Programming_Lesson_27-09-20

September 27, 2020

1 Hello, welcome to today's class

1.1 Let's head first into some more recaps:

Concepts ### #1. Numberical Operators

Operator	Name	Description
`a + b`	Addition	Sum of `a` and `b`
`a - b`	Subtraction	Difference of `a` and `b`
`a * b`	Multiplication	Product of `a` and `b`
`a / b`	True division	Quotient of `a` and `b`
`a // b`	Floor division	Quotient of `a` and `b`, removing fractional parts
`a % b`	Modulus	Integer remainder after division of `a` by `b`
`a ** b`	Exponentiation	`a` raised to the power of `b`
`-a`	Negation	The negative of `a`

Source: Kaggle python tutorial document

1.1.1 #2. Little more about functions

Just to discuss from last class, and add a little more. Implemention of functions happens in following steps. - ———— 1. Function defintion:
`def function_name(arguements):`
2. Arguments can be none, default or passed by the caller.
`def function_name():` `def function_name(a, b = 3):` `def function_name(a, b):` 3. Functions can also have a return statement which also be none or some variable(s).
`return` `return a` `return a, b, c` 4. Functions are called when they are needed, and are run by calling their name and depends on how they are defined. `function_name()` `function_name(a)` `function_name(a, b)` 5. Remember to use docstrings to add documentation for your function, and you can access the documentation for built in function using `help()` function.
`help(function_name)`

1.1.2 #3. Booleans

Booleans are expression or variables which when evaluated give out the value of either `True` or `False`.

```
In [3]: # Short examples of variables
        Sunny = True
        Rainy = False
        print(Sunny, Rainy)
```

```
True False
```

To write boolean expression we use comparison operators. ### **Comparison Operators** | Operation | Description | |————|————-| | `a == b` | a equal to b | | `a < b` | a less than b | | `a <= b` | a less than or equal to b | | `a != b` | a not equal to b | | `a > b` | a greater than b | | `a >= b` | a greater than or equal to b |

Table again inspired from kaggle tutorial

```
In [5]: # Examples of Boolean expressions
        6 == 4
```

```
Out[5]: False
```

```
In [6]: 'Rainy' == 'Sunny'
```

```
Out[6]: False
```

```
In [7]: 7 > 3
```

```
Out[7]: True
```

```
In [8]: 6.4 > 6
```

```
Out[8]: True
```

```
In [9]: 'string' > 8
```

```
        ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)

        <ipython-input-9-4942589c1f70> in <module>
    ----> 1 'string' > 8


        TypeError: '>' not supported between instances of 'str' and 'int'
```

Note that the error in last block is because the comparion is between two incompatible variables.
The above exression, it is also possible to create more compound expressions using `or`, `not`, `and` operators. These are boolean operators.

Similar to their english counterparts, in python their usage serve the same purpose.

```
In [10]: # Blocks showing usage of or, and, not operators

In [11]: 5 == 6 or 7 > 4

Out[11]: True

In [14]: 'rainy' != 'sunny' and 6<=10

Out[14]: True

In [15]: not True

Out[15]: False

In [16]: not 5 == 6 or 7 > 4

Out[16]: True
```

Not the discrepancy in the last code block. This is why it is sometimes import to use parentheses to create the intended logical order.

```
In [17]: not (5 == 6 or 7 > 4)

Out[17]: False
```

A short programming break to create function which checks whether a number is less than 0 or not?(Use Boolean)

(Hint: Remember we used boolean expressions in conditionals such as if-else statements and loops in the last class.)

1.1.3 #4. Type conversions possible

Just as there are different types are different type conversions possible. Examples of them till now are

```
int()
float()
str()
bool()
```

```
In [20]: # Integer to Boolean conversion
         int(True)

Out[20]: 1

In [21]: int(False)

Out[21]: 0
```

```
In [22]: bool(0)
```

```
Out[22]: False
```

```
In [25]: # Boolean of any number apart from 0 is True
         bool(-56)
```

```
Out[25]: True
```

Now let's look at some of data structures in python such as list and tuple.(*Remember: Type conversion is also possible for them*)

Let's	do a quick exercise here to illustrate a feature of python language.
"cha	llenge" : Swap the value of two variables.

