Check if Key Exists

To determine if a specified key is present in a dictionary use the `in` keyword:

Check if "model" is present in the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

Change Dictionary Items

Change Values

You can change the value of a specific item by referring to its key name:

Change the "year" to 2018:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
```

Update Dictionary

The `update()` method will update the dictionary with the items from the given argument.

The argument must be a dictionary, or an iterable object with key:value pairs.

Update the "year" of the car by using the `update()` method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"year": 2020})
```

Add Dictionary Items

Adding Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
```

Update Dictionary

The `update()` method will update the dictionary with the items from a given argument. If an item does not exist, it will be added.

The argument must be a dictionary, or an iterable object with key:value pairs.

Add a color item to the dictionary by using the `update()` method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"color": "red"})
```

Remove Dictionary Items

Removing Items

There are several methods to remove items from a dictionary:

The `pop()` method removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

The `popitem()` method removes the last inserted item (in versions before 3.7, a random item is removed instead):

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

The `del` keyword removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)
```

The `clear()` method empties the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
```

Loop Dictionaries

Loop Through a Dictionary

You can loop through a dictionary by using a `for` loop.

When looping through a dictionary, the return value is the dictionary's keys, but there are also methods to return the *values*.

Print all key names in the dictionary, one by one:

```
for x in thisdict:

  print(x)
```

Print all *values* in the dictionary, one by one:

```
for x in thisdict:

  print(thisdict[x])
```

You can also use the `values()` method to return values of a dictionary:

```
for x in thisdict.values():

  print(x)
```

You can use the `keys()` method to return the keys of a dictionary:

```
for x in thisdict.keys():

  print(x)
```

Loop through both *keys* and *values*, by using the `items()` method:

```
for x, y in thisdict.items():

  print(x, y)
```

Copy Dictionaries

Copy a Dictionary

You cannot copy a dictionary simply by typing `dict2 = dict1`, because: `dict2` will only be a *reference* to `dict1`, and changes made in `dict1` will automatically also be made in `dict2`.

There are ways to make a copy, one way is to use the built-in Dictionary method `copy()`.

Make a copy of a dictionary with the `copy()` method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```

Another way to make a copy is to use the built-in function `dict()`.

Make a copy of a dictionary with the `dict()` function:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = dict(thisdict)
print(mydict)
```

Nested Dictionaries

Nested Dictionaries

A dictionary can contain dictionaries, this is called nested dictionaries.

Create a dictionary that contain three dictionaries:

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
```

Or, if you want to add three dictionaries into a new dictionary:

Create three dictionaries, then create one dictionary that will contain the other three dictionaries:

```
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}

myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
```

Access Items in Nested Dictionaries

To access items from a nested dictionary, you use the name of the dictionaries, starting with the outer dictionary:

Print the name of child 2:

```
print(myfamily["child2"]["name"])
```

Loop Through Nested Dictionaries

You can loop through a dictionary by using the `items()` method like this:

Loop through the keys and values of all nested dictionaries:

```
for x, obj in myfamily.items():
  print(x)

  for y in obj:
    print(y + ':', obj[y])
```

Dictionary Methods

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key-value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Python Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Creating a Function

In Python a function is defined using the `def` keyword:

```
def my_function():
  print("Hello from a function")
```

```
# Example Python Code for User-Defined function
def square( num ):
  """
  This function computes the square of the number.
  """
  return num**2
object_ = square(6)
print( "The square of the given number is: ", object_ )
```

Calling a Function

To call a function, use the function name followed by parenthesis:

```
def my_function():

  print("Hello from a function")

my_function()
```

Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

***Arguments* are often shortened to *args* in Python documentation.**

Parameters or Arguments?

The terms *parameter* and *argument* can be used for the same thing: information that is passed into a function.

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", "Refsnes")
```

If you try to call the function with 1 or 3 arguments, you will get an error:

This function expects 2 arguments, but gets only 1:

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil")
```

Arbitrary Arguments, *args

If you do not know how many arguments will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments and can access the items accordingly:

If the number of arguments is unknown, add a * before the parameter name:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

***Arbitrary Arguments* are often shortened to **args* in Python documentation.**

Keyword Arguments

You can also send arguments with the *key* = *value* syntax.

This way the order of the arguments does not matter.

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 =
"Linus")
```

The phrase *Keyword Arguments* is often shortened to *kwargs* in Python documentation.

Arbitrary Keyword Arguments, **kwargs

If you do not know how many keyword arguments will be passed into your function, add two asterisk: ** before the parameter name in the function definition.

This way the function will receive a *dictionary* of arguments and can access the items accordingly:

If the number of keyword arguments is unknown, add a double ** before the parameter name:

```
def my_function(**kid):

  print("His last name is " + kid["lname"])


my_function(fname = "Tobias", lname = "Refsnes")
```

Arbitrary Kword Arguments **are often shortened to *****kwargs* **in Python documentation.**

```
def my_function(country = "Norway"):

  print("I am from " + country)

my_function("Sweden")

my_function("India")

my_function()

my_function("Brazil")
```

Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

```
def my_function(food):

  for x in food:

    print(x)
fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

Return Values

To let a function return a value, use the `return` statement:

```
def my_function(x):

  return 5 * x

print(my_function(3))

print(my_function(5))

print(my_function(9))
```

The pass Statement

`function` definitions cannot be empty, but if you for some reason have a `function` definition with no content, put in the `pass` statement to avoid getting an error.

```
def myfunction():
  Pass
```

Positional-Only Arguments

You can specify that a function can have ONLY positional arguments or ONLY keyword arguments.

To specify that a function can have only positional arguments, add `, /` after the arguments:

```
def my_function(x, /):

  print(x)

my_function(3)
```

Without the `, /` you are allowed to use keyword arguments even if the function expects positional arguments:

Without the , / you are allowed to use keyword arguments even if the function expects positional arguments:

```
def my_function(x):

  print(x)

my_function(x = 3)
```

But when adding the , / you will get an error if you try to send a keyword argument:

```
def my_function(x, /):

  print(x)

my_function(x = 3)
```

Without the *, you are allowed to use position arguments even if the function expects keyword arguments:

```
def my_function(x):

  print(x)

my_function(3)
```

But with the *, you will get an error if you try to send a positional argument:

```
def my_function(*, x):

  print(x)

my_function(3)
```

Combine Positional-Only and Keyword-Only

You can combine the two argument types in the same function.

Any argument *before* the / , are positional-only, and any argument *after* the *, are keyword-only.

```
def my_function(a, b, /, *, c, d):

  print(a + b + c + d)

my_function(5, 6, c = 7, d = 8)
```

The / in a Python function definition is used to specify that the arguments before it must be passed positionally, not by name. This makes the function more predictable in how it accepts inputs.

Here's what this means in simple terms:

Explanation:

In your example:

```
def my_function(x, /):

    print(x)
```

- **The / means that the x parameter must be provided directly by its position when you call the function.**

Example of correct usage:

```
my_function(3)  # Works fine because 3 is given by position.
```

Example of incorrect usage:

```
my_function(x=3)  # Throws an error because `x` is passed by name.
```

-

Why Use /?

1. **Clarity: It ensures the function is used only with positional arguments. This is helpful if you don't want users of your function to pass arguments by name.**
2. **Consistency: Some built-in Python functions, like len(), work the same way. They don't allow named arguments, just positional ones.**

So, the / is like saying:

> **"You can only give values to these parameters by their position, not their name."**

why use *

The `*` in a Python function definition is used to specify that the arguments after it must be passed by name, not by position. This improves code readability and ensures clarity when passing multiple parameters.

How it works:

When you include `*` in a function definition, all parameters after the `*` must be keyword arguments (passed using `name=value`).

Example:

```
def my_function(x, *, y):

    print(f"x: {x}, y: {y}")

# Correct Usage

my_function(5, y=10)  # x is passed positionally, y is passed by name.

# Output: x: 5, y: 10

# Incorrect Usage

my_function(5, 10)  # Error: y must be passed as a keyword argument.
```

Why Use `*`?

Improves Clarity: It ensures that some arguments are passed by name, which makes the function call more readable.
For example, in:
`calculate_tax(income, *, tax_rate=0.1, deductions=500)`

1. **It's clear what `tax_rate` and `deductions` represent when passed as `tax_rate=0.15` and `deductions=1000`.**
2. **Avoids Errors: It prevents accidental misuse of parameters by enforcing explicit naming.**
3. **Consistency with Built-ins: Some built-in functions, like `print`, allow keyword-only arguments:**
   `print("Hello", end="!")  # `end` must be passed by name.`

Recursion

Recursion is when a function calls itself to solve a smaller version of the same problem. It continues doing this until it reaches a point where the problem is so simple it can be solved without further calls. This point is called the base case.

Easy Way to Think About It:

Imagine you have to find the sum of numbers from 1 to 5:

- **Add 1 + (the sum of numbers from 2 to 5).**
- **To find the sum of numbers from 2 to 5, add 2 + (the sum of numbers from 3 to 5).**
- **Keep doing this until you get to the sum of just 5, which is easy: it's 5!**

This is how recursion works—it keeps breaking a problem into smaller parts until it hits a simple problem that it can solve.

The Two Key Parts of Recursion:

1. **Base Case: When the function stops calling itself (e.g., when the number is 1).**
2. **Recursive Case: When the function calls itself to solve a smaller problem.**

Most Easy Example 1: Factorial of a Number

The factorial of a number `n` is calculated as:

- **`n! = n × (n-1)!`**
- **And the factorial of `1` is `1`.**

```
def factorial(n):
    if n == 1:  # Base case
        return 1
    return n * factorial(n - 1)  # Recursive case

print(factorial(5))  # Output: 120
```

Recursion Remove duplicate

```
def remove_adjacent_duplicates(string):
    stack = []  # Using a stack to store non-duplicate
characters
    for char in string:
        if stack and char == stack[-1]:
            stack.pop()  # Remove the top element from the
stack if it is a duplicate
        else:
            stack.append(char)  # Push the character onto the
stack if it is not a duplicate
    return ''.join(stack)  # Convert the stack to a string
and return

# Driver code
string1 = "geeksforgeeg"
print(remove_adjacent_duplicates(string1))

string1 = "azxxxzy"
print(remove_adjacent_duplicates(string1))
string2 = "acbbcddc"
print(remove_adjacent_duplicates(string2))
```