00_introduction

January 18, 2025

Check online documentation on jupyter here

0.0.1 printing hacks

the function `print` allows also more powerful way of outputting numbers:

[ ]:
```
print("Art: %5d, Price per unit: %8.1f"% (453, 59.058))
```

0.0.2 multiple assignment

[9]:
```
a, b = 5, 1 # a python trick to assign values to more than one variable
print(a,b)
print(a+b)

c, d = 5, "1"
print(c,d)
print(c+int(d)) # this will give an error because you can't add an integer to a␣
 ↪string, unless you cast the string to an integer
```

```
5 1
6
5 1
6
```

[5]:
```
x = y = z = 7 # multiple assignment
print(x, y, z)
```

```
7 7 7
```

0.0.3 set values from command line

[8]:
```
# can set values from command line as well (rarely used)
x = input("set the value of x")
print(x)
```

```
5
```

0.0.4 Pointers in Python?

Pointers are widely used in C and C++. Essentially, they are variables that hold the memory address of another variable. Are there pointers in python? Essentially no. Pointers go against the Zen of Python:

Pointers encourage implicit changes rather than explicit. Often, they are complex instead of simple, especially for beginners. Even worse, they beg for ways to shoot yourself in the foot, or do something really dangerous like read from a section of memory you were not supposed to. Python tends to try to abstract away implementation details like memory addresses from its users. Python often focuses on usability instead of speed. As a result, pointers in Python doesn't really make sense.

It's all about two basics python concepts: 1. Mutable vs Immutable objects 2. Variable/Name in python

Mutable objects can be changed, immutable cannot. I.e. when a new value is assigned to a given immutable "variable", a new object is in reality created. **This as "variable" in python are actually just names bound to objects (PyObject).**

Everything in Python is an object!

For more details about all this refer e.g. to this review

0.1 Conditional Statements

Mind the indentation!

try/except: a very important and powerful type of conditional expression, use it and use it with care

[ ]:
```
a = 1
try:
    b = a + 2
    print (b)
except:
    print(a, " is not a number")
```

1 Functions

Notice that the function does not specify the types of the arguments, like you would see in statically typed languages. This is both useful and dangerous. Use the try/except construction to make it safe

[11]:
```
def my_function(a, b = 2):
    try:
        result = a + 2 * b
    except:
        print(a, " is not a number")
    return result

my_function('c',0)
```

```
c  is not a number
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Cell In[11], line 8
      5         print(a, " is not a number")
      6     return result
----> 8 my_function('c',0)

Cell In[11], line 6, in my_function(a, b)
      4 except:
      5     print(a, " is not a number")
----> 6 return result

UnboundLocalError: cannot access local variable 'result' where it is not␣
 ↪associated with a value
```

function can edit "global" variables as well, i.e. variables that are declared outside the function scope. To do so the statement `global` is used

[13]:
```
x = "awesome"

def myfunc():
  global x
  x = "fantastic"

myfunc()

print("Python is " + x)
```

```
Python is fantastic
```

2 Lists, Tuples, Dictionaries

2.1 Lists

Lists are exactly as the name implies. They are lists of objects. The objects can be any data type (including lists), and it is allowed to mix data types. In this way they are much more flexible than arrays. It is possible to append, delete, insert and count elements and to sort, reverse, etc. the list.

[ ]:
```
a_list = [1, 2, 3, "this is a string", 5.3]
b_list = ["A", "B", "F", "G", "d", "x", "c", a_list, 3]
print(b_list)
```

Manipulations of list is rather intuitive:

`list[start:stop:step]`

[21]:
```
a = [7, 5, 3, 4, 10]
print(a[0])
print(a[-1])
print(a[2:4])
print(a[:3])
print(a[3:])
print(a[-3:])
print(a[3:len(a)])
print(a[1::3])
```

```
7
10
[3, 4]
[7, 5, 3]
[4, 10]
[3, 4, 10]
[4, 10]
[5, 10]
```

Operations on lists are also straightforward. Note that some of them do not modify the list permanently.

