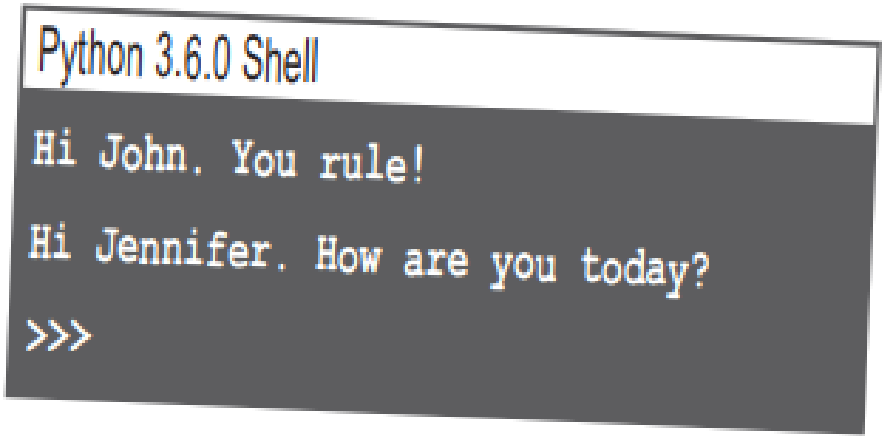
Python: Getting Functional

Going further with parameters: default values and keywords

```
def greet(name, message='You rule!’):
     print('Hi', name + '.', message)


greet('John’)


greet('Jennifer', 'How are you today?')
```



Always list your required parameters first!

```
def greet(name, message='You rule!', emoticon):
     print('Hi', name + '.', message, emoticon)
```

Python 3.6.0 Shell

```
File "defaults.py", line 1
    def greet(name, message='You
rule!', emoticon):
              ^
SyntaxError: non-default argument
follows default argument
```

Now the non-defaults (required) parameters are first...

...followed by the optional ones.

```
def greet(name, emoticon, message='You rule!’):
     print('Hi', name + '.', message, emoticon)
```

```
def greet(name, emoticon, message='You rule!’):
     print('Hi', name + '.', message, emoticon)
```

```
greet(message='Where have you been?', name='Jill', emoticon='thumbs up’)
```

What is the output here !!

Using keywords we can mix and match the order of our arguments and even omit them if they have defaults. Just make sure your calls provide any required arguments before the keywords arguments.

```
def greet(name, emoticon, message='You rule!’):
     print('Hi', name + '.', message, emoticon)



greet('Betty', message='Yo!', emoticon=':)')
```

What is the output here !!

how those keyword arguments work into your brain

```
def make_sundae(ice_cream='vanilla', sauce='chocolate’,
                nuts=True, banana=True, brownies=False,
                whipped_cream=True):

      recipe = ice_cream + ' ice cream and ' + sauce + ' sauce ‘

      if nuts:
            recipe = recipe + 'with nuts and ‘
      if banana:
            recipe = recipe + 'a banana and ‘
      if brownies:
            recipe = recipe + 'a brownie and ‘
      if not whipped_cream:
            recipe = recipe + 'no ‘

      recipe = recipe + 'whipped cream on top.’

      return recipe
```

how those keyword arguments work into your brain

```
def make_sundae(ice_cream='vanilla', sauce='chocolate’,
                nuts=True, banana=True, brownies=False,
                whipped_cream=True):

      recipe = ice_cream + ' ice cream and ' + sauce + ' sauce ‘
      if nuts:
            recipe = recipe + 'with nuts and ‘
      if banana:
            recipe = recipe + 'a banana and ‘
      if brownies:
            recipe = recipe + 'a brownie and ‘
      if not whipped_cream:
            recipe = recipe + 'no ‘
      recipe = recipe + 'whipped cream on top.’
      return recipe

sundae = make_sundae()
print('One sundae coming up with', sundae)
# One sundae coming up with vanilla ice cream and chocolate sauce with nuts and a banana and whipped cream on top.
```

how those keyword arguments work into your brain

```
def make_sundae(ice_cream='vanilla', sauce='chocolate’,
                nuts=True, banana=True, brownies=False,
                whipped_cream=True):

      recipe = ice_cream + ' ice cream and ' + sauce + ' sauce ‘
      if nuts:
            recipe = recipe + 'with nuts and ‘
      if banana:
            recipe = recipe + 'a banana and ‘
      if brownies:
            recipe = recipe + 'a brownie and ‘
      if not whipped_cream:
            recipe = recipe + 'no ‘
      recipe = recipe + 'whipped cream on top.’
      return recipe

sundae = make_sundae('chocolate’)
print('One sundae coming up with', sundae)
# One sundae coming up with chocolate ice cream and chocolate sauce with nuts and a banana and whipped cream on top.
```

