Week02

August 31, 2024

1 Collections

Python has 4 built-in data types to store collections of data. - List - Tuple - Set - Dictionary

Useful reference: https://docs.python.org/3/tutorial/datastructures.html

1.1 List

- A list is a collection of elements in a particular **order**.
- A list can include the letters of the alphabet, the digits from 0–9, etc. Elements in a list can have **different types**.
- `[]` indicates a list, and individual elements in the list are separated by commas.
- A list allows **duplicates**.

```
[ ]: [1,2,3]
```

```
[ ]: [1, 2, 3]
```

```
[ ]: x = ["apple", "banana", "orange"]
     x
```

```
[ ]: ['apple', 'banana', 'orange']
```

```
[ ]: x = [1, 2.379, "apple"] # you can have multiple data types!!!!!
     x
```

```
[ ]: [1, 2.379, 'apple']
```

```
[ ]: # list of lists
     [ [1, 2], ["apple", "bannana"] ]
```

```
[ ]: [[1, 2], ['apple', 'bannana']]
```

1.1.1 Create a numerical list using range()

```
[ ]: # create a list from 0 ... 100
     s = list(range(0,100))
     s
```

```
[ ]: [0,
      1,
      2,
      3,
      4,
      5,
      6,
      7,
      8,
      9,
      10,
      11,
      12,
      13,
      14,
      15,
      16,
      17,
      18,
      19,
      20,
      21,
      22,
      23,
      24,
      25,
      26,
      27,
      28,
      29,
      30,
      31,
      32,
      33,
      34,
      35,
      36,
      37,
      38,
      39,
      40,
      41,
      42,
      43,
      44,
      45,
      46,
47,
48,
49,
50,
51,
52,
53,
54,
55,
56,
57,
58,
59,
60,
61,
62,
63,
64,
65,
66,
67,
68,
69,
70,
71,
72,
73,
74,
75,
76,
77,
78,
79,
80,
81,
82,
83,
84,
85,
86,
87,
88,
89,
90,
91,
92,
93,
 94,
 95,
 96,
 97,
 98,
 99]
```

```
[ ]: # length of a list is the number of elements in a list
     len(s)
```

```
[ ]: 100
```

```
[ ]: y = [ [1, 2], ["apple", "bannana"] ]
     len(y)
```

```
[ ]: 2
```

1.1.2 Indexing elements in a list

Index positions start at 0, not 1. From left to right, 0, 1, 2, …. From right to left, -1, -2, …

```
[ ]: y = [ [1, 2], ["apple", "bannana"] ]
     y[0], y[1]
```

```
[ ]: ([1, 2], ['apple', 'bannana'])
```

1.1.3 Slicing a list

Specify the index of the first and last elements you want to work with. Python stops one item before the second index you specify.

```
[ ]: s = list(range(1,100))
     s[19:60]
```

```
[ ]: [20,
      21,
      22,
      23,
      24,
      25,
      26,
      27,
      28,
      29,
      30,
      31,
      32,
      33,
 34,
 35,
 36,
 37,
 38,
 39,
 40,
 41,
 42,
 43,
 44,
 45,
 46,
 47,
 48,
 49,
 50,
 51,
 52,
 53,
 54,
 55,
 56,
 57,
 58,
 59,
 60]
```

1.1.4 Changing elements in a list

Use the name of the list followed by the index of the element you want to change, and then provide the new value you want that item to have.

```
[ ]: x = [1, 2.379, "apple"]
     x[2] = "banana"
     x
```

```
[ ]: [1, 2.379, 'banana']
```

1.1.5 Adding elements to a list

1. `append()` elements to the end of a list
2. `insert()` a new element at a specified position. Need to specify the index and value of the new element.

```
[ ]: x.append("brat")
[ ]: x.insert(0, 1000)
     x
```

```
[ ]: [1000, 1, 2.379, 'banana', 'brat']
```

```
[ ]: x.insert(4, "apple") # insert AT the 4th position
     x
```

```
[ ]: [1000, 1, 2.379, 'banana', 'apple', 'brat']
```

