluates to True, Python runs the code indented in this if block and then skips to the rest of the code after the if statement.

2. `elif`: elif is short for "else if." An elif clause is used to check for an additional condition if the conditions in the previous clauses in the if statement evaluate to False. As you can see in the example, you can have multiple elif blocks to handle different situations.

3. `else`: Last is the else clause, which must come at the end of an if statement if used. This clause doesn't require a condition. The code in an else block is run if all conditions above that in the if statement evaluate to False.

For example:

In [46]:
```
season = input('What season are we in? ')

if season == 'spring':
    print('plant the garden!')
elif season == 'summer':
    print('water the garden!')
elif season == 'fall':
    print('harvest the garden!')
elif season == 'winter':
    print('stay indoors!')
else:
    print('unrecognized season')
```

```
What season are we in? fall
harvest the garden!
```

In [11]:
```
## Answer To Quiz 1
nairobi_population, nairobi_area = 864816, 231.89
mombasa_population, mombasa_area = 6453682, 486.5
```

```
#nairobi_pop_density = nairobi_population/nairobi_area
#mombasa_pop_density = mombasa_population/mombasa_area

# Write code that prints True if Nairobi is denser than Mombasa, and False otherwise

nairobi_population_density = nairobi_population / nairobi_area
mombasa_population_density = mombasa_population / mombasa_area

print(nairobi_population_density >  mombasa_population_density)
```

```
False
```

In [ ]: