

What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-orientated way or a functional way.

Good to know

- The most recent major version of Python is Python 3. However, Python 2, although not being updated with anything other than security updates, is still quite popular.

- It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

Python Syntax compared to other programming languages

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

Comments:

This is a single line comment

''' This is a multi
Line comment'''

Print:

```
print ("Hello world!")
```

```
print ("Hello world # how are you")    # here is ignored because it is used as a string
```

```
print (10 * 'Shyam')
```

```
print ("shyam's laptop")
```

```
print('shyam\'s "laptop" ')
```

```
print('c:\docs\naive')
```

```
print(r'c:\docs\naive') #r means raw string
```

Math:

```
print ("Hens", 25 + 30 / 6)
```

```
print ("Roosters", 100 - 25 * 3 % 4)  % = modulo. "X divided by Y with J
remaining." For example, "100 divided by 16 with 4 remaining." The result of %
is the J part, or the remaining part.
```

```
print (3 + 2 + 1 - 5 + 4 % 2 - 1 / 4 + 6)
```

```
print ("Is it true that 3 + 2 < 5 - 7?")
```

```
print (3 + 2 < 5 - 7)
```

```
print ("What is 3 + 2?", 3 + 2)
```

```
print ("What is 5 - 7?", 5 - 7)
```

```
print ("Is it greater?", 5 > - 2)
```

```
print ("Is it greater or equal?", 5 >= - 2)
```

```
print ("Is it less or equal?", 5 <= -2)
```

PEMDAS, which stands for Parentheses Exponents Multiplication Division Addition Subtraction. That's the order Python follows as well.

How to round a floating point number?

You can use the round() function like this: round(1.7333).

Python Numbers

There are three numeric types in Python:

- int
- float
- complex

Variables of numeric types are created when you assign a value to them:

Example

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex

print(type(x))
print(type(y))
print(type(z))
```

Python Casting

Specify a Variable Type

There may be times when you want to specify a type on to a variable. This can be done with casting. Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types.

Casting in python is therefore done using constructor functions:

- `int()` - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number), or a string literal (providing the string represents a whole number)
- `float()` - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- `str()` - constructs a string from a wide variety of data types, including strings, integer literals and float literals

Example

Integers:

```
x = int(1)    # x will be 1
y = int(2.8)  # y will be 2
z = int("3")  # z will be 3
```

Floats:

```
x = float(1)    # x will be 1.0
y = float(2.8)  # y will be 2.8
```

```
z = float("3")    # z will be 3.0
w = float("4.2")  # w will be 4.2
```

Strings:

```
x = str("s1") # x will be 's1'
y = str(2)    # y will be '2'
z = str(3.0)  # z will be '3.0'
```

Variables and names:

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables).

```
cars = 100
space_in_a_car = 4.0
drivers = 30
passengers = 90
cars_not_driven = cars - drivers
cars_driven = drivers
carpool_capacity = cars_driven * space_in_a_car
average_passengers_per_car = passengers / cars_driven
print ("There are", cars, "cars available.")
print ("There are only", drivers, "drivers available.")
```

```
print ("There will be", cars_not_driven, "empty cars today.")
print ("We can transport", carpool_capacity, "people today.")
print ("We have", passengers, "to carpool today.")
print ("We need to put about", average_passengers_per_car, "in each car.")
```

```
x = 6
y = 5
x + 10
_ + y    #_ represents output of previous number
```

More variables and printing:

What does %s, %r, and %d do?

They are “formatters.” They tell Python to take the variable on the right and put it in to replace the %s with its value.

%s - String (or any object with a string representation, like numbers)

%d - Integers

%f - Floating point numbers

%.<number of digits>f - Floating point numbers with a fixed amount of digits to the right of the dot.

%x/%X - Integers in hex representation (lowercase/uppercase)

%r - is the Python “representation” for the object, a string which, if presented to a Python interpreter should be parsed as a literal or as an instantiation of a new object of that time and with the same value.

```
my_name = 'Zed A. Shaw'
my_age = 35 # not a lie
my_height = 74 # inches
my_weight = 180 # lbs
```

```
my_eyes = 'Blue'
my_teeth = 'White'
my_hair = 'Brown'
print ("Let's talk about %s." % my_name)
print ("Let's talk about %s.", my_name)
print ("He's %d inches tall." % my_height)
print ("He's %d pounds heavy." % my_weight)
print ("Actually that's not too heavy.")
print ("He's got %s eyes and %s hair." % (my_eyes, my_hair))
print ("His teeth are usually %s depending on the coffee." % my_teeth)

# this line is tricky, try to get it exactly right
print ("If I add %d, %d, and %d I get %d." % (
my_age, my_height, my_weight, my_age + my_height + my_weight))
```

What is the difference between %r and %s?

We use %r for debugging, since it displays the “raw” data of the variable, but we use %s and others for displaying to users.

What’s the point of %s and %d when you can just use %r?

The %r is best for debugging, and the other formats are for actually displaying variables to users.

The %s specifier converts the object using `str()` and %r converts it using `repr()` (representation).

For some objects such as integers, they yield the same result, but `repr()` is special in that (for types where this is possible) it conventionally returns a result that is valid Python syntax, which could be used to unambiguously recreate the object it represents.

Here's an example, using a date:

```
>>> import datetime
>>> d = datetime.date.today()
>>> str(d)
```

```
'2011-05-14'  
>>> repr(d)  
'datetime.date(2011, 5, 14)'
```

Use the %r for debugging, since it displays the "raw" data of the variable, but the others are used for displaying to users.

That's how %r formatting works; it prints it the way you wrote it (or close to it). It's the "raw" format for debugging. Here \n used to display to users doesn't work. %r shows the representation of the raw data of the variable.

```
months = "\nJan\nFeb\nMar\nApr\nMay\nJun\nJul\nAug"
```

```
print "Here are the months: %r" % months
```

Output:

```
Here are the months: '\nJan\nFeb\nMar\nApr\nMay\nJun\nJul\nAug'
```

Strings and text:

String literals in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello".

Strings can be output to screen using the print function. For example: `print("hello")`.

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters. However, Python does not have a character data type, a single character is simply a string with a length of 1. Square brackets can be used to access elements of the string.

Example

Get the character at position 1 (remember that the first character has the position 0):

```
a = "Hello, World!"  
print(a[1])
```

Substring. Get the characters from position 2 to position 5 (not included):


```
b = "Hello, World!"  
print(b[2:5])
```

The strip() method removes any whitespace from the beginning or the end:

```
a = " Hello, World! "  
print(a.strip()) # returns "Hello, World!"
```

```
a = "Hello, World!"  
print(len(a))
```

The lower() method returns the string in lower case:

```
a = "Hello, World!"  
print(a.lower())
```

The upper() method returns the string in upper case:

```
a = "Hello, World!"  
print(a.upper())
```

The replace() method replaces a string with another string:

```
a = "Hello, World!"  
print(a.replace("H", "J"))
```

The split() method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"  
print(a.split(",")) # returns ['Hello', ' World!']
```

```
x = "There are %d types of people." % 10
binary = "binary"
do_not = "don't"
y = "Those who know %s and those who %s." % (binary, do_not)
print (x)
print (y)
print ("I said: %r." % x)
print ("I also said: '%s'." % y)
hilarious = False
joke_evaluation = "Isn't that joke so funny?! %r"
print (joke_evaluation % hilarious)
w = "This is the left side of..."
e = "a string with a right side."
print (w + e)
```

More printing:

```
formatter = "%r %r %r %r"
print (formatter % (1, 2, 3, 4))
print (formatter % ("one", "two", "three", "four"))
print (formatter % (True, False, False, True))
print (formatter % (formatter, formatter, formatter, formatter))
print (formatter % (
    "I had this thing.",
    "That you could type up right.",
    "But it didn't sing.",
    "So I said goodnight."))
```

Why do I have to put quotes around "one" but not around True or False?

That's because Python recognizes True and False as keywords representing the concept of true and false. If you put quotes around them, then they are turned into strings and won't work right

I tried putting Chinese (or some other non- ASCII characters) into these strings, but %r prints out weird symbols.

Use %s to print that instead and it'll work.

Why does %r sometimes print things with single- quotes when I wrote them with double- quotes?

Python is going to print the strings in the most efficient way it can, not replicate exactly the way you wrote them. This is perfectly fine since %r is used for debugging and inspection, so it's not necessary that it be pretty

More printing:

```
days = "Mon Tue Wed Thu Fri Sat Sun"
months = "Jan\nFeb\nMar\nApr\nMay\nJun\nJul\nAug"
print( "Here are the days: ", days)
print( "Here are the months: ", months)
print( """
There's something going on here.
With the three double- quotes.
We'll be able to type as much as we like.
Even 4 lines if we want, or 5, or 6.
""")
```

Why do the \n newlines not work when I use %r?

That's how %r formatting works; it prints it the way you wrote it (or close to it). It's the "raw" format for debugging.

Why do I get an error when I put spaces between the three double-quotes?

You have to type them like `"""` and not `" "`, meaning with no spaces between each one.

Escape sequences:

This use of the `\` (backslash) character is a way we can put difficult- to- type characters into a string.

There are plenty of these “escape sequences” available for different characters you might want to

put in, but there’s a special one, the double backslash, which is just two of them `\`. These two

characters will print just one backslash.

Another important escape sequence is to escape a single- quote `'` or double- quote `"`. Imagine you have a string that uses double- quotes and you want to put a double- quote in for the output.

If you do this `"I "understand" joe."` then Python will get confused since it will think the `"` around `"understand"` actually ends the string. You need a way to tell Python that the `"` inside the string isn’t a real double- quote.

To solve this problem, you escape double- quotes and single- quotes so Python knows what to include in the string. Here’s an example:

```
"I am 6'2\" tall." # escape double- quote inside string
```

```
'I am 6\'2" tall.' # escape single- quote inside string
```

The second way is by using triple- quotes, which is just `"""` and works like a string, but you also can put as many lines of text as you want until you type `"""` again.

```
tabby_cat = "\tI'm tabbed in."
```

```
persian_cat = "I'm split\n on a line."
```

```
backslash_cat = "I'm \\ a \\ cat."
```

```
fat_cat = """
```

```
I'll do a list:
```

```
\t* Cat food
```

```
\t* Fishies
```

```
\t* Catnip\n\t* Grass
```

```
"""
```

```
print (tabby_cat)
```

```
print (persian_cat)
```

```
print (backslash_cat)
```

```
print (fat_cat)
```

This is the list of all the escape sequences Python supports. You may not use many of these, but

memorize their format and what they do anyway. Also try them out in some strings to see if you

can make them work.

Escape	What it does.
\\	Backslash (\)
\'	Single- quote (')
\"	Double- quote (")
\a	ASCII bell (BEL)
\b	ASCII backspace (BS)
\f	ASCII form feed (FF)
\n	ASCII linefeed (LF)
\N{name} (Unicode only)	Character named name in the Unicode database
\r	ASCII carriage return (CR)
\t	ASCII horizontal tab (TAB)
\uxxxx	Character with 16- bit hex value xxxx (Unicode only)
\Uxxxxxxxx	Character with 32- bit hex value xxxxxxxx (Unicode only)
\v	ASCII vertical tab (VT)
\ooo	Character with octal value oo
\xhh	Character with hex value hh

What happens when we execute following code:

```
while True:
    for i in ["/", "- ", "|", "\\ ", "|"]:
        print ("%s\r" % i,)
```

When I write // or /n it doesn't work.

That's because you are using a forward- slash / and not a backslash \. They are different characters that do very different things.

When I use a %r format none of the escape sequences work.

That's because %r is printing out the raw representation of what you typed, which is going to include the original escape sequences. Use %s instead. Always remember this: %r is for debugging; %s is for displaying.

Taking input from user

```
print ("How old are you?")
age = input()
print ("How tall are you?")
height = input()
print ("How much do you weigh?")
weight = input()
print ("So, you're %r old, %r tall and %r heavy." % (age, height, weight) )
```

We can also take input to prompt user like:

```
age = input("How old are you? ")
height = input("How tall are you? ")
weight = input("How much do you weigh? ")
print ("So, you're %r old, %r tall and %r heavy." % (age, height, weight))
```

Python Collections (Arrays)

There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is unordered, changeable and indexed. No duplicate members.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

List

A list is a collection which is ordered and changeable. In Python lists are written with square brackets.

Example

Create a List:

```
thislist = ["apple", "banana", "cherry"]  
print(thislist)
```

Access Items

You access the list items by referring to the index number:

Example

Print the second item of the list:

```
thislist = ["apple", "banana", "cherry"]  
print(thislist[1])
```

Change Item Value

To change the value of a specific item, refer to the index number:

Example

Change the second item:

```
thislist = ["apple", "banana", "cherry"]  
thislist[1] = "blackcurrant"  
print(thislist)
```

Loop Through a List

You can loop through the list items by using a `for` loop:

Example

Print all items in the list, one by one:

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

Check if Item Exists

To determine if a specified item is present in a list use the `in` keyword:

Example

Check if "apple" is present in the list:

```
thislist = ["apple", "banana", "cherry"]  
if "apple" in thislist:  
    print("Yes, 'apple' is in the fruits list")
```


List Length

To determine how many items a list has, use the `len()` method:

Example

Print the number of items in the list:

```
thislist = ["apple", "banana", "cherry"]  
print(len(thislist))
```

Add Items

To add an item to the end of the list, use the `append()` method:

Example

Using the `append()` method to append an item:

```
thislist = ["apple", "banana", "cherry"]  
thislist.append("orange")  
print(thislist)
```

To add an item at the specified index, use the `insert()` method:

Example

Insert an item as the second position:

```
thislist = ["apple", "banana", "cherry"]  
thislist.insert(1, "orange")  
print(thislist)
```

Remove Item

There are several methods to remove items from a list:

Example

The `remove()` method removes the specified item:

```
thislist = ["apple", "banana", "cherry"]  
thislist.remove("banana")  
print(thislist)
```

Example

The `pop()` method removes the specified index, (or the last item if index is not specified):

```
thislist = ["apple", "banana", "cherry"]  
thislist.pop()  
print(thislist)
```

Example

The `del` keyword removes the specified index:

```
thislist = ["apple", "banana", "cherry"]  
del thislist[0]  
print(thislist)
```

Example

The `del` keyword can also delete the list completely:

```
thislist = ["apple", "banana", "cherry"]  
del thislist
```

Example

The `clear()` method empties the list:

```
thislist = ["apple", "banana", "cherry"]  
thislist.clear()  
print(thislist)
```

Copy a List

You cannot copy a list simply by typing `list2 = list1`, because: `list2` will only be a *reference* to `list1`, and changes made in `list1` will automatically also be made in `list2`.

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

Example

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

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

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

Example

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

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

The list() Constructor

It is also possible to use the `list()` constructor to make a new list.

Example

Using the `list()` constructor to make a List:

```
thislist = list(("apple", "banana", "cherry")) # note the double round-
brackets
print(thislist)
```

List Methods

Python has a set of built-in methods that you can use on lists.

Method	Description
<code>append()</code>	Adds an element at the end of the list
<code>clear()</code>	Removes all the elements from the list
<code>copy()</code>	Returns a copy of the list
<code>count()</code>	Returns the number of elements with the specified value
<code>extend()</code>	Add the elements of a list (or any iterable), to the end of the current list
<code>index()</code>	Returns the index of the first element with the specified value
<code>insert()</code>	Adds an element at the specified position
<code>pop()</code>	Removes the element at the specified position

[`remove\(\)`](#) Removes the item with the specified value

[`reverse\(\)`](#) Reverses the order of the list

[`sort\(\)`](#) Sorts the list

Tuple

A tuple is a collection which is ordered and **unchangeable**. In Python tuples are written with round brackets.

Example

Create a Tuple:

```
thistuple = ("apple", "banana", "cherry")  
print(thistuple)
```

Access Tuple Items

You can access tuple items by referring to the index number, inside square brackets:

Example

Return the item in position 1:

```
thistuple = ("apple", "banana", "cherry")  
print(thistuple[1])
```

Change Tuple Values

Once a tuple is created, you cannot change its values. Tuples are **unchangeable**.

Loop Through a Tuple

You can loop through the tuple items by using a `for` loop.

Example

Iterate through the items and print the values:

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
    print(x)
```

Check if Item Exists

To determine if a specified item is present in a tuple use the `in` keyword:

Example

Check if "apple" is present in the tuple:

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
    print("Yes, 'apple' is in the fruits tuple")
```

Tuple Length

To determine how many items a tuple has, use the `len()` method:

Example

Print the number of items in the tuple:

```
thistuple = ("apple", "banana", "cherry")  
print(len(thistuple))
```

Add Items

Once a tuple is created, you cannot add items to it. Tuples are **unchangeable**.

Example

You cannot add items to a tuple:

```
thistuple = ("apple", "banana", "cherry")  
thistuple[3] = "orange" # This will raise an error  
print(thistuple)
```

Remove Items

Note: You cannot remove items in a tuple.

Tuples are **unchangeable**, so you cannot remove items from it, but you can delete the tuple completely:

Example

The **del** keyword can delete the tuple completely:

```
thistuple = ("apple", "banana", "cherry")  
del thistuple  
print(thistuple) #this will raise an error because the tuple no longer  
exists
```

The tuple() Constructor

It is also possible to use the **tuple()** constructor to make a tuple.

Example

Using the tuple() method to make a tuple:

```
thistuple = tuple(("apple", "banana", "cherry")) # note the double  
round-brackets  
print(thistuple)
```

Tuple Methods

Python has two built-in methods that you can use on tuples.

Method	Description
<u>count()</u>	Returns the number of times a specified value occurs in a tuple
<u>index()</u>	Searches the tuple for a specified value and returns the position of where it was found

Set

A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.

Example

Create a Set:

```
thisset = {"apple", "banana", "cherry"}  
print(thisset)
```

Note: Sets are unordered, so the items will appear in a random order.

Access Items

You cannot access items in a set by referring to an index, since sets are unordered the items has no index.

But you can loop through the set items using a `for` loop, or ask if a specified value is present in a set, by using the `in` keyword.

Example

Loop through the set, and print the values:

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
    print(x)
```

Example

Check if "banana" is present in the set:

```
thisset = {"apple", "banana", "cherry"}

print("banana" in thisset)
```

Change Items

Once a set is created, you cannot change its items, but you can add new items.

Add Items

To add one item to a set use the `add()` method.

To add more than one item to a set use the `update()` method.

Example

Add an item to a set, using the `add()` method:

```
thisset = {"apple", "banana", "cherry"}  
  
thisset.add("orange")  
  
print(thisset)
```

Example

Add multiple items to a set, using the `update()` method:

```
thisset = {"apple", "banana", "cherry"}  
  
thisset.update(["orange", "mango", "grapes"])  
  
print(thisset)
```

Get the Length of a Set

To determine how many items a set has, use the `len()` method.

Example

Get the number of items in a set:

```
thisset = {"apple", "banana", "cherry"}  
  
print(len(thisset))
```

Remove Item

To remove an item in a set, use the `remove()`, or the `discard()` method.

Example

Remove "banana" by using the `remove()` method:

```
thisset = {"apple", "banana", "cherry"}  
  
thisset.remove("banana")  
  
print(thisset)
```

Note: If the item to remove does not exist, `remove()` will raise an error.

Example

Remove "banana" by using the `discard()` method:

```
thisset = {"apple", "banana", "cherry"}  
  
thisset.discard("banana")  
  
print(thisset)
```

Note: If the item to remove does not exist, `discard()` will **NOT** raise an error.

You can also use the `pop()` method to remove an item, but this method will remove the *last* item. Remember that sets are unordered, so you will not know what item that gets removed.

The return value of the `pop()` method is the removed item.

Example

Remove the last item by using the `pop()` method:

```
thisset = {"apple", "banana", "cherry"}  
  
x = thisset.pop()  
  
print(x)  
  
print(thisset)
```

Note: Sets are *unordered*, so when using the `pop()` method, you will not know which item that gets removed.

Example

The `clear()` method empties the set:

```
thisset = {"apple", "banana", "cherry"}  
  
thisset.clear()  
  
print(thisset)
```

Example

The `del` keyword will delete the set completely:

```
thisset = {"apple", "banana", "cherry"}  
  
del thisset  
  
print(thisset)
```

The set() Constructor

It is also possible to use the `set()` constructor to make a set.

Example

Using the `set()` constructor to make a set:

```
thisset = set(("apple", "banana", "cherry")) # note the double round-  
brackets  
print(thisset)
```

Set Methods

Python has a set of built-in methods that you can use on sets.

Method	Description
<code>add()</code>	Adds an element to the set
<code>clear()</code>	Removes all the elements from the set
<code>copy()</code>	Returns a copy of the set
<code>difference()</code>	Returns a set containing the difference between two or more sets
<code>difference_update()</code>	Removes the items in this set that are also included in another, specified set
<code>discard()</code>	Remove the specified item
<code>intersection()</code>	Returns a set, that is the intersection of two other sets
<code>intersection_update()</code>	Removes the items in this set that are not present in other, specified set(s)
<code>isdisjoint()</code>	Returns whether two sets have a intersection or not

<u>issubset()</u>	Returns whether another set contains this set or not
<u>issuperset()</u>	Returns whether this set contains another set or not
<u>pop()</u>	Removes an element from the set
<u>remove()</u>	Removes the specified element
<u>symmetric_difference()</u>	Returns a set with the symmetric differences of two sets
<u>symmetric_difference_update()</u>	inserts the symmetric differences from this set and another
<u>union()</u>	Return a set containing the union of sets
<u>update()</u>	Update the set with the union of this set and others

Dictionary

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

Example

Create and print a dictionary:

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

Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets:

Example

Get the value of the "model" key:

```
x = thisdict["model"]
```

There is also a method called `get()` that will give you the same result:

Example

Get the value of the "model" key:

```
x = thisdict.get("model")
```

Change Values

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

Example

Change the "year" to 2018:

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

Loop Through a Dictionary

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

When looping through a dictionary, the return value are the *keys* of the dictionary, but there are methods to return the *values* as well.

Example

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

```
for x in thisdict:  
    print(x)
```

Example

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

```
for x in thisdict:  
    print(thisdict[x])
```

Example

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

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

Example

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


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

Check if Key Exists

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

Example

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")
```

Dictionary Length

To determine how many items (key-value pairs) a dictionary has, use the `len()` method.

Example

Print the number of items in the dictionary:

```
print(len(thisdict))
```

Adding Items

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

Example

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

Removing Items

There are several methods to remove items from a dictionary:

Example

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

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

Example

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)
```

Example

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

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

Example

The `del` keyword can also delete the dictionary completely:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
del thisdict  
print(thisdict) #this will cause an error because "thisdict" no longer  
exists.
```

Example

The `clear()` keyword empties the dictionary:

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

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()`.

Example

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 method `dict()`.

Example

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

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

The dict() Constructor

It is also possible to use the `dict()` constructor to make a new dictionary:

Example

```
thisdict = dict(brand="Ford", model="Mustang", year=1964)  
# note that keywords are not string literals  
# note the use of equals rather than colon for the assignment  
print(thisdict)
```

Dictionary Methods

Python has a set of built-in methods that you can use on dictionaries.

Method	Description
<code>clear()</code>	Removes all the elements from the dictionary
<code>copy()</code>	Returns a copy of the dictionary
<code>fromkeys()</code>	Returns a dictionary with the specified keys and values
<code>get()</code>	Returns the value of the specified key
<code>items()</code>	Returns a list containing the a tuple for each key value pair
<code>keys()</code>	Returns a list containing the dictionary's keys
<code>pop()</code>	Removes the element with the specified key
<code>popitem()</code>	Removes the last inserted key-value pair
<code>setdefault()</code>	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 Conditions and If statements

Python supports the usual logical conditions from mathematics:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`

These conditions can be used in several ways, most commonly in "if statements" and loops.

An "if statement" is written by using the `if` keyword.

Example

If statement:

```
a = 33
b = 200
if b > a:
    print("b is greater than a")
```

In this example we use two variables, `a` and `b`, which are used as part of the if statement to test whether `b` is greater than `a`. As `a` is 33, and `b` is 200, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".

Indentation

Python relies on indentation, using whitespace, to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Example

If statement, without indentation (will raise an error):

```
a = 33
b = 200
if b > a:
print("b is greater than a") # you will get an error
```

Elif

The `elif` keyword is python's way of saying "if the previous conditions were not true, then try this condition".

Example

```
a = 33
b = 33
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal")
```

In this example `a` is equal to `b`, so the first condition is not true, but the `elif` condition is true, so we print to screen that "a and b are equal".

Else

The `else` keyword catches anything which isn't caught by the preceding conditions.

Example

```
a = 200
b = 33
if b > a:
```

```
    print("b is greater than a")
elif a == b:
    print("a and b are equal")
else:
    print("a is greater than b")
```

In this example `a` is greater to `b`, so the first condition is not true, also the `elif` condition is not true, so we go to the `else` condition and print to screen that "a is greater than b".

You can also have an `else` without the `elif`:

Example

```
a = 200
b = 33
if b > a:
    print("b is greater than a")
else:
    print("b is not greater than a")
```

Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

Example

One line if statement:

```
if a > b: print("a is greater than b")
```

Short Hand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

Example

One line if else statement:

```
print("A") if a > b else print("B")
```

You can also have multiple else statements on the same line:

Example

One line if else statement, with 3 conditions:

```
print("A") if a > b else print("=") if a == b else print("B")
```

And

The `and` keyword is a logical operator, and is used to combine conditional statements:

Example

Test if `a` is greater than `b`, AND if `c` is greater than `a`:

```
if a > b and c > a:  
    print("Both conditions are True")
```

Or

The `or` keyword is a logical operator, and is used to combine conditional statements:

Example

Test if `a` is greater than `b`, OR if `a` is greater than `c`:

```
if a > b or a > c:  
    print("At least one of the conditions is True")
```

Python Loops

Python has two primitive loop commands:

- `while` loops
- `for` loops

The while Loop

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

Example

Print i as long as i is less than 6:

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

Note: remember to increment i, or else the loop will continue forever.

The `while` loop requires relevant variables to be ready, in this example we need to define an indexing variable, `i`, which we set to 1.

The break Statement

With the `break` statement we can stop the loop even if the while condition is true:

Example

Exit the loop when i is 3:

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

```
if i == 3:
    break
i += 1
```

The continue Statement

With the `continue` statement we can stop the current iteration, and continue with the next:

Example

Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)
```

Python For Loops

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

This is less like the `for` keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the `for` loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Example

Print each fruit in a fruit list:

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

The `for` loop does not require an indexing variable to set beforehand.

Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

Example

Loop through the letters in the word "banana":

```
for x in "banana":  
    print(x)
```

The break Statement

With the `break` statement we can stop the loop before it has looped through all the items:

Example

Exit the loop when `x` is "banana":

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

Example

Exit the loop when `x` is "banana", but this time the break comes before the print:

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

The continue Statement

With the `continue` statement we can stop the current iteration of the loop, and continue with the next:

Example

Do not print banana:

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

The range() Function

To loop through a set of code a specified number of times, we can use the `range()` function,

The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example

Using the range() function:

```
for x in range(6):
    print(x)
```

Note that `range(6)` is not the values of 0 to 6, but the values 0 to 5.

The `range()` function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: `range(2, 6)`, which means values from 2 to 6 (but not including 6):

Example

Using the start parameter:

```
for x in range(2, 6):  
    print(x)
```

The `range()` function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: `range(2, 30, 3)`:

Example

Increment the sequence with 3 (default is 1):

```
for x in range(2, 30, 3):  
    print(x)
```

Else in For Loop

The `else` keyword in a `for` loop specifies a block of code to be executed when the loop is finished:

Example

Print all numbers from 0 to 5, and print a message when the loop has ended:

```
for x in range(6):  
    print(x)  
else:  
    print("Finally finished!")
```

Nested Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

Example

Print each adjective for every fruit:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
    for y in fruits:
        print(x, y)
```

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:

Example

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

Calling a Function

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

Example

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

```
my_function()
```

Parameters

Information can be passed to functions as parameter.

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

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

Example

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

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

Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without parameter, it uses the default value:

Example

```
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 a Parameter

You can send any data types of parameter 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 a parameter, it will still be a List when it reaches the function:

Example

```
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:

Example

```
def my_function(x):  
    return 5 * x
```

```
print(my_function(3))  
print(my_function(5))  
print(my_function(9))
```

Recursion

Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

In this example, `tri_recursion()` is a function that we have defined to call itself ("recurse"). We use the `k` variable as the data, which decrements (`-1`) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

Example

Recursion Example

```
def tri_recursion(k):
    if(k>0):
        result = k+tri_recursion(k-1)
        print(result)
    else:
        result = 0
    return result

print("\n\nRecursion Example Results")
tri_recursion(6)
```

Python Lambda

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

Syntax

`lambda arguments : expression`

The expression is executed and the result is returned:

Example

A lambda function that adds 10 to the number passed in as an argument, and print the result:

```
x = lambda a : a + 10
print(x(5))
```

Lambda functions can take any number of arguments:

Example

A lambda function that multiplies argument a with argument b and print the result:

```
x = lambda a, b : a * b
print(x(5, 6))
```

Example

A lambda function that sums argument a, b, and c and print the result:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

Why Use Lambda Functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
    return lambda a : a * n
```

Use that function definition to make a function that always doubles the number you send in:

Example

```
def myfunc(n):  
    return lambda a : a * n
```

```
mydoubler = myfunc(2)
```

```
print(mydoubler(11))
```

Or, use the same function definition to make a function that always *triples* the number you send in:

Example

```
def myfunc(n):  
    return lambda a : a * n
```

```
mytripler = myfunc(3)
```

```
print(mytripler(11))
```

Or, use the same function definition to make both functions, in the same program:

Example

```
def myfunc(n):  
    return lambda a : a * n
```

```
mydoubler = myfunc(2)
```

```
mytripler = myfunc(3)
```

```
print(mydoubler(11))
```

```
print(mytripler(11))
```

Use lambda functions when an anonymous function is required for a short period of time.

Arrays

Arrays are used to store multiple values in one single variable:

Example

Create an array containing car names:

```
cars = ["Ford", "Volvo", "BMW"]
```

What is an Array?

An array is a special variable, which can hold more than one value at a time.

If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

```
car1 = "Ford"  
car2 = "Volvo"  
car3 = "BMW"
```

However, what if you want to loop through the cars and find a specific one? And what if you had not 3 cars, but 300?

The solution is an array!

An array can hold many values under a single name, and you can access the values by referring to an index number.

They can be useful when we have to manipulate only a specific data type values.

Access the Elements of an Array

You refer to an array element by referring to the *index number*.

Example

Get the value of the first array item:

```
x = cars[0]
```

Example

Modify the value of the first array item:

```
cars[0] = "Toyota"
```

The Length of an Array

Use the `len()` method to return the length of an array (the number of elements in an array).

Example

Return the number of elements in the `cars` array:

```
x = len(cars)
```

Note: The length of an array is always one more than the highest array index.

Looping Array Elements

You can use the `for in` loop to loop through all the elements of an array.

Example

Print each item in the `cars` array:

```
for x in cars:  
    print(x)
```

Adding Array Elements

You can use the `append()` method to add an element to an array.

Example

Add one more element to the `cars` array:

```
cars.append("Honda")
```

Removing Array Elements

You can use the `pop()` method to remove an element from the array.

Example

Delete the second element of the `cars` array:

```
cars.pop(1)
```

You can also use the `remove()` method to remove an element from the array.

Example

Delete the element that has the value "Volvo":

```
cars.remove("Volvo")
```

Note: The `remove()` method only removes the first occurrence of the specified value.

Array Methods

Python has a set of built-in methods that you can use on lists/arrays.

Method	Description
--------	-------------

[append\(\)](#) Adds an element at the end of the list

[clear\(\)](#) Removes all the elements from the list

[copy\(\)](#) Returns a copy of the list

[count\(\)](#) Returns the number of elements with the specified value

[extend\(\)](#) Add the elements of a list (or any iterable), to the end of the current list

[index\(\)](#) Returns the index of the first element with the specified value

[insert\(\)](#) Adds an element at the specified position

[pop\(\)](#) Removes the element at the specified position

[remove\(\)](#) Removes the first item with the specified value

[reverse\(\)](#) Reverses the order of the list

[sort\(\)](#) Sorts the list

Other way to use Arrays

```
from array import *  
Vals = array('i', [3,4,5,6,9])  
print(Vals)
```

Operations on Array:

1. array (data type, value list): - This function is used to **create** an array with data type and value list specified in its arguments. Some of the data types are mentioned in the table below.

TYPE CODE	C TYPE	PYTHON TYPE	MINIMUM SIZE IN BYTES
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4

TYPE CODE	C TYPE	PYTHON TYPE	MINIMUM SIZE IN BYTES
'L'	unsigned long	int	4
'q'	signed long	int	8
'Q'	unsigned long	int	8
'f'	Float	float	4
'd'	Double	float	8

2. append() :- This function is used to **add the value** mentioned in its arguments at the **end** of the array.

3. insert(i,x) :- This function is used to **add the value at the position** specified in its argument.

4. pop() :- This function **removes the element at the position** mentioned in its argument, and returns it.

5. remove() :- This function is used to **remove the first occurrence** of the value mentioned in its arguments.

6. index() :- This function returns the **index of the first occurrence** of value mentioned in arguments.

7. reverse() :- This function **reverses** the array.

8. Copying an array

```
from array import *
vals = array ('i', [3,4,5,6,9])
newArrr = array (vals.typecode, (a for a in vals) )
for e in newArr:
    print(e)
```

Suppose we want to get the square of original numbers of vals

```
from array import *
vals = array ('i', [3,4,5,6,9])
newArrr = array (vals.typecode, (a*a for a in vals) )
for e in newArr:
    print(e)
```

Ex: Array values from user

```
from array import *
arr = array ('i', [])
n = int (input ("Enter the length of the array"))

for i in range(n):
    x = int (input ("Enter next value"))
    arr.append (x)

print(arr)
```

Python Classes/Objects

Python is an object-oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword `class`:

Example

Create a class named MyClass, with a property named x:

```
class MyClass:  
    x = 5
```

Create Object

Now we can use the class named myClass to create objects:

Example

Create an object named p1, and print the value of x:

```
p1 = MyClass()  
print(p1.x)
```

Dunder or magic methods in Python

Dunder or magic methods in Python are the methods having two prefix and suffix underscores in the method name. Dunder here means "Double Under (Underscores)". These are commonly used for operator overloading. Few examples for magic methods are: `__init__`, `__add__`, `__len__`, `__repr__` etc. The `__init__` method for initialization is invoked without any call, when an instance of a class is created, like constructors in certain other programming languages such as C++, Java, C#, PHP etc. These methods are the reason we can add two strings with '+' operator without any explicit typecasting. Here's a simple implementation :

```
# declare our own string class
class String:

    # magic method to initiate object
    def __init__(self, string):
        self.string = string

# Driver Code
if __name__ == '__main__':

    # object creation
    string1 = String('Hello')

    # print object location
    print(string1)
```

Output :

```
<__main__.String object at 0x7fe992215390>
```

The above snippet of code prints only the memory address of the string object. Let's add a `__repr__` method to represent our object.

```
# declare our own string class
class String:

    # magic method to initiate object
    def __init__(self, string):
        self.string = string
```

```
# print our string object
def __repr__(self):
    return 'Object: {}'.format(self.string)

# Driver Code
if __name__ == '__main__':

    # object creation
    string1 = String('Hello')

    # print object location
    print(string1)
```

Output :

```
Object: Hello
```

If we try to add a string to it :

```
# declare our own string class
class String:

    # magic method to initiate object
    def __init__(self, string):
        self.string = string

    # print our string object
    def __repr__(self):
        return 'Object: {}'.format(self.string)
```

```
# Driver Code
if __name__ == '__main__':

    # object creation
    string1 = String('Hello')

    # concatenate String object and a string
    print(string1 + ' world')
```

Output :

TypeError: unsupported operand type(s) for +: 'String' and 'str'

Now add `__add__` method to String class :

```
# declare our own string class
class String:

    # magic method to initiate object
    def __init__(self, string):
        self.string = string

    # print our string object
    def __repr__(self):
        return 'Object: {}'.format(self.string)

    def __add__(self, other):
        return self.string + other

# Driver Code
```

```
if __name__ == '__main__':  
  
    # object creation  
    string1 = String('Hello')  
  
    # concatenate String object and a string  
    print(string1 + ' Geeks')
```

Output :

```
Hello Geeks
```

The __init__() Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in __init__() function.

All classes have a function called __init__(), which is always executed when the class is being initiated.

Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Example

Create a class named Person, use the __init__() function to assign values for name and age:

```
class Person:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

```
p1 = Person("John", 36)
```



```
print(p1.name)
print(p1.age)
```

Note: The `__init__()` function is called automatically every time the class is being used to create a new object.

Object Methods

Objects can also contain methods. Methods in objects are functions that belongs to the object.

Let us create a method in the Person class:

Example

Insert a function that prints a greeting, and execute it on the p1 object:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def myfunc(self):
        print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

Note: The `self` parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

The self Parameter

The `self` parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It does not have to be named `self` , you can call it whatever you like, but it has to be the first parameter of any function in the class:

Example

Use the words *mysillyobject* and *abc* instead of *self*:

```
class Person:
    def __init__(mysillyobject, name, age):
        mysillyobject.name = name
        mysillyobject.age = age

    def myfunc(abc):
        print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

Modify Object Properties

You can modify properties on objects like this:

Example

Set the age of p1 to 40:

```
p1.age = 40
```

Delete Object Properties

You can delete properties on objects by using the `del` keyword:

Example

Delete the age property from the p1 object:

```
del p1.age
```

Delete Objects

You can delete objects by using the `del` keyword:

Example

Delete the p1 object:

```
del p1
```

Python Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

Create a Parent Class

When naming a class, we use capital letters in the starting and we don't use underscore like when we name variables. We use camel casing when defining class.

Any class can be a parent class, so the syntax is the same as creating any other class:

Example

Create a class named `Person`, with `firstname` and `lastname` properties, and a `printname` method:

```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
```

```
print(self.firstname, self.lastname)
```

#Use the Person class to create an object, and then execute the printname method:

```
x = Person("John", "Doe")  
x.printname()
```

Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

Example

Create a class named **Student**, which will inherit the properties and methods from the **Person** class:

```
class Student(Person):  
    pass
```

Note: Use the **pass** keyword when you do not want to add any other properties or methods to the class.

Now the Student class has the same properties and methods as the Person class.

Example

Use the **Student** class to create an object, and then execute the **printname** method:

```
x = Student("Mike", "Olsen")  
x.printname()
```

Add the __init__() Function

So far we have created a child class that inherits the properties and methods from its parent.

We want to add the `__init__()` function to the child class (instead of the `pass` keyword).

Note: The `__init__()` function is called automatically every time the class is being used to create a new object.

Example

Add the `__init__()` function to the `Student` class:

```
class Student(Person):
    def __init__(self, fname, lname):
        #add properties etc.
```

When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()` function.

Note: The child's `__init__()` function **overrides** the inheritance of the parent's `__init__()` function.

To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()` function:

Example

```
class Student(Person):
    def __init__(self, fname, lname):
        Person.__init__(self, fname, lname)
```

Now we have successfully added the `__init__()` function, and kept the inheritance of the parent class, and we are ready to add functionality in the `__init__()` function.

Add Properties

Example

Add a property called `graduationyear` to the `Student` class:

```
class Student(Person):
    def __init__(self, fname, lname):
```

```
Person.__init__(self, fname, lname)
self.graduationyear = 2019
```

In the example below, the year **2019** should be a variable, and passed into the **Student** class when creating student objects. To do so, add another parameter in the `__init__()` function:

Example

Add a **year** parameter, and pass the correct year when creating objects:

```
class Student(Person):
    def __init__(self, fname, lname, year):
        Person.__init__(self, fname, lname)
        self.graduationyear = year
```

```
x = Student("Mike", "Olsen", 2019)
```

Add Methods

Example

Add a method called **welcome** to the **Student** class:

```
class Student(Person):
    def __init__(self, fname, lname, year):
        Person.__init__(self, fname, lname)
        self.graduationyear = year

    def welcome(self):
        print("Welcome", self.firstname, self.lastname, "to the class of",
self.graduationyear)
```

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

Python Decorators

Now, what are decorators really? They “decorate” or “wrap” another function and let you execute code before and after the wrapped function runs.

Decorators allow you to define reusable building blocks that can change or extend the behavior of other functions. And they let you do that without permanently modifying the wrapped function itself. The function’s behavior changes only when it’s *decorated*.

Now what does the implementation of a simple decorator look like? In basic terms, a decorator is *a callable that takes a callable as input and returns another callable*.

The following function has that property and could be considered the simplest decorator one could possibly write:

```
def null_decorator(func):  
  
    return func
```

As you can see, `null_decorator` is a callable (it’s a function), it takes another callable as its input, and it returns the same input callable without modifying it. Let’s use it to *decorate* (or *wrap*) another function:

```
def greet():  
  
    return 'Hello!'  
  
greet = null_decorator(greet)  
  
>>> greet()  
  
'Hello!'
```

In this example I've defined a greet function and then immediately decorated it by running it through the null_decorator function. I know this doesn't look very useful yet (I mean we specifically designed the null decorator to be useless, right?) but in a moment it'll clarify how Python's decorator syntax works. Instead of explicitly calling null_decorator on greet and then reassigning the greet variable, you can use Python's @ syntax for decorating a function in one step:

```
@null_decorator
```

```
def greet():
```

```
    return 'Hello!'
```

```
>>> greet()
```

```
'Hello!'
```

Putting an @null_decorator line in front of the function definition is the same as defining the function first and then running through the decorator. Using the @syntax is just *syntactic sugar*, and a shortcut for this commonly used pattern.

Note that using the @ syntax decorates the function immediately at definition time. This makes it difficult to access the undecorated original without brittle hacks. Therefore, you might choose to decorate some functions manually in order to retain the ability to call the undecorated function as well.

Decorators Can Modify Behavior

Now that you're a little more familiar with the decorator syntax, let's write another decorator that *actually does something* and modifies the behavior of the decorated function.

Here's a slightly more complex decorator which converts the result of the decorated function to uppercase letters:

```
def uppercase(func):
```

```
    def wrapper():
```

```
        original_result = func()
```



```
modified_result = original_result.upper()

return modified_result

return wrapper
```

Instead of simply returning the input function like the null decorator did, this uppercase decorator defines a new function on the fly (a closure) and uses it to *wrap* the input function in order to modify its behavior at call time. The wrapper closure has access to the undecorated input function and it is free to execute additional code before and after calling the input function. (Technically, it doesn't even need to call the input function at all.) Note how up until now the decorated function has never been executed. Actually, calling the input function at this point wouldn't make any sense—you'll want the decorator to be able to modify the behavior of its input function when it gets called eventually.

Time to see the uppercase decorator in action. What happens if you decorate the original greet function with it?

```
@uppercase
```

```
def greet():

    return 'Hello!'
```

```
>>> greet()
```

```
'HELLO!'
```

I hope this was the result you expected. Let's take a closer look at what just happened here. Unlike `null_decorator`, our uppercase decorator returns a *different function object* when it decorates a function:

```
>>> greet
```

```
<function greet at 0x10e9f0950>
```

```
>>> null_decorator(greet)
```

```
<function greet at 0x10e9f0950>
```

```
>>> uppercase(greet)
```

```
<function uppercase.<locals>.wrapper at 0x10da02f28>
```

And as you saw earlier, it needs to do that in order to modify the behavior of the decorated function when it finally gets called. The uppercase decorator is a function itself. And the only way to influence the “future behavior” of an input function it decorates is to replace (or *wrap*) the input function with a closure. That’s why uppercase defines and returns another function (the closure) that can then be called at a later time, run the original input function, and modify its result.

Decorators modify the behavior of a callable through a wrapper so you don’t have to permanently modify the original. The callable isn’t permanently modified—its behavior changes only when decorated.

This lets you “tack on” reusable building blocks, like logging and other instrumentation, to existing functions and classes. It’s what makes decorators such a powerful feature in Python that’s frequently used in the standard library and in third-party packages.

Applying Multiple Decorators to a Single Function

Perhaps not surprisingly, you can apply more than one decorator to a function. This accumulates their effects and it’s what makes decorators so helpful as reusable building blocks.

Here’s an example. The following two decorators wrap the output string of the decorated function in HTML tags. By looking at how the tags are nested you can see which order Python uses to apply multiple decorators:

```
def strong(func):  
    def wrapper():  
        return '<strong>' + func() + '</strong>'  
    return wrapper
```

```
def emphasis(func):  
  
    def wrapper():  
  
        return '<em>' + func() + '</em>'  
  
    return wrapper
```

Now let's take these two decorators and apply them to our greet function at the same time. You can use the regular @ syntax for that and just "stack" multiple decorators on top of a single function:

```
@strong  
  
@emphasis  
  
def greet():  
  
    return 'Hello!'
```

What output do you expect to see if you run the decorated function? Will the @emphasis decorator add its tag first or does @strong have precedence? Here's what happens when you call the decorated function:

```
>>> greet()  
  
'<strong><em>Hello!</em></strong>'
```

This clearly shows in what order the decorators were applied: from *bottom to top*. First, the input function was wrapped by the @emphasis decorator, and then the resulting (decorated) function got wrapped again by the @strong decorator.

To help me remember this bottom to top order I like to call this behavior *decorator stacking*. You start building the stack at the bottom and then keep adding new blocks on top to work your way upwards.

If you break down the above example and avoid the @ syntax to apply the decorators, the chain of decorator function calls looks like this:

```
decorated_greet = strong(emphasis(greet))
```

Again, you can see here that the emphasis decorator is applied first and then the resulting wrapped function is wrapped again by the strong decorator. This also means that deep levels of decorator stacking will have an effect on performance eventually because they keep adding nested function calls. Usually this won't be a problem in practice, but it's something to keep in mind if you're working on performance intensive code.

Decorating Functions That Accept Arguments

All examples so far only decorated a simple *nullary* greet function that didn't take any arguments whatsoever. So the decorators you saw here up until now didn't have to deal with forwarding arguments to the input function. If you try to apply one of these decorators to a function that takes arguments it will not work correctly. How do you decorate a function that takes arbitrary arguments?

This is where Python's `*args` and `**kwargs` feature for dealing with variable numbers of arguments comes in handy.

`*args`

The special syntax `*args` in function definitions in python is used to pass a variable number of arguments to a function. It is used to pass a non-keyworded, variable-length argument list.

- The syntax is to use the symbol `*` to take in a variable number of arguments; by convention, it is often used with the word `args`.
- What `*args` allows you to do is take in more arguments than the number of formal arguments that you previously defined. With `*args`, any number of extra arguments can be tacked on to your current formal parameters (including zero extra arguments).
- For example : we want to make a multiply function that takes any number of arguments and able to multiply them all together. It can be done using `*args`.
- Using the `*`, the variable that we associate with the `*` becomes an iterable meaning you can do things like iterate over it, run some higher order functions such as `map` and `filter`, etc.

```
# Python program to illustrate
# *args for variable number of arguments
def myFun(*argv):
    for arg in argv:
        print (arg)

myFun('Hello', 'Fraands', 'chai', 'pi lo')
```

```

# Python program to illustrate
# *args with first extra argument

def myFun(arg1, *argv):
    print ("First argument :", arg1)

    for arg in argv:
        print("Next argument through *argv :", arg)

myFun('Hello', 'Fraands', 'chai', 'pi lo')

```

****kwargs**

The special syntax ***kwargs* in function definitions in python is used to pass a keyworded, variable-length argument list. We use the name *kwargs* with the double star. The reason is because the double star allows us to pass through keyword arguments (and any number of them).

- A keyword argument is where you provide a name to the variable as you pass it into the function.
- One can think of the *kwargs* as being a dictionary that maps each keyword to the value that we pass alongside it. That is why when we iterate over the *kwargs* there doesn't seem to be any order in which they were printed out.

```

# Python program to illustrate
# *kwargs for variable number of keyword arguments

def myFun(**kwargs):
    for key, value in kwargs.items():
        print ("%s == %s" %(key, value))

# Driver code
myFun(first='I', mid='like', last='tea')

```

```

# Python program to illustrate **kwargs for
# variable number of keyword arguments with
# one extra argument.

def myFun(arg1, **kwargs):
    for key, value in kwargs.items():
        print ("%s == %s" %(key, value))

# Driver code

myFun("I", first = 'like', mid = 'tea', last='very much lol')

```

The following proxy decorator takes advantage of *args and **kwargs:

```

def proxy(func):

    def wrapper(*args, **kwargs):

        return func(*args, **kwargs)

    return wrapper

```

There are two notable things going on with this decorator:

- It uses the * and ** operators in the wrapper closure definition to collect all positional and keyword arguments and stores them in variables (args and kwargs).
- The wrapper closure then forwards the collected arguments to the original input function using the * and ** “argument unpacking” operators. (It’s a bit unfortunate that the meaning of the star and double-star operators is overloaded and changes depending on the context they’re used in. But I hope you get the idea.)

Let's expand the technique laid out by the proxy decorator into a more useful practical example. Here's a trace decorator that logs function arguments and results during execution time:

```
def trace(func):  
  
    def wrapper(*args, **kwargs):  
  
        print(f'TRACE: calling {func.__name__}() '  
              f'with {args}, {kwargs}')  
        original_result = func(*args, **kwargs)  
  
        print(f'TRACE: {func.__name__}() '  
              f'returned {original_result!r}')  
        return original_result  
  
    return wrapper
```

Decorating a function with trace and then calling it will print the arguments passed to the decorated function and its return value. This is still somewhat of a toy example—but in a pinch it makes a great debugging aid:

```
@trace  
  
def say(name, line):  
  
    return f'{name}: {line}'  
  
>>> say('Jane', 'Hello, World')  
  
'TRACE: calling say() with ("Jane", "Hello, World"), {}'  
  
'TRACE: say() returned "Jane: Hello, World"'  
  
'Jane: Hello, World'
```

Speaking of debugging—there are some things you should keep in mind when debugging decorators:

How to Write “Debuggable” Decorators

When you use a decorator, really what you’re doing is replacing one function with another. One downside of this process is that it “hides” some of the metadata attached to the original (undecorated) function.

For example, the original function name, its docstring, and parameter list are hidden by the wrapper closure:

```
def greet():  
    """Return a friendly greeting."""  
    return 'Hello!'  
  
decorated_greet = uppercase(greet)
```

If you try to access any of that function metadata, you’ll see the wrapper closure’s metadata instead:

```
>>> greet.__name__  
'greet'  
  
>>> greet.__doc__  
'Return a friendly greeting.'  
  
>>> decorated_greet.__name__  
'wrapper'  
  
>>> decorated_greet.__doc__  
None
```


This makes debugging and working with the Python interpreter awkward and challenging. Thankfully there's a quick fix for this: the [functools.wraps decorator](#) included in Python's standard library.

You can use `functools.wraps` in your own decorators to copy over the lost metadata from the undecorated function to the decorator closure. Here's an example:

```
import functools

def uppercase(func):

    @functools.wraps(func)

    def wrapper():

        return func().upper()

    return wrapper
```

Applying `functools.wraps` to the wrapper closure returned by the decorator carries over the docstring and other metadata of the input function:

```
@uppercase

def greet():

    """Return a friendly greeting."""

    return 'Hello!'

>>> greet.__name__

'greet'

>>> greet.__doc__

'Return a friendly greeting.'
```

As a best practice I'd recommend that you use `functools.wraps` in all of the decorators you write yourself. It doesn't take much time and it will save you (and others) debugging headaches down the road.

Python Iterators

An iterator is an object that contains a countable number of values.

An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.

Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods `__iter__()` and `__next__()`.

Iterator vs Iterable

Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable *containers* which you can get an iterator from.

All these objects have a `iter()` method which is used to get an iterator:

Example

Return an iterator from a tuple, and print each value:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

Even strings are iterable objects, and can return an iterator:

Example

Strings are also iterable objects, containing a sequence of characters:

```
mystr = "banana"
myit = iter(mystr)
```

```
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

Looping Through an Iterator

We can also use a `for` loop to iterate through an iterable object:

Example

Iterate the values of a tuple:

```
mytuple = ("apple", "banana", "cherry")

for x in mytuple:
    print(x)
```

Example

Iterate the characters of a string:

```
mystr = "banana"

for x in mystr:
    print(x)
```

The `for` loop actually creates an iterator object and executes the `next()` method for each loop.

Create an Iterator

To create an object/class as an iterator you have to implement the methods `__iter__()` and `__next__()` to your object.

As we know that all classes have a function called `__init__()`, which allows you do some initializing when the object is being created.

The `__iter__()` method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.

The `__next__()` method also allows you to do operations, and must return the next item in the sequence.

Example

Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

```
class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self

    def __next__(self):
        x = self.a
        self.a += 1
        return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

StopIteration

The example above would continue forever if you had enough `next()` statements, or if it was used in a `for` loop.

To prevent the iteration to go on forever, we can use the `StopIteration` statement.

In the `__next__()` method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

Example

Stop after 20 iterations:

```
class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self

    def __next__(self):
        if self.a <= 20:
            x = self.a
            self.a += 1
            return x
        else:
            raise StopIteration
```

```
myclass = MyNumbers()
myiter = iter(myclass)
```

```
for x in myiter:
    print(x)
```

What are generators in Python?

There is a lot of overhead in building an iterator in Python; we have to implement a class with `__iter__()` and `__next__()` method, keep track of internal states, raise `StopIteration` when there was no values to be returned etc.

This is both lengthy and counter intuitive. Generator comes into rescue in such situations.

Python generators are a simple way of creating iterators. All the overhead we mentioned above are automatically handled by generators in Python.

Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

How to create a generator in Python?

It is fairly simple to create a generator in Python. It is as easy as defining a normal function with `yield` statement instead of a `return` statement.

If a function contains at least one `yield` statement (it may contain other `yield` or `return` statements), it becomes a generator function.

Both `yield` and `return` will return some value from a function.

The difference is that, while a `return` statement terminates a function entirely, `yield` statement pauses the function saving all its states and later continues from there on successive calls.

Differences between Generator function and a Normal function

Here is how a generator function differs from a normal function.

- Generator function contains one or more `yield` statement.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like `__iter__()` and `__next__()` are implemented automatically. So we can iterate through the items using `next()`.
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, `StopIteration` is raised automatically on further calls.

```
# A simple generator function
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
    yield n

a = my_gen()
next(a)
```

Filter(), Map(), Reduce()

Suppose we have a code like this:

```
def is_even(n):  
    return n%2 == 0  
  
nums = [2,3,4,5,6,7,8]  
evens = list(filter(is_even, nums))  
print(evens)
```

Here, we can see that the function `is_even` is only used once. So instead of making a whole function, we can use lambda function in this case.

```
nums = [2,3,4,5,6,7,8]  
evens = list(filter (lambda n: n%2==0, nums)) #filter(function,  
sequence)  
print(evens)
```

Now what if you want to double all the values in the list. We do –

```
nums = [2,3,4,5,6,7,8]

evens = list (filter (lambda n : n%2==0, nums))

doubles = list (map (lambda n : n *2, evens))

print(doubles)
```

Now what if you want to reduce all the values in the list to a single value

```
from functools import reduce

nums = [2,3,4,5,6,7,8]

evens = list (filter (lambda n : n%2==0, nums))

doubles = list (map (lambda n : n *2, evens))

add = reduce (lambda a,b : a+b, doubles)

print(add)
```

Multithreading in Python

1. **Multiprogramming** – A computer running more than one program at a time (like running Excel and Firefox simultaneously).
2. **Multiprocessing** – A computer using more than one CPU at a time.
3. **Multitasking** – Tasks sharing a common resource (like 1 CPU).
4. **Multithreading** is an extension of multitasking.

Just like multiprocessing, multithreading is a way of achieving multitasking. In multithreading, the concept of **threads** is used.

Thread

In computing, a **process** is an instance of a computer program that is being executed. Any process has 3 basic components:

- An executable program.
- The associated data needed by the program (variables, work space, buffers, etc.)
- The execution context of the program (State of process)

A **thread** is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS.

In simple words, a **thread** is a sequence of such instructions within a program that can be executed independently of other code. For simplicity, you can assume that a thread is simply a subset of a process.

A thread contains all this information in a **Thread Control Block (TCB)**:

- **Thread Identifier:** Unique id (TID) is assigned to every new thread
- **Stack pointer:** Points to thread's stack in the process. Stack contains the local variables under thread's scope.
- **Program counter:** a register which stores the address of the instruction currently being executed by thread.
- **Thread state:** can be running, ready, waiting, start or done.
- **Thread's register set:** registers assigned to thread for computations.
- **Parent process Pointer:** A pointer to the Process control block (PCB) of the process that the thread lives on.

Multithreading

Multiple threads can exist within one process where:

- Each thread contains its own **register set** and **local variables (stored in stack)**.
- All thread of a process shares **global variables (stored in heap)** and the **program code**.

In a simple, single-core CPU, it is achieved using frequent switching between threads. This is termed as **context switching**. In context switching, the state of a thread is saved and state of another thread is loaded whenever any interrupt (due to I/O or manually set) takes place. Context switching takes place so frequently that all the threads appear to be running parallelly (this is termed as **multitasking**).

Example code:

```
from time import sleep
from threading import *

class Hello(Thread):
    def run(self):
        for i in range(5):
            print('Hello')
            sleep(1)

class Hi(Thread):
    def run(self):
        for i in range(5):
            print('Hi')
            sleep(1)

t1 = Hello()
t2 = Hi()

t1.start()
sleep(0.2)
t2.start()

t1.join()
t2.join()

print('Bye')
```

What is a Module?

Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application.

Create a Module

To create a module just save the code you want in a file with the file extension `.py`:

Example

Save this code in a file named `mymodule.py`

```
def greeting(name):  
    print("Hello, " + name)
```

Use a Module

Now we can use the module we just created, by using the `import` statement:

Example

Import the module named `mymodule`, and call the `greeting` function:

```
import mymodule  
  
mymodule.greeting("Jonathan")
```

Note: When using a function from a module, use the syntax: `module_name.function_name`.

Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Example

Save this code in the file `mymodule.py`

```
person1 = {  
    "name": "John",  
    "age": 36,  
    "country": "Norway"  
}
```

Example

Import the module named mymodule, and access the person1 dictionary:

```
import mymodule  
  
a = mymodule.person1["age"]  
print(a)
```

Naming a Module

You can name the module file whatever you like, but it must have the file extension `.py`

Re-naming a Module

You can create an alias when you import a module, by using the `as` keyword:

Example

Create an alias for `mymodule` called `mx`:

```
import mymodule as mx  
  
a = mx.person1["age"]  
print(a)
```

Built-in Modules

There are several built-in modules in Python, which you can import whenever you like.

Example

Import and use the `platform` module:

```
import platform

x = platform.system()
print(x)
```

Using the dir() Function

There is a built-in function to list all the function names (or variable names) in a module. The `dir()` function:

Example

List all the defined names belonging to the `platform` module:

```
import platform

x = dir(platform)
print(x)
```

Note: The `dir()` function can be used on *all* modules, also the ones you create yourself.

Import From Module

You can choose to import only parts from a module, by using the `from` keyword.

Example

The module named `mymodule` has one function and one dictionary:

```
def greeting(name):
    print("Hello, " + name)
```

```
person1 = {  
    "name": "John",  
    "age": 36,  
    "country": "Norway"  
}
```

Example

Import only the person1 dictionary from the module:

```
from mymodule import person1  
  
print (person1["age"])
```

Note: When importing using the `from` keyword, do not use the module name when referring to elements in the module.

Example: `person1["age"]`, **not** `mymodule.person1["age"]`

Python Dates

A date in Python is not a data type of its own, but we can import a module named `datetime` to work with dates as date objects.

Example

Import the datetime module and display the current date:

```
import datetime  
  
x = datetime.datetime.now()  
print(x)
```

Date Output

When we execute the code from the example above the result will be:

`2019-06-10 00:31:46.866411`

The date contains year, month, day, hour, minute, second, and microsecond.

The `datetime` module has many methods to return information about the date object.

Here are a few examples, you will learn more about them later in this chapter:

Example

Return the year and name of weekday:

```
import datetime

x = datetime.datetime.now()

print(x.year)
print(x.strftime("%A"))
```

Creating Date Objects

To create a date, we can use the `datetime()` class (constructor) of the `datetime` module.

The `datetime()` class requires three parameters to create a date: year, month, day.

Example

Create a date object:

```
import datetime

x = datetime.datetime(2020, 5, 17)

print(x)
```

The `datetime()` class also takes parameters for time and timezone (hour, minute, second, microsecond, tzzone), but they are optional, and has a default value of `0`, (`None` for timezone).

The strftime() Method

The `datetime` object has a method for formatting date objects into readable strings.

The method is called `strftime()`, and takes one parameter, `format`, to specify the format of the returned string:

Example

Display the name of the month:

```
import datetime

x = datetime.datetime(2018, 6, 1)

print(x.strftime("%B"))
```

A reference of all the legal format codes:

Directive	Description	Example
%a	Weekday, short version	Wed
%A	Weekday, full version	Wednesday
%w	Weekday as a number 0-6, 0 is Sunday	3
%d	Day of month 01-31	31
%b	Month name, short version	Dec

%B	Month name, full version	December
%m	Month as a number 01-12	12
%y	Year, short version, without century	18
%Y	Year, full version	2018
%H	Hour 00-23	17
%I	Hour 00-12	05
%p	AM/PM	PM
%M	Minute 00-59	41
%S	Second 00-59	08
%f	Microsecond 000000-999999	548513
%z	UTC offset	+0100

%Z	Timezone	CST
%j	Day number of year 001-366	365
%U	Week number of year, Sunday as the first day of week, 00-53	52
%W	Week number of year, Monday as the first day of week, 00-53	52
%c	Local version of date and time	Mon Dec 31 17:41:00 2018
%x	Local version of date	12/31/18
%X	Local version of time	17:41:00
%%	A % character	%

Python JSON

JSON is a syntax for storing and exchanging data.

JSON is text, written with JavaScript object notation.

JSON in Python

Python has a built-in package called `json`, which can be used to work with JSON data.

Example

Import the json module:

```
import json
```

Parse JSON - Convert from JSON to Python

If you have a JSON string, you can parse it by using the `json.loads()` method.

The result will be a [Python dictionary](#).

Example

Convert from JSON to Python:

```
import json

# some JSON:
x = '{ "name":"John", "age":30, "city":"New York"}'

# parse x:
y = json.loads(x)

# the result is a Python dictionary:
print(y["age"])
```

Convert from Python to JSON

If you have a Python object, you can convert it into a JSON string by using the `json.dumps()` method.

Example

Convert from Python to JSON:

```
import json

# a Python object (dict):
x = {
    "name": "John",
    "age": 30,
    "city": "New York"
}

# convert into JSON:
y = json.dumps(x)

# the result is a JSON string:
print(y)
```

You can convert Python objects of the following types, into JSON strings:

- dict
- list
- tuple
- string
- int
- float
- True
- False
- None

Example

Convert Python objects into JSON strings, and print the values:

```
import json

print(json.dumps({"name": "John", "age": 30}))
print(json.dumps(["apple", "bananas"]))
print(json.dumps(("apple", "bananas")))
print(json.dumps("hello"))
```

```
print(json.dumps(42))
print(json.dumps(31.76))
print(json.dumps(True))
print(json.dumps(False))
print(json.dumps(None))
```

When you convert from Python to JSON, Python objects are converted into the JSON (JavaScript) equivalent:

Python	JSON
Dict	Object
List	Array
Tuple	Array
Str	String
Int	Number
Float	Number
True	true

False	false
None	null

Example

Convert a Python object containing all the legal data types:

```
import json

x = {
    "name": "John",
    "age": 30,
    "married": True,
    "divorced": False,
    "children": ("Ann", "Billy"),
    "pets": None,
    "cars": [
        {"model": "BMW 230", "mpg": 27.5},
        {"model": "Ford Edge", "mpg": 24.1}
    ]
}

print(json.dumps(x))
```

Format the Result

The example above prints a JSON string, but it is not very easy to read, with no indentations and line breaks.

The `json.dumps()` method has parameters to make it easier to read the result:

Example

Use the `indent` parameter to define the numbers of indents:

```
json.dumps(x, indent=4)
```

You can also define the separators, default value is (" ", ": "), which means using a comma and a space to separate each object, and a colon and a space to separate keys from values:

Example

Use the `separators` parameter to change the default separator:

```
json.dumps(x, indent=4, separators=(". ", " = "))
```

Order the Result

The `json.dumps()` method has parameters to order the keys in the result:

Example

Use the `sort_keys` parameter to specify if the result should be sorted or not:

```
json.dumps(x, indent=4, sort_keys=True)
```

Python RegEx

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.

RegEx can be used to check if a string contains the specified search pattern.

RegEx Module

Python has a built-in package called `re`, which can be used to work with Regular Expressions.

Import the `re` module:

```
import re
```

RegEx in Python

When you have imported the `re` module, you can start using regular expressions:

Example

Search the string to see if it starts with "The" and ends with "Spain":

```
import re
```

```
txt = "The rain in Spain"  
x = re.search("^The.*Spain$", txt)
```

RegEx Functions

The `re` module offers a set of functions that allows us to search a string for a match:

Function	Description
----------	-------------

findall	Returns a list containing all matches
-------------------------	---------------------------------------

[search](#) Returns a [Match object](#) if there is a match anywhere in the string

[split](#) Returns a list where the string has been split at each match

[sub](#) Replaces one or many matches with a string

Metacharacters

Metacharacters are characters with a special meaning:

Character	Description	Example
[]	A set of characters	"[a-m]"
\	Signals a special sequence (can also be used to escape special characters)	"\d"
.	Any character (except newline character)	"he..o"
^	Starts with	"^hello"

\$	Ends with	"world\$"
*	Zero or more occurrences	"aix*"
+	One or more occurrences	"aix+"
{}	Exactly the specified number of occurrences	"al{2}"
	Either or	"falls stays"
()	Capture and group	

Example of meta characters:

```
import re

str = "The rain in Spain"

#Find all lower case characters alphabetically between "a" and "m":

x = re.findall("[a-m]", str)

print(x)
```

Example 2:

```
import re

str = "hello world"