1.1.4 #5. Lists

Lists as the name suggests are used for storing multiple values in a container. An example would be

```
In [26]: days = ['Maandag', 'Dinsdag', 'Woensdag', 'Donderdag', 'Vrijdag', 'Zaterdag', 'Zondag
```

Values within a list are accessed by index, whcih is the positional arguement of list.

```
In [27]: days[2]
```

```
Out[27]: 'Woensdag'
```

In python indexing starts with 0. There are multiple ways to access values using an index or indices. The foolowing code bloacks illustrate some of them.

```
In [29]: days[-1]
         # '-' starts referencing the values from the last position in the list.
```

```
Out[29]: 'Zondag'
```

```
In [32]: # A list can be sliced to obtain a sublist of values using ':'.
         print(days[1:5])
```

```
['Dinsdag', 'Woensdag', 'Donderdag', 'Vrijdag']
```

```
In [33]: print(days[:5])
```

```
['Maandag', 'Dinsdag', 'Woensdag', 'Donderdag', 'Vrijdag']
```

```
In [34]: print(days[1:])
```

```
['Dinsdag', 'Woensdag', 'Donderdag', 'Vrijdag', 'Zaterdag', 'Zondag']
```

```
In [35]: print(days[:]) # This notation will come in handy when passing lists to a function.
```

```
['Maandag', 'Dinsdag', 'Woensdag', 'Donderdag', 'Vrijdag', 'Zaterdag', 'Zondag']
```

We can change the value within a list using an index as well.

```
In [37]: days[2] = "Wednesday"
         print(days)
```

```
['Maandag', 'Dinsdag', 'Wednesday', 'Donderdag', 'Vrijdag', 'Zaterdag', 'Zondag']
```

There some python functions that help with lists

```
In [38]: len(days)
         # This gives the number of elements within the list
```

```
Out[38]: 7
```

```
In [39]: sorted(days)
         # Sorts the elements within the list in alphabetical order
```

```
Out[39]: ['Dinsdag',
          'Donderdag',
          'Maandag',
          'Vrijdag',
          'Wednesday',
          'Zaterdag',
          'Zondag']
```

Recall: Everything is python is an object. So lists are two. And objects contain extras known as attributes and methods(another name for such functions).

Let's have a look at what they are.

```
In [40]: help(days)
```

```
Help on list object:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty list.
 |  The argument must be an iterable if specified.
 |
```

```
|  Methods defined here:
|
|  __add__(self, value, /)
|      Return self+value.
|
|  __contains__(self, key, /)
|      Return key in self.
|
|  __delitem__(self, key, /)
|      Delete self[key].
|
|  __eq__(self, value, /)
|      Return self==value.
|
|  __ge__(self, value, /)
|      Return self>=value.
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __getitem__(...)
|      x.__getitem__(y) <==> x[y]
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __iadd__(self, value, /)
|      Implement self+=value.
|
|  __imul__(self, value, /)
|      Implement self*=value.
|
|  __init__(self, /, *args, **kwargs)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mul__(self, value, /)
```

```
|      Return self*value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __reversed__(self, /)
|      Return a reverse iterator over the list.
|
|  __rmul__(self, value, /)
|      Return value*self.
|
|  __setitem__(self, key, value, /)
|      Set self[key] to value.
|
|  __sizeof__(self, /)
|      Return the size of the list in memory, in bytes.
|
|  append(self, object, /)
|      Append object to the end of the list.
|
|  clear(self, /)
|      Remove all items from list.
|
|  copy(self, /)
|      Return a shallow copy of the list.
|
|  count(self, value, /)
|      Return number of occurrences of value.
|
|  extend(self, iterable, /)
|      Extend list by appending elements from the iterable.
|
|  index(self, value, start=0, stop=9223372036854775807, /)
|      Return first index of value.
|
|      Raises ValueError if the value is not present.
|
|  insert(self, index, object, /)
|      Insert object before index.
|
|  pop(self, index=-1, /)
|      Remove and return item at index (default last).
|
|      Raises IndexError if list is empty or index is out of range.
|
```

```
 |  remove(self, value, /)
 |      Remove first occurrence of value.
 |
 |      Raises ValueError if the value is not present.
 |
 |  reverse(self, /)
 |      Reverse *IN PLACE*.
 |
 |  sort(self, /, *, key=None, reverse=False)
 |      Stable sort *IN PLACE*.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __hash__ = None
```

```
In [ ]: # some useful methods
        days.append("8th day") # This adds a function to the end of the list
        days
```

```
In [45]: # some useful methods
         days.remove("8th day") # As can be guessed, the list removes the occurrence of mentio
         days
```

```
Out[45]: ['Maandag',
          'Dinsdag',
          'Wednesday',
          'Donderdag',
          'Vrijdag',
          'Zaterdag',
          'Zondag']
```

