* ****List**** is a collection which is ordered and changeable. Allows duplicate members.
* **[Tuple](https://www.w3schools.com/python/python_tuples.asp)** is a collection which is ordered and unchangeable. Allows duplicate members.
* **[Set](https://www.w3schools.com/python/python_sets.asp)** is a collection which is unordered, unchangeable\*, and unindexed. No duplicate members.
* **[Dictionary](https://www.w3schools.com/python/python_dictionaries.asp)** is a collection which is ordered\*\* and changeable. No duplicate members.

help(round)

Python isn't smart enough to read my code and turn it into a nice English description. However, when I write a function, I can provide a description in what's called the **docstring**.

Docstrings

In [7]:

linkcode

def least\_difference(a, b, c):

*"""Return the smallest difference between any two numbers*

*among a, b and c.*

*>>> least\_difference(1, 5, -5)*

*4*

*"""*

diff1 = abs(a - b)

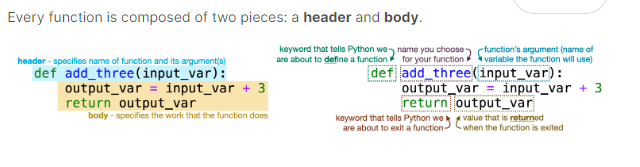
diff2 = abs(b - c)

diff3 = abs(a - c)

return min(diff1, diff2, diff3)

The docstring is a triple-quoted string (which may span multiple lines) that comes immediately after the header of a function. When we call help() on a function, it shows the docstring.

**Functions**



def greet(who="Colin"):

print("Hello,", who)

greet()

greet(who="Kaggle")

*# (In this case, we don't need to specify the name of the argument, because it's unambiguous.)*

greet("world")

Hello, Colin

Hello, Kaggle

Hello, world

## **Functions Applied to Functions**

Here's something that's powerful, though it can feel very abstract at first. You can supply functions as arguments to other functions. Some example may make this clearer:

In [14]:

def mult\_by\_five(x):

return 5 \* x

def call(fn, arg):

*"""Call fn on arg"""*

return fn(arg)

def squared\_call(fn, arg):

*"""Call fn on the result of calling fn on arg"""*

return fn(fn(arg))

print(

call(mult\_by\_five, 1),

squared\_call(mult\_by\_five, 1),

sep='**\n**', *# '\n' is the newline character - it starts a new line*

)

5

25

Functions that operate on other functions are called "higher-order functions." You probably won't write your own for a little while. But there are higher-order functions built into Python that you might find useful to call.

Here's an interesting example using the max function.

By default, max returns the largest of its arguments. But if we pass in a function using the optional key argument, it returns the argument x that maximizes key(x) (aka the 'argmax').

In [15]:

def mod\_5(x):

*"""Return the remainder of x after dividing by 5"""*

return x % 5

print(

'Which number is biggest?',

max(100, 51, 14),

'Which number is the biggest modulo 5?',

max(100, 51, 14, key=mod\_5),

sep='**\n**',

)

Which number is biggest?

100

Which number is the biggest modulo 5?

14

**List comprehension**

squares = [n\*\*2 for n **in** range(10)]squares

Out[6]:

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Here's how we would do the same thing without a list comprehension:

In [7]:

squares = []for n **in** range(10):

squares.append(n\*\*2)squares

Out[7]:

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

We can also add an if condition:

In [8]:

short\_planets = [planet for planet **in** planets if len(planet) < 6]short\_planets

Out[8]:

['Venus', 'Earth', 'Mars']

*# str.upper() returns an all-caps version of a string*loud\_short\_planets = [planet.upper() + '!' for planet **in** planets if len(planet) < 6]loud\_short\_planets

Out[9]:

['VENUS!', 'EARTH!', 'MARS!']

People usually write these on a single line, but you might find the structure clearer when it's split up over 3 lines:

In [10]:

linkcode

[

planet.upper() + '!'

for planet **in** planets

if len(planet) < 6]

Out[10]:

['VENUS!', 'EARTH!', 'MARS!']

