2/14/2023

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

Python Data Types and Operators

In todays class, you will learn how to write basic Python statements using building blocks like the print statement, variables, and different data types.

Here are the topics you'll learn about:

1. Data Types: Integers, Floats, Booleans, Strings
2. Operators: Arithmetic, Assignment, Comparison, Logical
3. Built-In Functions, Type Conversion
4. Whitespace and Style Guidelines
5. Print Statements print () - built-in function that displays input values as text in the output

Python Operators

Operators are used to perform operations on variables and values.

In the example below, we use the + operator to add together two values:

In [ ]:
```
# Basic Arithmetic:
print( 2 * 3 )       # Basic Arithmetic: +, -, /, *
print( 2**3 )        # Basic Arithmetic: +, -, /, * Exponentiation
print( 10 % 3 )      # Modulus Op. : returns remainder of 10/3
print( 1 + 2 * 3 )   # order of operations
print(10 / 3.0)      # int's and doubles : Division
print(10 // 3)       #Floor division
```

Quiz 1: Average Electricity Bill

It's time to try a calculation in Python!

My electricity bills for the last three months have been `$23, $32 and $64`. What is the average monthly electricity bill over the three month period? Write an expression to calculate the mean, and use `print()` to view the result?.

Variables

Variables are containers for storing data values.

Creating Variables

Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

In [2]:
```
#Example
x = 5
y = "John"
print(x)
print(y)
```

```
5
John
```

note:

Variables do not need to be declared with any particular type, and can even change type after they have been set.

In [1]:
```
x = 5 # x is of type int
x = 'MANU' # x is now of type str
print(x)
```

```
MANU
```

Integers and Floats

There are two Python data types that could be used for numeric values:

`int` - for integer values `float` - for decimal or floating point values You can create a value that follows the data type by using the following syntax:

In [7]:
```
x = int(4.7)   # x is now an integer 4
y = float(4)   # y is now a float of 4.0
```

In [8]:
```
print(type(x))
print(type(y))
```

```
<class 'int'>
<class 'float'>
```

Quiz 2: What happens if you divide by zero in Python?

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

In [15]:
```
# TODO: Fix this string!
quote = 'Whether you think you can, or you think you can't--you're right.'
print(quote)
```

```
  File "C:\Users\Isaac\AppData\Local\Temp\ipykernel_2708\3726383911.py", line 2
    quote = 'Whether you think you can, or you think you can't--you're right.'
                                                              ^
SyntaxError: invalid syntax
```

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 [42]:
```
print(elements["helium"])
```

```
2
```

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

```
2
```

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

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

In [ ]:

In [ ]:

In [ ]:

In [ ]:
```
# Answer To QUIZ 1

print((23+32+64)/3)

print((23+32+64)//3)


bills = 23 + 32 + 64
mean = bills // 3

print(mean)
```

Answer To Quiz 2

```
Traceback (most recent call last):
  File "/tmp/vmuser_tnryxwdmhw/quiz.py", line 1, in <module>
    print(5/0)

ZeroDivisionError: division by zero
```

Traceback means "What was the programming doing when it broke"! This part is usually less helpful than the very last line of your error. Though you can dig through the rest of the error, looking at just the final line ZeroDivisionError, and the message says we divided by zero. Python is enforcing the rules of arithmetic!

In general, there are two types of errors to look out for

> Exceptions Syntax

An Exception is a problem that occurs when the code is running, but a '`Syntax Error`' is a problem detected when Python checks the code before it runs it. For more information, see the Python tutorial page on Errors and Exceptions.https://docs.python.org/3/tutorial/errors.html

In [11]:
```
## Answer To Quiz 3
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 [ ]:
```

```