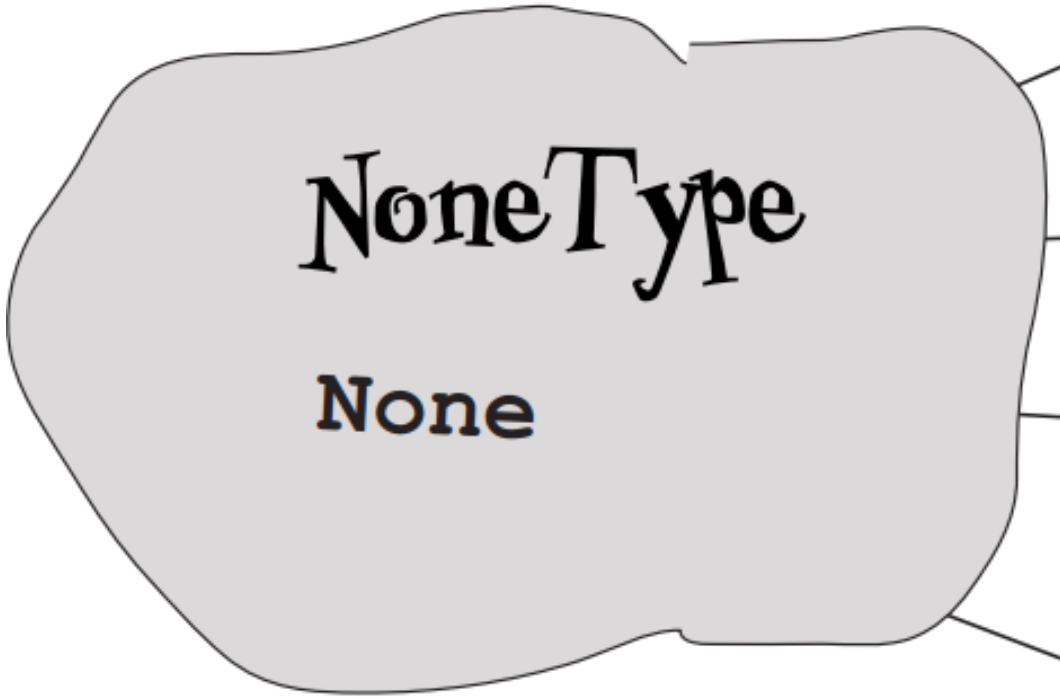
how those keyword arguments work into your brain

```
def make_sundae(ice_cream='vanilla', sauce='chocolate’,
nuts=True, banana=True, brownies=False,
whipped_cream=True):

      recipe = ice_cream + ' ice cream and ' + sauce + ' sauce ‘
      if nuts:
            recipe = recipe + 'with nuts and ‘
      if banana:
            recipe = recipe + 'a banana and ‘
      if brownies:
            recipe = recipe + 'a brownie and ‘
      if not whipped_cream:
            recipe = recipe + 'no ‘
      recipe = recipe + 'whipped cream on top.’
      return recipe

sundae = make_sundae(sauce='caramel', whipped_cream=False, banana=False)
print('One sundae coming up with', sundae)
# One sundae coming up with vanilla ice cream and caramel sauce with nuts and no whipped cream on top.
```

how those keyword arguments work into your brain

```
def make_sundae(ice_cream='vanilla', sauce='chocolate',
                nuts=True, banana=True, brownies=False,
                whipped_cream=True):
      recipe = ice_cream + ' ice cream and ' + sauce + ' sauce ‘
      if nuts:
            recipe = recipe + 'with nuts and ‘
      if banana:
            recipe = recipe + 'a banana and ‘
      if brownies:
            recipe = recipe + 'a brownie and ‘
      if not whipped_cream:
            recipe = recipe + 'no ‘
      recipe = recipe + 'whipped cream on top.’
      return recipe

sundae = make_sundae(whipped_cream=False, banana=True,
                     brownies=True, ice_cream='peanut butter’)
print('One sundae coming up with', sundae)
# One sundae coming up with peanut butter ice cream and chocolate sauce with nuts and a banana and a brownie and no
whipped cream on top.
```

If a function doesn’t have a return statement, does it return anything?



Has a single value None.

Uses no quotes around None.

First letter always capitalized.