1.1.6 Removing elements from a list

1. Use `del` to remove an element according to its position.
2. Use `pop()` to remove an element in a list at a specified position, and it will return the poped element. If no index is provided, pop out the last element.
3. Use `remove()` to remove an element by value. Note the `remove()` method deletes only the first occurrence of the value you specify.

```
[ ]: a = x.pop() # removes last value
     a, x
```

```
[ ]: ('brat', [1000, 1, 2.379, 'banana', 'apple'])
```

```
[ ]: a = x.pop(1) # argument removes specific value
     a, x
```

```
[ ]: (1, [1000, 2.379, 'banana', 'apple'])
```

```
[ ]: x = ["apple", "banana", "orange", "peach", "apple"]
```

```
[ ]: a = x.remove("apple") # only removes one
     a, x
```

```
[ ]: (None, ['banana', 'orange', 'peach', 'apple'])
```

```
[ ]: x = ["apple", "banana", "orange", "peach", "apple"]
     while "apple" in x:
         x.remove("apple")
     x
```

```
[ ]: ['banana', 'orange', 'peach']
```

1.1.7 Joining multiple lists

```
[ ]: a = list(range(5))
     b = list(range(10,15))
     c = a+b
     c
```

```
[ ]: [0, 1, 2, 3, 4, 10, 11, 12, 13, 14]
```

```
[ ]: a.extend(b)
     a
```

```
[ ]: [0, 1, 2, 3, 4, 10, 11, 12, 13, 14]
```

1.1.8 Ordering a list

1. Use `sort()` to sort a list permanently
2. Use `sorted()` to temporarily sort a list
3. Reverse the list by `reverse()` or slicing

```
[ ]: x = [67, 78, 0, 1, 44]
     x.sort()
     x
```

```
[ ]: [0, 1, 44, 67, 78]
```

```
[ ]: x = [67, 78, 0, 1, 44]
     sorted(x)
```

```
[ ]: [0, 1, 44, 67, 78]
```

```
[ ]: x
```

```
[ ]: [67, 78, 0, 1, 44]
```

```
[ ]: sorted(x, reverse=True)
```

```
[ ]: [78, 67, 44, 1, 0]
```

```
[ ]: x = [67, 78, 0, 1, 44]
     x.reverse() # sticky reverse
     x
```

```
[ ]: [44, 1, 0, 78, 67]
```

```
[ ]: x[::-1] # reverse it but not sticky
```

```
[ ]: [67, 78, 0, 1, 44]
```

```
[ ]: x[:3]
```

```
[ ]: [44, 1, 0]
```

1.2 Tuple

- A tuple is **immutable**. It can be viewed as immutable list.

- () indicates a tuple and indidual elements in the tuple are separated by commas.
- Elements in a tuple can be indexed, but tuple object does not support item assignment.

```
[ ]: x = (1, 3.14, "hello")
     x
```

```
[ ]: (1, 3.14, 'hello')
```

```
[ ]: x[0]
```

```
[ ]: 1
```

```
[ ]: x[0] = 100
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[86], line 1
----> 1 x[0] = 100

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

1.3 Set

- A set is an **unordered** collection with **no duplicate** elements.
- Basic uses include membership testing and eliminating duplicate entries.
- Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.
- Use `set()` or `{}` to create a set. Note: to create an empty set you have to use `set()`, not `{}`; the latter creates an empty dictionary.

```
[ ]: x = set()
     for i in range(10):
         x.add(i)
     x
```

```
[ ]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
[ ]: x = set([1,1,2,2,3,3,4]) # may be time consuming if list is long
     x
```

```
[ ]: {1, 2, 3, 4}
```

```
[ ]: y = {2,3,8}
```

```
[ ]: # Set operation
     x - y # removes contents of y from x
```

```
[ ]: {1, 4}
[ ]: x & y # returns common contents of x & y
```

```
[ ]: {2, 3}
```

```
[ ]: x or y
```

```
[ ]: {1, 2, 3, 4}
```

```
[ ]:
```

1.4 Dictionary

- A dictionary is a collection of **key-value** pairs.
- Use a key to access the value associated with that key.
- Keys in a dictionary should be unique.

1.4.1 Creating, indexing, modifying, adding, and removing

```
[ ]: x = {"name": "Pikachu", "Type": "Electric", "height": 40, "weight": 6}
     x
```

```
[ ]: # create from keys
     x = {}
     x['name'] = "Pikachu"
     x['type'] = "Electric"
     x['height'] = 40
     x['weight'] = 6
     x
```