One last point to add about lists are that they can be nested, one list inside another. They acessed with following format in syntax

```
list[indx1][]indx2]...[indxn]
```

1.1.5 #6. Tuples

Tuples are just like list, except they are defined with `()` instead of `[]`. The important feature of tuples is that there are immutable. Let's look at an example.

```
In [46]: Oceans = ("Indian", "Pacific", "Arctic", "Atlantic")

In [47]: Oceans[1] = "Arabian"


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-47-8436116d3ba7> in <module>
    ----> 1 Oceans[1] = "Arabian"


        TypeError: 'tuple' object does not support item assignment
```

Tuples are useful when returning multiple values from a function. An example would be.

```
In [48]: def division(num, div):
             '''
             Gives ou
             '''
             return (num//div, num%div)

         q,r = division(8,5)
         print(q,r)
```

```
1 3
```

in operator `in` is an operator which is also quite handy when it comes to lists and tuples. If you recall they also come to aid in `for` loop definitions. `in` creates a boolean expression checking for a value within a list or tuple.

```
In [50]: 'Maandag' in days

Out[50]: True
```

Concept revison: Loops Remember the loops we learnt about last class. `for` loops are generally iterated over lists or tuples. The following code gives a small illustration.

```
In [9]: for i in [1, 2, 3, 4, 5]:
            print(i, end=' ')
```

```
1 2 3 4 5
```

There is also an interesting feature in python known as list comprehensions which is a more succinct way for creating lists. The following code shows an example

```
In [11]: numbers = range(1,10)

         # The following are list comprehensions
         evens = [n for n in numbers if n%2==0]
         odds = [n for n in numbers if n%2!=0]

         print("Evens:", evens)
         print("Odds:", odds)
```

```
Evens: [2, 4, 6, 8]
Odds: [1, 3, 5, 7, 9]
```

1.1.6 Time for a quick challenge:

Let's make a quick program: Krish likes to eat fruit and nuts. At the moment his breakfast routine consists of apples, cashews and yogurt. - ——- ——— Write function that checks whether he has all three ingredients in his breakfast and returns 'best' if so. If only two of the ingredients are present, returns 'good', and returns 'not enough' otherwise. ———————– –

```
In [53]: import random
         def givebreakfast(ingredients):
             '''
             Returns a potential breakfast based on ingredients
             '''
             n = random.randint()
             return random.sample(ingredients, 2)
         breakfast = givebreakfast(['apples', 'cashews', 'yogurt'])

         def rateBreakfast(breakfast):
             """
             See the question description
             """
             pass
```

```
['apples', 'yogurt']
```