Expressions can evaluate to None.

```
balance = 10500
camera_on = True

def steal(balance, amount):
    global camera_on
    camera_on = False
    if (amount < balance):
        balance = balance - amount
    return amount
    camera_on = True

proceeds = steal(balance, 1250)
print('Criminal: you stole', proceeds)
```

This is the real, actual bank balance in the account.

sorting and nested iteration

Experimental Test Data on 400 Mhz Pentium II RedHat Linux Machine								
	100	**1000**	**2000**	**4000**	**10,000**	**100,000**	**1,000,000**	**10,000,000**
Bubble	1	131	439	1954	7612	***760000*** *(12.67 min)*	***76000000*** *(21.11 hrs)*	***7600000000*** *(87.96 days)*
Selection	1	35	179	588	4280	***428000*** *(7.13 min)*	***42800000*** *(11.89 hrs)*	***4280000000*** *(49.54 days)*
Insertion	1	30	98	336	2425	***242500*** *(4.04 min)*	***24250000*** *(6.74 hrs)*	***2425000000*** *(28.07 days)*
Merge	3	5	8	15	47	521	5580	***60000***
Quick	1	3	4	7	17	189	2290	27006

`time in ms (second * 1000)`

Data: circa 2019

Understanding bubble sort

```
Input:  [6, 2, 5, 3, 9]

Return: [2, 3, 5, 6, 9]
```

Sharpen your pencil

Take the list below and bubble sort it

```
['coconut', 'strawberry', 'banana', 'pineapple']
```

just compare them alphabetically (what a computer scientist would call lexicographically, otherwise known as dictionary order).

Some bubble sort pseudocode

```
DEFINE a function bubble_sort(list):
    DECLARE a variable swapped and set to True.

    WHILE swapped:
        SET swapped to False.
        FOR variable i in range(0, len(list)-1)
            IF list[i] > list[i+1]:
                DECLARE a variable temp and set to list[i].
                SET list[i] to list[i+1]
                SET list[i+1] to temp
                SET swapped to True.
```

This pseudocode sorts a list in ascending order. What change would you need to make to sort in descending order?

```
for i in range(0,4):
    for j in range(0,4):
        print(i * j)
```

```
for i in range(0,4):
    for j in range(0,4):
        print(i * j)
```

Python 3.6.0 Shell

```
0
0
0
0
0
1
2
3
0
2
4
6
0
3
6
9
>>>
```

```
for word in ['ox', 'cat', 'lion', 'tiger', 'bobcat']:
    for i in range(2, 7):
        letters = len(word)
        if (letters % i) == 0:
            print(i, word)
```

```
for word in ['ox', 'cat', 'lion', 'tiger', 'bobcat']:
    for i in range(2, 7):
        letters = len(word)
        if (letters % i) == 0:
            print(i, word)
```

Python 3.6.0 Shell

```
2 ox
3 cat
2 lion
4 lion
5 tiger
2 bobcat
3 bobcat
6 bobcat
>>>
```

```
full = False

donations = []
full_load = 45

toys = ['robot', 'doll', 'ball', 'slinky']

while not full:
    for toy in toys:
        donations.append(toy)
        size = len(donations)
        if (size >= full_load):
            full = True

print('Full with', len(donations), 'toys')
print(donations)
```

```
full = False

donations = []
full_load = 45

toys = ['robot', 'doll', 'ball', 'slinky']

hile not full:
    for toy in toys:
        donations.append(toy)
        size = len(donations)
        if (size >= full_load):
            full = True

int('Full with', len(donations), 'toys')
int(donations)
```

Python 3.6.0 Shell

```
Full with 48 toys
['robot', 'doll', 'ball', 'slinky', 'robot', 'doll',
'ball', 'slinky', 'robot', 'doll', 'ball', 'slinky',
'robot', 'doll', 'ball', 'slinky', 'robot', 'doll',
'ball', 'slinky', 'robot', 'doll', 'ball', 'slinky',
'robot', 'doll', 'ball', 'slinky', 'robot', 'doll',
'ball', 'slinky', 'robot', 'doll', 'ball', 'slinky',
'robot', 'doll', 'ball', 'slinky', 'robot', 'doll', 'ball',
'slinky', 'robot', 'doll', 'ball', 'slinky']
>>>
```

Implementing bubble sort in Python

```
def bubble_sort(scores):
    swapped = True
    while swapped:
        swapped = False
        for i in range(0, len(scores)-1):
            if scores[i] > scores[i+1]:
                temp = scores[i]
                scores[i] = scores[i+1]
                scores[i+1] = temp
                swapped = True
```

We’d like to have the solutions with the highest bubble scores first (in other words we want descending order, not ascending).