```
[ ]: {'name': 'Pikachu', 'type': 'Electric', 'height': 40, 'weight': 6}
```

```
[ ]: x.keys()
```

```
[ ]: dict_keys(['name', 'type', 'height', 'weight'])
```

```
[ ]: x.values()
```

```
[ ]: dict_values(['Pikachu', 'Electric', 40, 6])
```

```
[ ]: x["name"]
```

```
[ ]: 'Pikachu'
```

```
[ ]: x["weight"] = 10
     x
```

```
[ ]: {'name': 'Pikachu', 'type': 'Electric', 'height': 40, 'weight': 10}
[ ]: x["attack"] = ["thunderbolt", "quick attack", "iron tail"]
```

```
[ ]: x = {}
     x["name"] = "Pikachu"
     x
```

```
[ ]: {'name': 'Pikachu'}
```

```
[ ]: x["type"] = "electric"
     x
```

```
[ ]: {'name': 'Pikachu', 'type': 'electric'}
```

```
[ ]: del x["type"]
     x
```

```
[ ]: {'name': 'Pikachu'}
```

```
[ ]: # nested dictionary
     pokemon_collection = {} # empty dictionary
     pokemon_collection["Pikachu"] = {"type": "electric", "height": 0.4, "weight":␣
       ↪6} # add pikachu to empty dictionary
     pokemon_collection
```

```
[ ]: {'Pikachu': {'type': 'electric', 'height': 0.4, 'weight': 6}}
```

```
[ ]: pokemon_collection["Eevee"] = {"type": "normal", "height": 0.3, "weight": 6.5}
     pokemon_collection
```

```
[ ]: {'Pikachu': {'type': 'electric', 'height': 0.4, 'weight': 6},
      'Eevee': {'type': 'normal', 'height': 0.3, 'weight': 6.5}}
```

```
[ ]:
```

1.4.2 Using get() to Access Values

```
[ ]: pokemon_collection["Bulbasaur"] # returns KeyError
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[14], line 1
----> 1 pokemon_collection["Bulbasaur"]

KeyError: 'Bulbasaur'
```

```
[ ]: pokemon_collection.get("Bulbasaur") # returns nothing
```

[ ]:

```
pokemon_collection.get("Pikachu")
```

[ ]:

```
{'type': 'electric', 'height': 0.4, 'weight': 6}
```

[ ]:

```
pokemon_collection["Pikachu"]
```

[ ]:

```
{'type': 'electric', 'height': 0.4, 'weight': 6}
```

2 Control Flow

- `if`, `if-else`, `if-elif-else`
- `for` loop, `while` loop
- `break` and `continue`
- `pass`
- More on iterations: list comprehension

Useful reference: https://docs.python.org/3/tutorial/controlflow.html 4.1-4.5

2.0.1 if statements

`if` statement evaluates whether a **conditional test** is `True` or `False`.
If a conditional test evaluates to True, Python executes the code following the if statement. If the test evaluates to False, Python ignores the code following the if statement.

[ ]:

```
if 1+1 == 2:
    print(":3")
```

```
:3
```

[ ]:

```
x = 1
if x < 0 :
    print("negative value")
else:
    print("positive value or zero")
```

```
positive value or zero
```

[ ]:

```
x = 1
if x < 0 :
    print("negative value")
elif x == 0:
    print("zero")
else:
    print("positive value")
```

```
positive value
```

[ ]:

```
# multiple elif
x = 85
if x >= 90:
    print("A")
elif x >= 80:
    print("B")
elif x >= 70:
    print("C")
else:
    print("F")
```

```
B
```

[ ]:

```
if x >= 90:
    print("A")
elif x >= 70: # changed order
    print("C")
elif x >= 80:
    print("B")
else:
    print("F")
```

```
C
```

[ ]:

```

```

[ ]:

```
fruits = ["apple", "banana", "orange", "cherry", "blueberry"]
```

[ ]:

```
x = "apple"
if x in fruits:
    print("available")
else:
    print("not available")
```

```
available
```

[ ]:

```

```

[ ]:

```
# multiple conditions
x, y = "apple", "pineapple"
if x in fruits and y in fruits:
    print(":)")
elif (x in fruits and y not in fruits) or (x not in fruits and y in fruits):
    print(":-|")
else:
    print(":( go shopping")
```