**Examples:-**

def count\_negatives(nums):

*"""Return the number of negative numbers in the given list. >>> count\_negatives([5, -1, -2, 0, 3]) 2 """*

n\_negative = 0

for num **in** nums:

if num < 0:

n\_negative = n\_negative + 1

return n\_negative

Here's a solution using a list comprehension:

In [13]:

linkcode

def count\_negatives(nums):

return len([num for num **in** nums if num < 0])

def count\_negatives(nums):

*# Reminder: in the "booleans and conditionals" exercises, we learned about a quirk of*

*# Python where it calculates something like True + True + False + True to be equal to 3.*

return sum([num < 0 for num **in** nums])

def elementwise\_greater\_than(L, thresh):

"""Return a list with the same length as L, where the value at index i is

True if L[i] is greater than thresh, and False otherwise.

>>> elementwise\_greater\_than([1, 2, 3, 4], 2)

[False, False, True, True]

"""

return [i > thresh for i in L]

# Check your answer

q2.check()

**Format**

pluto\_mass = 1.303 \* 10\*\*22earth\_mass = 5.9722 \* 10\*\*24population = 52910390*# 2 decimal points 3 decimal points, format as percent separate with commas*"**{}** weighs about **{:.2}** kilograms (**{:.3%}** of Earth's mass). It is home to **{:,}** Plutonians.".format(

planet, pluto\_mass, pluto\_mass / earth\_mass, population,)

Out[26]:

"Pluto weighs about 1.3e+22 kilograms (0.218% of Earth's mass). It is home to 52,910,390 Plutonians."

In [27]:

linkcode

*# Referring to format() arguments by index, starting from 0*s = """Pluto's a **{0}**.No, it's a **{1}**.**{0}**!**{1}**!""".format('planet', 'dwarf planet')print(s)

Pluto's a planet.

No, it's a dwarf planet.

planet!

dwarf planet!

**Dictionaries**

planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']planet\_to\_initial = {planet: planet[0] for planet **in** planets}planet\_to\_initial

Out[32]:

{'Mercury': 'M',

'Venus': 'V',

'Earth': 'E',

'Mars': 'M',

'Jupiter': 'J',

'Saturn': 'S',

'Uranus': 'U',

'Neptune': 'N'}

For loop

for k **in** numbers:

print("**{}** = **{}**".format(k, numbers[k]))

one = Pluto

two = 2

three = 3

eleven = 11

*# Get all the initials, sort them alphabetically, and put them in a space-separated string.*' '.join(sorted(planet\_to\_initial.values()))

Out[36]:

'E J M M N S U V'

planet\_to\_initial = {planet: planet[0] for planet **in** planets}planet\_to\_initial

Out[32]:

{'Mercury': 'M',

'Venus': 'V',

'Earth': 'E',

'Mars': 'M',

'Jupiter': 'J',

'Saturn': 'S',

'Uranus': 'U',

'Neptune': 'N'}

for planet, initial **in** planet\_to\_initial.items():

print("**{}** begins with **\"{}\"**".format(planet.rjust(10), initial))

Mercury begins with "M"

Venus begins with "V"

Earth begins with "E"

Mars begins with "M"

Jupiter begins with "J"

Saturn begins with "S"

Uranus begins with "U"

Neptune begins with "N"

**Imports**

from math import \*

print(pi, log(32, 2))

\* is all names from a module

Arrays and lists

xlist = [[1,2,3],[2,4,6],]*# Create a 2-dimensional array*x = numpy.asarray(xlist)print("xlist = **{}\n**x =**\n{}**".format(xlist, x))

xlist = [[1, 2, 3], [2, 4, 6]]

x =

[[1 2 3]

[2 4 6]]

In [24]:

*# Get the last element of the second row of our numpy array*x[1,-1]

Out[24]:

6

Qstn

# 2.

This is a very challenging problem. Don't forget that you can receive a hint!

Luigi is trying to perform an analysis to determine the best items for winning races on the Mario Kart circuit. He has some data in the form of lists of dictionaries that look like...

[

{'name': 'Peach', 'items': ['green shell', 'banana', 'green shell',], 'finish': 3},

{'name': 'Bowser', 'items': ['green shell',], 'finish': 1},

# Sometimes the racer's name wasn't recorded

{'name': None, 'items': ['mushroom',], 'finish': 2},

{'name': 'Toad', 'items': ['green shell', 'mushroom'], 'finish': 1},

]

'items' is a list of all the power-up items the racer picked up in that race, and 'finish' was their placement in the race (1 for first place, 3 for third, etc.).

He wrote the function below to take a list like this and return a dictionary mapping each item to how many times it was picked up by first-place finishers.

sample = [

{'name': 'Peach', 'items': ['green shell', 'banana', 'green shell',], 'finish': 3},

{'name': 'Bowser', 'items': ['green shell',], 'finish': 1},

{'name': None, 'items': ['mushroom',], 'finish': 2},

{'name': 'Toad', 'items': ['green shell', 'mushroom'], 'finish': 1},

]

best\_items(sample)

Code:-

# Import luigi's full dataset of race data

from learntools.python.luigi\_analysis import full\_dataset

# Fix me!

def best\_items(racers):

winner\_item\_counts = {}

for i in range(len(racers)):

# The i'th racer dictionary

racer = racers[i]

# We're only interested in racers who finished in first

if racer['finish'] == 1:

for n in racer['items']:

# Add one to the count for this item (adding it to the dict if necessary)

if n not in winner\_item\_counts:

winner\_item\_counts[i] = 0

winner\_item\_counts[i] += 1

# Data quality issues :/ Print a warning about racers with no name set. We'll take care of it later.

if racer['name'] is None:

print("WARNING: Encountered racer with unknown name on iteration {}/{} (racer = {})".format(

i+1, len(racers), racer['name'])

)

return winner\_item\_counts

# Try analyzing the imported full dataset

best\_items(full\_dataset)

**Crazy code:-**

#def blackjack\_hand\_greater\_than(hand\_1, hand\_2):

"""

Return True if hand\_1 beats hand\_2, and False otherwise.

In order for hand\_1 to beat hand\_2 the following must be true:

- The total of hand\_1 must not exceed 21

- The total of hand\_1 must exceed the total of hand\_2 OR hand\_2's total must exceed 21

Hands are represented as a list of cards. Each card is represented by a string.

When adding up a hand's total, cards with numbers count for that many points. Face

cards ('J', 'Q', and 'K') are worth 10 points. 'A' can count for 1 or 11.

When determining a hand's total, you should try to count aces in the way that

maximizes the hand's total without going over 21. e.g. the total of ['A', 'A', '9'] is 21,

the total of ['A', 'A', '9', '3'] is 14.

Examples:

>>> blackjack\_hand\_greater\_than(['K'], ['3', '4'])

True

>>> blackjack\_hand\_greater\_than(['K'], ['10'])

False

>>> blackjack\_hand\_greater\_than(['K', 'K', '2'], ['3'])

False

"""

**Code:-**

def hand\_total(hand):

total = 0

# Count the number of aces and deal with how to apply them at the end.

aces = 0

for card in hand:

if card in ['J', 'Q', 'K']:

total += 10

elif card == 'A':

aces += 1

else:

# Convert number cards (e.g. '7') to ints

total += int(card)

# At this point, total is the sum of this hand's cards \*not counting aces\*.

# Add aces, counting them as 1 for now. This is the smallest total we can make from this hand

total += aces

# "Upgrade" aces from 1 to 11 as long as it helps us get closer to 21

# without busting

while total + 10 <= 21 and aces > 0:

# Upgrade an ace from 1 to 11

total += 10

aces -= 1

return total

def blackjack\_hand\_greater\_than(hand\_1, hand\_2):

total\_1 = hand\_total(hand\_1)

total\_2 = hand\_total(hand\_2)

return total\_1 <= 21 and (total\_1 > total\_2 or total\_2 > 21)

# Check your answer

q3.check()