```
# Tutorial week 6
# author: B. Schoen-Phelan
# date: Oct 2020
# topic: data structures and comprehensions
# examples from RealPython
#
https://realpython.com/python-data-structures/#array-da
ta-structures
# and
#
https://realpython.com/list-comprehension-python/#using
-set-and-dictionary-comprehensions
# accessed Oct 2020

# 1: Lists

# my_array = ["one", "two", "three"]
# print(my_array[0])

# nice print
# print(my_array)

# Lists are mutable:
# my_array[1] = "hello"
# print(my_array)

# del my_array[1]
# print(my_array)

# Lists can hold arbitrary data types:
# my_array.append(41)
# print(my_array)

# 1.2 comprehensions

# squares = []
# for i in range(10):
#     squares.append(i * i)
#
```

```
# print(squares)

# with a comprehension
# squares = [i * i for i in range(10)]
# print(squares)

# listOfWords = ["this","is","a","list","of","words"]
#
# new_list = [ word[0] for word in listOfWords ]
#
# print (new_list)

# random_string = "Hello 12345 World"
# new_list = [x for x in random_string if x.isdigit()]
# print(new_list)

# new_list = [i.lower() for i in ["A","B","C"]]
# print(new_list)

# my_file = open("simpleTextFile.txt", "r")
#
# new_list = [i for i in my_file if "line4" in i]
#
# print(new_list)

# new_list = [x+y for x in [10,30,50] for y in
[20,40,60]]
# print(new_list)

# def fourth(x):
#   return x**4
#
# print(fourth(2))

# new_list = [fourth(i) for i in range(10)]
# new_list = [fourth(i) for i in range(10) if i%2==0]
# print(new_list)

def is_narcissistic_num(num):
  return num == sum([int(x) ** len(str(num)) for x in
str(num)])

print(is_narcissistic_num(153))


#2: tuples
# my_array = ("one", "two", "three")
# print(my_array[0])

# easy printing
# print(my_array)

# Tuples are immutable:
# my_array[1] = "hello"  #causes an TypeError message
# del my_array[1] #causes an TypeError message

# Tuples can hold arbitrary data types:
# (Adding elements creates a copy of the tuple)
# print(id(my_array)) #original: 140527352248512
# my_array = my_array + (41,)
# print(my_array) # prints ('one', 'two', 'three', 41)
# print(id(my_array)) #copy of same name:
140527363259184

# 3 namedTuples
# from collections import namedtuple
#
# Car = namedtuple("Car", "color mileage automatic")
# my_car = Car("red", 29812.3, False)

# Instances have a nice repr:
# print(my_car)

# Accessing fields:
# print(my_car.mileage)

# Fields are immtuable:
# my_car.mileage = 12 #causes an error
```

```
#
# my_car.windshield = "broken" #causes an error


# 4 improved namedTuples
# from typing import NamedTuple
#
# class Car(NamedTuple):
#     color: str
#     mileage: float
#     automatic: bool
#
# my_car = Car("red", 3812.4, True)

# print(my_car)
# print(my_car.mileage)

# Fields are immutable:
# my_car.mileage = 12 #causes an error
# my_car.windshield = "broken" #causes an error



# Type annotations are not enforced without
# a separate type checking tool like mypy:
# my_second_car = Car("red", "NOT_A_FLOAT", 99)
# print(my_second_car)

# 5: arrays

# import array
# my_array = array.array("f", (1.0, 1.5, 2.0, 2.5))
# print(my_array[1])

# nice print
# print(my_array)

# Arrays are mutable:
# my_array[1] = 41.0
# print(my_array)
#
# del my_array[1]
# print(my_array)
#
# my_array.append(42.0)
# print(my_array)


# Arrays are "typed":
# my_array[1] = "hello" #causes an error message

# 6: string arrays
# my_array = "abcd"
# print(my_array[1])

# nice print
# print(my_array)

# Strings are immutable:
# my_array[1] = "e" #causes an error

# del my_array[1] #causes an error

# Strings can be unpacked into a list to
# get a mutable representation:
# my_array_lst = list("abcd")
# print(my_array_lst)
#
# from_lst_to_str = "".join(list("abcd"))

# Strings are recursive data structures:
# print(type(from_lst_to_str))
#
# print(type(from_lst_to_str[0]))

# 7: byte arrays
# my_array = bytes((0, 1, 2, 3))
# print(my_array[1])
# Bytes literals have their own syntax:
# print(my_array)
#
# my_array = b"\x00\x01\x02\x03"
# print(my_array)

# Only valid `bytes` are allowed:
# print(bytes((0, 300))) #causes an out of range error

# Bytes are immutable:
# my_array[1] = 41 #causes an error
# del my_array[1] #causes an error

# 8: bytearrays
# my_array = bytearray((0, 1, 2, 3))
# print(my_array[1])

# The bytearray repr:
# print(my_array)

# Bytearrays are mutable:
# my_array[1] = 41
# print(my_array)
#
# print(my_array[1])

# Bytearrays can grow and shrink in size:
# del my_array[1]
# print(my_array)
# my_array.append(42)
# print(my_array)

# Bytearrays can only hold `bytes`
# (integers in the range 0 <= x <= 255)
# my_array[1] = "hello" #causes an error
# my_array[1] = 277 #causes an error

# Bytearrays can be converted back into bytes objects:
```

```
# (This will copy the data)
# print(type(my_array))
# my_array = bytes(my_array)
# print(type(my_array))

# 9: simpleNamespace
# from types import SimpleNamespace
# my_car = SimpleNamespace(color="red", mileage=3812.4,
automatic=True)
# print(my_car)

# Instances support attribute access and are mutable:
# my_car.mileage = 12
# my_car.windshield = "broken"
# del my_car.automatic
# print(my_car)

# 10: struct
# from struct import Struct
# my_struct = Struct("i?f")
# my_data = my_struct.pack(2, False, 41.0) #packed
according to a given format
#                                           #returns a
bytes object

# All you get is a blob of data:
# print(my_data)


# Data blobs can be unpacked again:
# print(my_struct.unpack(my_data))

# 11: class example
# class Car:
#     def __init__(self, color, mileage, automatic):
#         self.color = color
#         self.mileage = mileage
#         self.automatic = automatic
#
```

```
# my_first_car = Car("red", 22812.4, True)
# my_second_car = Car("grey", 40357.7, False)

# Get the mileage
# print(my_second_car.mileage)


# Classes are mutable:
# my_second_car.mileage = 12
# print(my_second_car.mileage)
# my_second_car.windshield = "broken"
# print(my_second_car)

# String representation is not very useful
# print(my_first_car)


# 12: data class example
# from dataclasses import dataclass
# @dataclass
# class Car:
#     color: str
#     mileage: float
#     automatic: bool
#
# my_first_car = Car("red", 3812.4, True)

# Instances have a nice repr:
# print(my_first_car)


# Accessing fields:
# print(my_first_car.mileage)

# Fields are mutable:
# my_first_car.mileage = 12
# my_first_car.windshield = "broken"
# print(my_first_car.mileage)
# Type annotations are not enforced without
# a separate type checking tool like mypy:
# my_second_car = Car("red", "NOT_A_FLOAT", 99)
# print(my_second_car)


# 13: set
# vowels = {"a", "e", "i", "o", "u"}
# print("e" in vowels)
#
#
# letters = set("bianca")
# print(letters.intersection(vowels))
#
# vowels.add("x")
# print(vowels)
# print(len(vowels))

# 14: frozenSet
# vowels = frozenset({"a", "e", "i", "o", "u"})
# vowels.add("p")
```