#Check if the string starts with 'hello':

x = re.findall("^hello", str)


if (x):

    print("Yes, the string starts with 'hello'")

else:

    print("No match")
```

Special Sequences

A special sequence is a  followed by one of the characters in the list below, and has a special meaning:

Character	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word	r"\bain" r"ain\b"

<code>\B</code>	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word	<code>r"\Bain"</code> <code>r"ain\B"</code>
<code>\d</code>	Returns a match where the string contains digits (numbers from 0-9)	<code>"\d"</code>
<code>\D</code>	Returns a match where the string DOES NOT contain digits	<code>"\D"</code>
<code>\s</code>	Returns a match where the string contains a white space character	<code>"\s"</code>
<code>\S</code>	Returns a match where the string DOES NOT contain a white space character	<code>"\S"</code>
<code>\w</code>	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore <code>_</code> character)	<code>"\w"</code>
<code>\W</code>	Returns a match where the string DOES NOT contain any word characters	<code>"\W"</code>
<code>\Z</code>	Returns a match if the specified characters are at the end of the string	<code>"Spain\Z"</code>

Example of special sequences:

```
import re

str = "The rain in Spain"

#Check if the string starts with "The":

x = re.findall("\AThe", str)

print(x)

if (x):

    print("Yes, there is a match!")

else:

    print("No match")
```

Example 2:

```
import re

str = "The rain in Spain"

#Return a match at every NON white-space character:

x = re.findall("\S", str)

print(x)

if (x):

    print("Yes, there is at least one match!")

else:

    print("No match")
```

Sets

A set is a set of characters inside a pair of square brackets `[]` with a special meaning:

Set	Description
<code>[arn]</code>	Returns a match where one of the specified characters (<code>a</code> , <code>r</code> , or <code>n</code>) are present
<code>[a-n]</code>	Returns a match for any lower case character, alphabetically between <code>a</code> and <code>n</code>
<code>[^arn]</code>	Returns a match for any character EXCEPT <code>a</code> , <code>r</code> , and <code>n</code>
<code>[0123]</code>	Returns a match where any of the specified digits (<code>0</code> , <code>1</code> , <code>2</code> , or <code>3</code>) are present
<code>[0-9]</code>	Returns a match for any digit between <code>0</code> and <code>9</code>
<code>[0-5][0-9]</code>	Returns a match for any two-digit numbers from <code>00</code> and <code>59</code>
<code>[a-zA-Z]</code>	Returns a match for any character alphabetically between <code>a</code> and <code>z</code> , lower case OR upper case

[+]

In sets, `+`, `*`, `.`, `|`, `()`, `$`, `{}` has no special meaning,
so `[+]` means: return a match for any `+` character in the string

Sets example:

```
import re

str = "The rain in Spain"

#Check if the string has any a, r, or n characters:

x = re.findall("[arn]", str)

print(x)

if (x):

    print("Yes, there is at least one match!")

else:

    print("No match")
```

Example 2:

```
import re

str = "8 times before 11:45 AM"

#Check if the string has any two-digit numbers, from 00 to 59:

x = re.findall("[0-5][0-9]", str)

print(x)

if (x):

    print("Yes, there is at least one match!")
```

else:

```
print("No match")
```

The findall() Function

The `findall()` function returns a list containing all matches.

Example

Print a list of all matches:

```
import re

str = "The rain in Spain"
x = re.findall("ai", str)
print(x)
```

The list contains the matches in the order they are found.

If no matches are found, an empty list is returned:

Example

Return an empty list if no match was found:

```
import re

str = "The rain in Spain"
x = re.findall("Portugal", str)
print(x)
```

The search() Function

The `search()` function searches the string for a match, and returns a [Match object](#) if there is a match.

If there is more than one match, only the first occurrence of the match will be returned:

Example

Search for the first white-space character in the string:

```
import re

str = "The rain in Spain"
x = re.search("\s", str)

print("The first white-space character is located in position:", x.start())
```

If no matches are found, the value `None` is returned:

Example

Make a search that returns no match:

```
import re

str = "The rain in Spain"
x = re.search("Portugal", str)
print(x)
```

The split() Function

The `split()` function returns a list where the string has been split at each match:

Example

Split at each white-space character:

```
import re

str = "The rain in Spain"
x = re.split("\s", str)
print(x)
```

You can control the number of occurrences by specifying the `maxsplit` parameter:

Example

Split the string only at the first occurrence:

```
import re

str = "The rain in Spain"
x = re.split("\s", str, 1)
print(x)
```

The sub() Function

The `sub()` function replaces the matches with the text of your choice:

Example

Replace every white-space character with the number 9:

```
import re

str = "The rain in Spain"
x = re.sub("\s", "9", str)
print(x)
```

You can control the number of replacements by specifying the `count` parameter:

Example

Replace the first 2 occurrences:

```
import re

str = "The rain in Spain"
x = re.sub("\s", "9", str, 2)
print(x)
```

Match Object

A Match Object is an object containing information about the search and the result.

Note: If there is no match, the value `None` will be returned, instead of the Match Object.

Example

Do a search that will return a Match Object:

```
import re

str = "The rain in Spain"
x = re.search("ai", str)
print(x) #this will print an object
```

The Match object has properties and methods used to retrieve information about the search, and the result:

- `.span()` returns a tuple containing the start-, and end positions of the match.
- `.string` returns the string passed into the function
- `.group()` returns the part of the string where there was a match

Example

Print the position (start- and end-position) of the first match occurrence.

The regular expression looks for any words that starts with an upper case "S":

```
import re

str = "The rain in Spain"
x = re.search(r"\bS\w+", str)
print(x.span())
```

Example

Print the string passed into the function:

```
import re

str = "The rain in Spain"
x = re.search(r"\bS\w+", str)
print(x.string)
```

Example

Print the part of the string where there was a match.

The regular expression looks for any words that starts with an upper case "S":

```
import re

str = "The rain in Spain"
x = re.search(r"\bS\w+", str)
print(x.group())
```

Note: If there is no match, the value `None` will be returned, instead of the Match Object.

Python PIP

What is PIP?

PIP is a package manager for Python packages, or modules if you like.

Note: If you have Python version 3.4 or later, PIP is included by default.

What is a Package?

A package contains all the files you need for a module.

Modules are Python code libraries you can include in your project.

Check if PIP is Installed

Navigate your command line to the location of Python's script directory, and type the following:

Example

Check PIP version:

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip --version
```

Install PIP

If you do not have PIP installed, you can download and install it from this page: <https://pypi.org/project/pip/>

Download a Package

Downloading a package is very easy.

Open the command line interface and tell PIP to download the package you want.

Navigate your command line to the location of Python's script directory, and type the following:

Example

Download a package named "camelcase":

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip install camelcase
```

Now you have downloaded and installed your first package!

Using a Package

Once the package is installed, it is ready to use.

Import the "camelcase" package into your project.

Example

Import and use "camelcase":

```
import camelcase
```

```
c = camelcase.CamelCase()

txt = "hello world"

print(c.hump(txt))
```

Find Packages

Find more packages at <https://pypi.org/>.

Remove a Package

Use the `uninstall` command to remove a package:

Example

Uninstall the package named "camelcase":

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip
uninstall camelcase
```

The PIP Package Manager will ask you to confirm that you want to remove the camelcase package:

```
Uninstalling camelcase-02.1:
  Would remove:
    c:\users\Your Name\appdata\local\programs\python\python36-32\lib\site-
packages\camecase-0.2-py3.6.egg-info
    c:\users\Your Name\appdata\local\programs\python\python36-32\lib\site-
packages\camecase\*
Proceed (y/n)?
```

Press **y** and the package will be removed.

List Packages

Use the `list` command to list all the packages installed on your system:

Example

List installed packages:

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip list
```

Result:

Package	Version

camelcase	0.2
mysql-connector	2.1.6
pip	18.1
pymongo	3.6.1
setuptools	39.0.1

Python Try Except

The `try` block lets you test a block of code for errors.

The `except` block lets you handle the error.

The `finally` block lets you execute code, regardless of the result of the try- and except blocks.

Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the `try` statement:

Example

The `try` block will generate an exception, because `x` is not defined:

```
try:  
    print(x)  
except:  
    print("An exception occurred")
```

Since the try block raises an error, the except block will be executed.

Without the try block, the program will crash and raise an error:

Example

This statement will raise an error, because `x` is not defined:

```
print(x)
```

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example

Print one message if the try block raises a `NameError` and another for other errors:

```
try:
    print(x)
except NameError:
    print("Variable x is not defined")
except:
    print("Something else went wrong")
```

If we execute the following code,

```
age = int( input ('Age : ' ) )
print(age)
```

It prints exit code 0 when input is numerical value. Exit code 0 means the program executed successfully. But if we put anything other than numbers as input it will give us an exit code of 1 and will show `ValueError`.

To execute the program successfully, we have try and except statements:


```
try:
    age = int( input ('Age : ' ) )
    print(age)
except ValueError:
    print('Invalid value')
```

If we put anything other than numbers in the following code, the program will execute successfully with exit code 0 and will print invalid value statement. We do this because we don't want our entire program to crash just because user entered something other than numbers.

Multiple except constructs:

```
try:
    age = int( input ('Age : ' ) )
    income = 40000
    risk = income / age
    print(age)
except ZeroDivisionError:
    print("Age cannot be zero")
except ValueError:
    print('Invalid value')
```

Else

You can use the **else** keyword to define a block of code to be executed if no errors were raised:

Example

In this example, the `try` block does not generate any error:

```
try:
    print("Hello")
except:
    print("Something went wrong")
else:
    print("Nothing went wrong")
```

Finally

The `finally` block, if specified, will be executed regardless if the `try` block raises an error or not.

Example

```
try:
    print(x)
except:
    print("Something went wrong")
finally:
    print("The 'try except' is finished")
```

This can be useful to close objects and clean up resources:

Example

Try to open and write to a file that is not writable:

```
try:
    f = open("demofile.txt")
    f.write("Lorum Ipsum")
except:
    print("Something went wrong when writing to the file")
finally:
    f.close()
```

The program can continue, without leaving the file object open.