```
:-|
```

[ ]:

```
# multiple conditions (optimized)
x, y = "apple", "pineapple"
if x in fruits and y in fruits:
    print(":)")
elif (x in fruits) or (y in fruits):
    print(":-|")
else:
    print(":( go shopping")
```

```
:-|
```

2.0.2 for loop

A `for` loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

```
[ ]: fruits = ["apple", "banana", "orange", "cherry", "blueberry"]
     for x in fruits:
         print(x)
```

```
apple
banana
orange
cherry
blueberry
```

```
[ ]: for i in [0,1,2,3,4]:
         print(i)
```

```
0
1
2
3
4
```

```
[ ]: for i in list(range(10)):
         print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

```
[ ]: count = 0
     for i in range(10):
         count += 1
     count
```

[ ]: 10

[ ]:

```
total = 0
for i in range(10):
    total += i
total
```

[ ]:

```
count = 0
total = 0
for i in range(10):
    count += 1
    total += i
avg = total/count
avg
```

[ ]: 4.5

$$\frac{\sum_{i=0}^{9} i}{9}$$

[ ]:

```
```

[ ]:

```
for i, j in enumerate(range(10,20)):
    print(i,j)
```

```
0 10
1 11
2 12
3 13
4 14
5 15
6 16
7 17
8 18
9 19
```

[ ]:

```
for i, j in enumerate(range(1,5)):
    print(i, j**2)
```

```
0 1
1 4
2 9
3 16
```

[ ]:

```
```

[ ]:

```
for i in range(1, 10):
    if i % 2 == 0: # is i even?
```

```
        print(i**2)
    else:
        print(i)
```

```
1
4
3
16
5
36
7
64
9
```

[ ]:

```

```

[ ]:

```
x = {"name": "Pikachu", "Type": "Electric", "height": 40, "weight": 6}
for k, v in x.items(): # assumes you're talking about keys and values
    print(k)
    print(v)
```

```
name
Pikachu
Type
Electric
height
40
weight
6
```

[ ]:

```

```

2.0.3 While loop

`while` loop requires a condition. We can execute a set of statements as long as the condition is true.

[ ]:

```
i = 1
while i < 6:
    print(i)
    i += 1
```

```
1
2
3
4
5
```

[ ]:

```
# be careful to avoid infinite loop
i = 1
while i < 6:
    # print(i) <- commented out so it doesn't create an infinite loop
```

```
  Cell In[48], line 4
    # print(i)
              ^
SyntaxError: incomplete input
```

[ ]:

```
```

[ ]:

```
# Using a flag
end = False
i = 1

while not end:
    i += 1
    end = i > 5 # bool statement changes value of end
i
```

[ ]:

```
6
```

[ ]:

```
```

[ ]:

```
# find a largest power of 3 <= 1000
x = 1
while 3*x <= 1000:
    x = 3*x
x
```

[ ]:

```
729
```

[ ]:

```
x = 1
itr = 0
while 3*x <= 1000:
    x = 3*x
    itr += 1
x, itr # -> learn that it's the power of 6 by counting iterations
```

[ ]:

```
(729, 6)
```

[ ]:

```
```

2.0.4 Break, continue, pass

- Use `break` statement to exit a `for` loop or `while` loop immediately without running any remaining code in the loop.
- Use `continue` statement to stop the current iteration, and continue with the next.
- `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action.

[ ]:
```
for i in range(1, 10):
    if i % 2 == 0:
        break
    print(i)
```

```
1
```

[ ]:
```
for i in range(1, 10):
    if i % 2 == 0:
        continue
    print(i)
```

```
1
3
5
7
9
```

[ ]:
```
for i in range(1, 10):
    if i % 2 == 0:
        print("pass")
        pass
    print(i)
```

```
1
pass
2
3
pass
4
5
pass
6
7
pass
8
9
```

[ ]:
```
i = 0
while i < 10:
    i += 1
    if i % 2 == 0:
        break
    print(i)
```

```
1
```

[ ]:

```
i = 0
while i < 10:
    i += 1
    if i % 2 == 0:
        continue
    print(i)
```

```
1
3
5
7
9
```

[ ]:

```

```

2.0.5 List comprehension

A list comprehension combines the for loop and the creation of new elements into one line, and automatically appends each new element.