[22]:
```
a.insert(0, 0) # position, value
a.reverse()
i = a.pop()
print(i)
print(a)
```

```
[0, 7, 5, 3, 4, 10]
[0, 7, 5, 3, 4, 10, 8]
[8, 10, 4, 3, 5, 7, 0]
[0, 3, 4, 5, 7, 8, 10]
10
[0, 3, 4, 5, 7, 8]
```

strings are lists and feature all operations permitted on lists, comprehensions as well

[23]:
```
first_sentence = "It was a dark and stormy night."
characters = [x for x in first_sentence]
print(characters)
```

```
['I', 't', ' ', 'w', 'a', 's', ' ', 'a', ' ', 'd', 'a', 'r', 'k', ' ', 'a', 'n',
'd', ' ', 's', 't', 'o', 'r', 'm', 'y', ' ', 'n', 'i', 'g', 'h', 't', '.']
```

the opposite is also possible, but in a different way:

[24]:
```
second_sentence = ''.join(characters)
print(second_sentence)
```

```
It was a dark and stormy night.
```

2.1.1 Strings and String Handling

One of the most important features of Python is its powerful and easy handling of strings. Defining strings is simple enough in most languages. But in Python, it is easy to search and replace, convert cases, concatenate, or access elements. We'll discuss a few of these here. For a complete list, see this tutorial

[25]:
```
a = "A string of characters, with newline \n CAPITALS, etc."
print(a)
b = 5.0
newstring = a + "\n We can format strings for printing %.2f"
print(newstring % b)
```

```
A string of characters, with newline
 CAPITALS, etc.
A string of characters, with newline
 CAPITALS, etc.
 We can format strings for printing 5.00
```

Operations are easy (remember strings are lists!)

[26]:
```
a = "ABC DEFG"
print(a[1:3])
print(a[0:5])
```

```
BC
ABC D
```

[27]:
```
a = "ABC defg"
print(a.lower())
print(a.upper())
print(a.find('d'))
print(a.replace('de', 'a'))
print(a)
b = a.replace('def', 'aaa')
print(b)
b = b.replace('a', 'c')
print(b)
b.count('c')
```

```
abc defg
ABC DEFG
4
ABC afg
ABC defg
ABC aaag
ABC cccg
```

[27]:
```
3
```

[28]:
```
print("ABC defg".lower())
```

```
abc defg
```

2.2 Tuples

Tuples are like lists with one very important difference. Tuples are not changeable.

[29]:
```
a = (1, 2, 3, 4)
print(a)
a[1] = 2
```

```
(1, 2, 3, 4)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[29], line 3
      1 a = (1, 2, 3, 4)
      2 print(a)
----> 3 a[1] = 2

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

2.3 Dictionaries

Dictionaries are of paramount importance and a major asset of python. They are unordered, keyed lists. Lists are ordered, and the index may be viewed as a key.

[ ]:
```
a = {'anItem' : "A", 'anotherItem' : ["a,bc"], 3 : "C", 'afourthItem' : 7} #␣
 ↪dictionary example
print(a['anItem'])
```

[ ]:
```
for i in a: print(i, a[i])
```

[ ]:
```
print("Keys:", a.keys())
print("Values:", a.values())
```

2.4 Sets

Sets are used to store multiple items in a single variable, but differently from lists and dictionaries, they are unordered and do not support duplicates. In addition you cannot access items in a set by referring to an index or a key, but you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

[30]:
```
a = {"apple", 2, "cherry", 2}
print(a)
```

```
{'apple', 'cherry', 2}
```

[31]:
```
print(a[1])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[31], line 1
----> 1 print(a[1])

TypeError: 'set' object is not subscriptable
```

[32]:
```
for i in a: print(i)
```

```
apple
cherry
2
```

[35]:
```
a.add("3")
print(a)
a.update({7, "banana"})
print(a)
print(3 in a)
print("banana" in a)
```

```
{'cherry', 2, 'banana', 7, 'apple', '3'}
{'cherry', 2, 'banana', 7, 'apple', '3'}
False
True
```