Link to download anachonda for python:

https://repo.anaconda.com/archive/Anaconda3-2023.07-2-Windows-x86_64.exe

Python

Python is a powerful general-purpose programming language. It is used in web development, data science, creating software, and so on. Fortunately for beginners, Python has simple easy-to-use syntax. This makes Python an excellent language to learn to program for beginners.

Why Learn Python?

- Python is easy to learn. Its syntax is easy and code is very readable.
- Python has a lot of applications. It's used for developing web applications, data science, rapid application development, and so on.
- Python allows you to write programs in fewer lines of code than most of the programming languages.
- The popularity of Python is growing rapidly. Now it's one of the most popular programming languages.

Python is a cross-platform programming language, which means that it can run on multiple platforms like Windows, macOS, Linux. It is free and open-source. Even though most of today's Linux and Mac have Python pre-installed in it, the version might be out-of-date. So, it is always a good idea to install the most current version.

```
////////////////////////////////////////////////////////////////
print("Hello, World!")
/////////////////////////////////////////
Here check : in if statement end;;;;;
if 5 > 2:
  print("Five is greater than two!")
/////////////////////////////////////////////////////
```

You have to use the same number of spaces in the same block of code, otherwise Python will give you an error:

```
if 5 > 2:
 print("Five is greater than two!")
        print("Five is greater than two!")#this line will show error
```

but this will not;;;;;;;;;;;;

```
if 5 > 2:
 print("Five is greater than two!")
if 5 > 2:
        print("Five is greater than two!")
///////////////////////////////////////////////////////////////////
```

Python Variables

```
x = 5
y = "Hello, World!"
```

```
/////////////////////////////////////////////////////////////////////
```

Comments

Python has commenting capability for the purpose of in-code documentation.

Comments start with a #, and Python will render the rest of the line as a comment:

Example

Comments in Python:

```
#This is a comment.
print("Hello, World!")
```

```
print("Hello, World!") #This is a comment
```

Comments can be used to explain Python code.

Comments can be used to make the code more readable.

Comments can be used to prevent execution when testing code.

Multi Line Comments

Python **does not** really have a syntax for multi line comments.

To add a multiline comment you could insert a `#` for each line:

Example

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

(triple quotes) in your code, and place your comment inside it:

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

//

Creating Variables

Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

Example

```
x = 5
y = "John"
print(x)
print(y)
```

Variables do not need to be declared with any particular *type*, and can even change type after they have been set.

Example

```
x = 4       # x is of type int
x = "Sally" # x is now of type str
print(x)
```

//

Casting

If you want to specify the data type of a variable, this can be done with casting.

Example

```
x = str(3)    # x will be '3'
y = int(3)    # y will be 3
z = float(3)  # z will be 3.0
```

///

Get the Type

You can get the data type of a variable with the `type()` function.

Example

```
x = 5
y = "John"
print(type(x))
print(type(y))
```

//

Single or Double Quotes?

String variables can be declared either by using single or double quotes:

Example

```
x = "John"
# is the same as
x = 'John'
```

//

Case-Sensitive

Variable names are case-sensitive.

Example

This will create two variables:

```
a = 4
A = "Sally"
#A will not overwrite a
```

//

Python - Variable Names

Variable 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)

Example

Legal variable names:

```
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"
```

Try it Yourself »

Example

Illegal variable names:

```
2myvar = "John"
my-var = "John"
my var = "John"
```

Try it Yourself »

Remember that variable names are case-sensitive

Multi Words Variable Names

Variable names with more than one word can be difficult to read.

There are several techniques you can use to make them more readable:

Camel Case

Each word, except the first, starts with a capital letter:

```
myVariableName = "John"
```

Pascal Case

Each word starts with a capital letter:

```
MyVariableName = "John"
```

Snake Case

Each word is separated by an underscore character:

```
my_variable_name = "John"
```

//

Python Variables - Assign Multiple Values

Many Values to Multiple Variables

Python allows you to assign values to multiple variables in one line:

Example

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

Try it Yourself »

Note: Make sure the number of variables matches the number of values, or else you will get an error.