[ ]:

```
squares = [x**2 for x in range(1,11)]
squares
```

[ ]:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

[ ]:

```
[x**2 for x in range(1,11) if x % 2 != 0 ]
```

[ ]:

```
[1, 9, 25, 49, 81]
```

[ ]:

```
[x**2 if x % 2 != 0 else 1 for x in range(1,11)]
```

[ ]:

```
[1, 1, 9, 1, 25, 1, 49, 1, 81, 1]
```

[ ]:

```
{k:v for k, v in enumerate(range(5,0,-1))}
```

[ ]:

```
{0: 5, 1: 4, 2: 3, 3: 2, 4: 1}
```

[ ]:

```
{k:v for k, v in enumerate("abcd")}
```

[ ]:

```
{0: 'a', 1: 'b', 2: 'c', 3: 'd'}
```

[ ]:

```
x = ["apple", "banana", "orange", "peach", "apple"]
```

[ ]:

```
[i for i in x if i != "apple"]
```

[ ]:

```
['banana', 'orange', 'peach']
```

3 exercise

2. Write a program that prints the numbers from 1 to 100. But for multiples of three print “Fizz” instead of the number and for the multiples of five print “Buzz”. For numbers which are multiples of both three and five print “FizzBuzz”.
3. Write a program to print all the numbers between 1000 and 2000 which are divisible by 7 but are not a multiple of 5.
4. Write a program to calculate factorial of a number.
5. Fibonacci Sequence. Write a program that asks the user for a positive integer n and prints the first n numbers of the Fibonacci sequence.
6. List Processing. Write a program that takes a list of numbers and prints the largest and smallest numbers in the list.

[ ]:

```
for i in range(1, 101):
    if i % 15 == 0:
        print("FizzBuzz")
    elif i % 5 == 0:
        print("Buzz")
    elif i % 3 == 0:
        print("Fizz")
    else:
        print(i)
```

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
Fizz
```

```
22
23
Fizz
Buzz
26
Fizz
28
29
FizzBuzz
31
32
Fizz
34
Buzz
Fizz
37
38
Fizz
Buzz
41
Fizz
43
44
FizzBuzz
46
47
Fizz
49
Buzz
Fizz
52
53
Fizz
Buzz
56
Fizz
58
59
FizzBuzz
61
62
Fizz
64
Buzz
Fizz
67
68
Fizz
```

```
Buzz
71
Fizz
73
74
FizzBuzz
76
77
Fizz
79
Buzz
Fizz
82
83
Fizz
Buzz
86
Fizz
88
89
FizzBuzz
91
92
Fizz
94
Buzz
Fizz
97
98
Fizz
Buzz
```

[ ]:

```
for i in range(1000, 2000):
    if i % 7 == 0 and i % 5 != 0:
        print(i)
```

```
1001
1008
1022
1029
1036
1043
1057
1064
1071
1078
1092
1099
```

```
1106
1113
1127
1134
1141
1148
1162
1169
1176
1183
1197
1204
1211
1218
1232
1239
1246
1253
1267
1274
1281
1288
1302
1309
1316
1323
1337
1344
1351
1358
1372
1379
1386
1393
1407
1414
1421
1428
1442
1449
1456
1463
1477
1484
1491
1498
1512
1519
```

```
1526
1533
1547
1554
1561
1568
1582
1589
1596
1603
1617
1624
1631
1638
1652
1659
1666
1673
1687
1694
1701
1708
1722
1729
1736
1743
1757
1764
1771
1778
1792
1799
1806
1813
1827
1834
1841
1848
1862
1869
1876
1883
1897
1904
1911
1918
1932
1939
```

```
1946
1953
1967
1974
1981
1988
```

[ ]:

```
def factorial(n: int) -> int:
    """Finds the factorial of a number."""
    fac_n = n
    n -= 1
    while n > 0:
        fac_n *= n
        n -= 1
    return fac_n

factorial(10)
```

[ ]:

```
3628800
```

[ ]:

```
def fibonacci(n: int) -> list[int]:
    f = [0, 1]
    for i in range(1, n + 1):
        f.append(f[i - 1] + f[i])
    return f[0:n-1]

fibonacci(20)
```

[ ]:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584]
```

[ ]:

```
def minmax(l: list[int]) -> int:
    min = l[0]
    max = l[0]
    for i in l:
        if i > max:
            max = i
        elif i < min:
            min = i
    return min, max

minmax([23, 22, 33, 11, 55, 11, 32])
```

[ ]:

```
(11, 55)
```

4 Functions

A function is a block of reusable code that performs a specific task. Functions help organize code into manageable sections, make the code more readable, and allow for code reuse.