1.1.7 #7. Dictionaries

Let's talk about one last builtin data structure before moving on to external libraries. Dictionaries are useful in storing paired data values. Unlike, tuples they are mutable. ——- ———— The following code blocks show definition and usage of dictionaries.

```
In [1]: # dictionaries are defined using {} symbols
        # The below dictionary time in minutes for different daily routines
        routines_time = {}

        #Every dictionary has a pair of keys and values
```

```
routines_time["sleeping"] = 20
routines_time["walking"] = 60
routines_time["studying"] = 120

print(routines_time)
```

```
{'sleeping': 20, 'walking': 60, 'studying': 120}
```

Similar to lists, and tuples, the values in dictionary can be accessed by keys. Addionally, dictionaries too have a list of useful methods and attributes

```
In [2]: help(routines_time)
```

```
Help on dict object:

class dict(object)
 |  dict() -> new empty dictionary
 |  dict(mapping) -> new dictionary initialized from a mapping object's
 |      (key, value) pairs
 |  dict(iterable) -> new dictionary initialized as if via:
 |      d = {}
 |      for k, v in iterable:
 |          d[k] = v
 |  dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |      in the keyword argument list.  For example:  dict(one=1, two=2)
 |
 |  Methods defined here:
 |
 |  __contains__(self, key, /)
 |      True if the dictionary has the specified key, else False.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(self, value, /)
 |      Return self>value.
```

```
|
|  __init__(self, /, *args, **kwargs)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __setitem__(self, key, value, /)
|      Set self[key] to value.
|
|  __sizeof__(...)
|      D.__sizeof__() -> size of D in memory, in bytes
|
|  clear(...)
|      D.clear() -> None.  Remove all items from D.
|
|  copy(...)
|      D.copy() -> a shallow copy of D
|
|  get(self, key, default=None, /)
|      Return the value for key if key is in the dictionary, else default.
|
|  items(...)
|      D.items() -> a set-like object providing a view on D's items
|
|  keys(...)
|      D.keys() -> a set-like object providing a view on D's keys
|
|  pop(...)
|      D.pop(k[,d]) -> v, remove specified key and return the corresponding value.
|      If key is not found, d is returned if given, otherwise KeyError is raised
|
|  popitem(...)
```

```
 |      D.popitem() -> (k, v), remove and return some (key, value) pair as a
 |      2-tuple; but raise KeyError if D is empty.
 |
 |  setdefault(self, key, default=None, /)
 |      Insert key with a value of default if key is not in the dictionary.
 |
 |      Return the value for key if key is in the dictionary, else default.
 |
 |  update(...)
 |      D.update([E, ]**F) -> None.  Update D from dict/iterable E and F.
 |      If E is present and has a .keys() method, then does:  for k in E: D[k] = E[k]
 |      If E is present and lacks a .keys() method, then does:  for k, v in E: D[k] = v
 |      In either case, this is followed by: for k in F:  D[k] = F[k]
 |
 |  values(...)
 |      D.values() -> an object providing a view on D's values
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  fromkeys(iterable, value=None, /) from builtins.type
 |      Create a new dictionary with keys from iterable and values set to value.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __hash__ = None
```

```
In [8]: # examples
        values = routines_time.values()
        print(values)
        keys = routines_time.keys()
        print(keys)
```

```
dict_values([20, 60, 120])
dict_keys(['sleeping', 'walking', 'studying'])
```

1.2 Revision challenges:

1. Create a dictionary that takes keys different concepts learnt till now in python, and values as their level of clarity in you between 0 to 5.
2. Create a function that helps you calculate the amount of time you need for revision or practice in each. Say 1 unit means 20 minutes of practise or revision or reading. So 5 points meaning 0 minutes needed, 4 points being 20 minutes, 3 pts being 40 minutes and so on.

1.2.1 Good luck! :) <3

In next class we will look at external libraries, and file handling