One Value to Multiple Variables

And you can assign the *same* value to multiple variables in one line:

Example

```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

Try it Yourself »

Unpack a Collection

If you have a collection of values in a list, tuple etc. Python allows you to extract the values into variables. This is called *unpacking*.

Example

Unpack a list:

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

//

Python - Output Variables

Output Variables

The Python `print` statement is often used to output variables.

To combine both text and a variable, Python uses the `+` character:

Example

```
x = "awesome"
print("Python is " + x)
```

Try it Yourself »

You can also use the `+` character to add a variable to another variable:

Example

```
x = "Python is "
y = "awesome"
z =  x + y
print(z)
```

Try it Yourself »

For numbers, the `+` character works as a mathematical operator:

Example

```
x = 5
y = 10
print(x + y)
```

Try it Yourself »

If you try to combine a string and a number, Python will give you an error:

Example

```
x = 5
y = "John"
print(x + y)
```

//

Python - Global Variables

Global Variables

Variables that are created outside of a function (as in all of the examples above) are known as global variables.

Global variables can be used by everyone, both inside of functions and outside.

Example

Create a variable outside of a function, and use it inside the function

```
x = "awesome"

def myfunc():
  print("Python is " + x)

myfunc()
```

Try it Yourself »

If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value.

Example

Create a variable inside a function, with the same name as the global variable

```
x = "awesome"
```

```
def myfunc():
  x = "fantastic"
  print("Python is " + x)

myfunc()

print("Python is " + x)
```

The **global** Keyword

Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.

To create a global variable inside a function, you can use the `global` keyword.

Example

If you use the `global` keyword, the variable belongs to the global scope:

```
def myfunc():
  global x
  x = "fantastic"

myfunc()

print("Python is " + x)
```

Also, use the `global` keyword if you want to change a global variable inside a function.

Example

To change the value of a global variable inside a function, refer to the variable by using the `global` keyword:

```
x = "awesome"
```

```
def myfunc():
  global x
  x = "fantastic"

myfunc()

print("Python is " + x)
```

//

Python Data Types

Built-in Data Types

In programming, data type is an important concept.

Variables can store data of different types, and different types can do different things.

Python has the following data types built-in by default, in these categories:

Text Type:	`str`
Numeric Types:	`int`, `float`, `complex`
Sequence Types:	`list`, `tuple`, `range`
Mapping Type:	`dict`
Set Types:	`set`, `frozenset`
Boolean Type:	`bool`
Binary Types:	`bytes`, `bytearray`, `memoryview`

Getting the Data Type

You can get the data type of any object by using the `type()` function:

Example

Print the data type of the variable x:

```
x = 5
print(type(x))
```

Try it Yourself »

Setting the Data Type

In Python, the data type is set when you assign a value to a variable:

Example	Data Type
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple

`x = range(6)`	range
`x = {"name" : "John", "age" : 36}`	dict
`x = {"apple", "banana", "cherry"}`	set
`x = True`	bool

The list is dynamic, whereas the tuple has static characteristics. This means that lists can be modified whereas tuples cannot be modified, the tuple is faster than the list because of static in nature. Lists are denoted by the square brackets but tuples are denoted as parenthesis.

```
my_list = [1, 2, 3]
my_list.append(14)
my_list.remove(2)
print(my_list)  # Output: [1, 2, 3, 14]

my_list = [1, 2, 3]
my_list.insert(1, 4)  # Insert 4 at index 1
print(my_list)  # Output: [1, 4, 2, 3]

my_list = [1, 2, 3]
my_list.extend([14, 15])
print(my_list)  # Output: [1, 2, 3, 14, 15]

my_list = [1, 2, 3]
my_list += [14, 15]
print(my_list)  # Output: [1, 2, 3, 14, 15]
```

Dictionary

```
my_dict = {'name': 'Alice', 'age': 30}

my_dict['city'] = 'New York'  # Adding a new key-value pair

print(my_dict)

# Output: {'name': 'Alice', 'age': 30, 'city': 'New York'}
```

```
my_dict = {'name': 'Alice', 'age': 30}

my_dict.update({'city': 'New York', 'country': 'USA'})  #
Adding multiple key-value pairs

print(my_dict)

# Output: {'name': 'Alice', 'age': 30, 'city': 'New York',
'country': 'USA'}
```

```
my_dict = {'name': 'Alice', 'age': 30}
my_dict.update({'age': 31})  # Updating the 'age' key
my_dict.__delitem__("name")
print(my_dict)
# Output: {'age': 31}
```

Set