- `def function_name():` function definition, followed by the name of this funciton. Use meaning function names. The parentheses hold the information the function needs to do the job. Finally, the definition ends in a colon.
- indentation is important.
- docstring

[ ]:
```
def greeting():
    print("hello world!")
```

[ ]:
```
greeting()
```

```
hello world!
```

[ ]:
```

```

[ ]:
```
def f(a, b):
    """
    Do something with a and b.
    """

    return 2*a + 3*b
```

[ ]:
```
f(1,2)
```

[ ]:
```
8
```

[ ]:
```
# match each argument in the function call with a parameter in the function␣
 ↪definition
```

[ ]:
```
f(a=2, b=1)
```

[ ]:
```
7
```

[ ]:
```
f(b=1, a=2)
```

[ ]:
```
7
```

[ ]:
```
f()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[9], line 1
----> 1 f()

TypeError: f() missing 2 required positional arguments: 'a' and 'b'
```

[ ]:

```python
# default parameter value
def f(a, b = 4):
    return 4*a + 2*b

f(0)
```

[ ]:

```
8
```

[ ]:

```python
def f(a, b, c=10):
    return 2*a + 3*b - c
```

[ ]:

```python
f(2,3)
```

[ ]:

```
3
```

[ ]:

```python
f(2,3,20)
```

[ ]:

```
-7
```

When you use default values, any parameter with a default value needs to be listed after all the parameters that don't have default values. This allows Python to continue interpreting positional arguments correctly.

[ ]:

```python
# match parameter types
f('hello', 'world')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[15], line 2
      1 # match parameter types
----> 2 f('hello', 'world')

Cell In[12], line 2, in f(a, b, c)
      1 def f(a, b, c=10):
----> 2     return 2*a + 3*b - c

TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

[ ]:

```python
f([1,2,3], ['a', 'b', 'c'])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[16], line 1
----> 1 f([1,2,3], ['a', 'b', 'c'])

Cell In[12], line 2, in f(a, b, c)
      1 def f(a, b, c=10):
```

```
----> 2     return 2*a + 3*b - c

TypeError: unsupported operand type(s) for -: 'list' and 'int'
```

[ ]:
```
# variables
x = 3
def f(a,b):
    return a+b-x

f(2,5)
```

[ ]: 4

[ ]:
```
def f(a,b):
    return a+b-y
f(2,5)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[18], line 3
      1 def f(a,b):
      2     return a+b-y
----> 3 f(2,5)

Cell In[18], line 2, in f(a, b)
      1 def f(a,b):
----> 2     return a+b-y

NameError: name 'y' is not defined
```

[ ]:
```
# variables defined within a function are not available within the global scope
def f(a,b):
    y = 10
    return a + b -2*y
f(2,3)
```

[ ]: -15

[ ]:
```
y
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[20], line 1
----> 1 y
```

```
NameError: name 'y' is not defined
```

[ ]:

```
# return
```

[ ]:

```
def f(a,b):
    y = 10
    res = a + b -2*y
    return res
```

[ ]:

```
def f(a,b):
    return a, b, 2*a + 3*b
f(2,3)
```

[ ]:

```
(2, 3, 13)
```

[ ]:

```
x, y, z = f(2,3)
print(x, y, z)
```

```
2 3 13
```

[ ]:

```

```

[ ]:

```
# write if-statement, for loop, ... in the function
```

[ ]:

```
def is_odd(x):
    if x % 2 == 1:
        return True
    else:
        return False
```

[ ]:

```
is_odd(3)
```

[ ]:

```
True
```

[ ]:

```
is_odd(4)
```

[ ]:

```
False
```

[ ]:

```

```

[ ]:

```
# calculate the avergae score for each session
def avg_score(x):
    """
    Input:
    x: a list of scores for one session

    Output:
    average score of this session
    """
    total_score = 0
    num = 0
    for i in x:
        total_score += i
        num += 1

    avg_score = total_score/num
    return avg_score
```

[ ]:
```
avg_score([100,90,80, 85, 60])
```

[ ]:
```
83.0
```

[ ]:
```
def largest_power_below(a, max_num=1000):
    num = 1
    while a*num <= max_num:
        num *= a
    return num
```

[ ]:
```
largest_power_below(3)
```

[ ]:
```
729
```

[ ]:
```
largest_power_below(3, 5000)
```

[ ]:
```
2187
```

[ ]:

The special `*args` argument can be passed to the function. `*arg` tells Python to make an empty tuple and pack whatever values it receives into this tuple.

[ ]:
```
def better_sum(*args):
    total = 0
    for i in args:
        total += i
    return total
```

[ ]:
```
better_sum(1)
```

[ ]:
```
1
```

[ ]:
```
better_sum(1,5,7)
```

[ ]:
```
13
```

[ ]:

Sometimes you'll want to accept an arbitrary number of arguments, but you won't know ahead of time what kind of information will be passed to the function. `**kwargs` allows writing functions that accept as many key-value pairs as the calling statement provides.

[ ]:
```
def shopping(**kwargs):
    for key, val in kwargs.items():
        print(key, val)

shopping(produce = "lettuce", fruit = "apple")
```

```
produce lettuce
fruit apple
```

We will see more examples later.

[ ]:
```

```

4.0.1 Store functions in modules

We can store functions in a separate file called *module*, and then use `import` to import that module into the main program when we need it. An `import` statement tells Python to make the code in a module available in the currently running program file.

Advantages: 1. focus on higher-level logic in programming 2. reuse functions in many different programs 3. easy to share a single file without sharing the entire program. 4. Knowing how to import functions also allows us to use libraries of functions that other programmers have written.

[ ]:
```
%%file grades.py

def avg_score(x):
    return sum(x)/len(x)
```

```
Writing grades.py
```

[ ]:
```
scores = [100,94.6, 93.2, 85, 78]
```

[ ]:
```
import grades
grades.avg_score(scores)
```

```
[ ]: 90.16
```

[ ]:
```
from grades import avg_score
avg_score(scores)
```

[ ]:
```
from grades import *
avg_score([100,94.6, 93.2, 85, 78])
```

```
[ ]: 90.16
```

[ ]:

```
import grades as gr
gr.avg_score(scores)
```

[ ]:

```
90.16
```

[ ]:

```
```

[ ]:

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
```

[ ]:

```
```

Recursive functions Anonymous functions Lazy evaluation Higher-order functions Decorators Partial application Using operator Using functional Using itertools Pipelines with toolz

4.0.2 Recursive functions

A recursive function is one that calls itself. It's quite useful for some data structures such as trees. But be careful.

[ ]:

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

[ ]:

```
fibonacci(10)
```

[ ]:

```
55
```

[ ]:

```
```

4.0.3 Anonymous functions

An anonymous function in Python is a function without a name. A `lambda` function is a small anonymous function, i.e., `lambda arguments : expression`. It can take any number of arguments, but can only have one expression.

[ ]:

```
x = lambda a: 2**a
x(5)
```

[ ]:

```
32
```

[ ]:

```
text = "hello world"

upper = lambda string: string.upper()
```

```
print(upper(text))
```

[ ]:

```
add = lambda x, y: x+y
add(5, 8)
```

[ ]:

```
get_min = lambda x, y: x if x < y else y
get_min(7, 5)
```

[ ]:

```
x = [1,2,4,5,3,7,8,3]
sorted(x, key = lambda x: x%2 == 0)
```

[ ]:

```
student_tuples = [
    ('john', 'A', 15),
    ('jane', 'B', 12),
    ('dave', 'B', 10),
]
sorted(student_tuples, key=lambda student: student[2])
```

[ ]:

```
x = {"a": [1,2,3, 4], "b": [1,3,4], "c": [1,2,3,4,5]}
x_new = dict(sorted(x.items(), key = lambda item: len(item[1]), reverse=True))
x_new
```

[ ]:

```
# map, zip, filter
```