```
my_set = {1, 2, 3}
my_set.add(14)  # Adding a single value
print(my_set)  # Output: {1, 2, 3, 14}

my_set = {1, 2, 3}
my_set.update([13,14, 15])  # Adding multiple values
my_set.discard(3)  # delete a number
print(my_set)  # Output: {1, 2, 13, 14, 15}
```

Lists	Tuples	Sets	Dictionaries
A list is a collection of *ordered* data.	A tuple is an *ordered* collection of data.	A set is an *unordered* collection.	A dictionary is an *unordered* collection of data that stores data in key-value pairs.
Lists are *mutable*.	Tuples are *immutable*.	Sets are *mutable* and have *no duplicate elements*.	Dictionaries are mutable and keys do not allow duplicates.
Lists are declared with square braces.	Tuples are enclosed within parenthesis.	Sets are represented in curly brackets.	Dictionaries are enclosed in curly brackets in the form of key-value pairs.

Python User Input

User Input

Python allows for user input.

That means we are able to ask the user for input.

The method is a bit different in Python 3.6 than Python 2.7.

Python 3.6 uses the `input()` method.

Python 2.7 uses the `raw_input()` method.

The following example asks for the username, and when you entered the username, it gets printed on the screen:

Python 3.6

```
username = input("Enter username:")
print("Username is: " + username)
```

Run Example »

Python 2.7

```
username = raw_input("Enter username:")
print("Username is: " + username)
```

Run Example »

Python stops executing when it comes to the `input()` function, and continues when the user has given some input.

Setting the Specific Data Type

If you want to specify the data type, you can use the following constructor functions:

Example	Data Type
`x = str("Hello World")`	str
`x = int(20)`	int
`x = float(20.5)`	float
`x = complex(1j)`	complex
`x = list(("apple", "banana", "cherry"))`	list
`x = tuple(("apple", "banana", "cherry"))`	tuple
`x = range(6)`	range
`x = dict(name="John", age=36)`	dict
`x = set(("apple", "banana", "cherry"))`	set

x = bool(5)	bool

Test Yourself With Exercises

Exercise:

The following code example would print the data type of x, what data type would that be?

```
x = 5
print(type(x))
```

//

Python Numbers

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
```

To verify the type of any object in Python, use the `type()` function:

Example

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

Try it Yourself »

Int

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

Example

Integers:

```
x = 1
y = 35656222554887711
z = -3255522

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

Try it Yourself »

Float

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

Example

Floats:

```
x = 1.10
y = 1.0
z = -35.59

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

Try it Yourself »

Complex

Complex numbers are written with a "j" as the imaginary part:

Example

Complex:

```
x = 3+5j
y = 5j
z = -5j

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

Try it Yourself »

Type Conversion

You can convert from one type to another with the `int()`, `float()`, and `complex()` methods:

Example

Convert from one type to another:

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

#convert from int to float:
a = float(x)

#convert from float to int:
b = int(y)

#convert from int to complex:
c = complex(x)

print(a)
print(b)
print(c)

print(type(a))
print(type(b))
print(type(c))
```

Try it Yourself »

Note: You cannot convert complex numbers into another number type.

Random Number

Python does not have a `random()` function to make a random number, but Python has a built-in module called `random` that can be used to make random numbers:

Example

Import the random module, and display a random number between 1 and 9:

```
import random
print(random.randrange(1,10))
```

Exercise:

Insert the correct syntax to convert x into a floating point number.

```
x = 5
x = [      ](x)
```

///

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 removing all decimals), 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
```

Try it Yourself »

Example

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
```

Try it Yourself »

Example

Strings:

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

Try it Yourself »

//

Python Strings are not the part of course work

//

Python Iterative Statements

Iteration statements or loop statements allow us to execute a block of statements as long as the condition is true.

Loops statements are used when we need to run same code again and again, each time with a different value.

Type of Iteration Statements In Python 3
In Python Iteration (Loops) statements are of three types :-

1. While Loop

2. For Loop

3. Nested For Loops

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)
```

Try it Yourself »

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)
```

Try it Yourself »

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
```

Try it Yourself »

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)
```

Try it Yourself »

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)
```

Try it Yourself »

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)
```

Try it Yourself »

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)
```

Try it Yourself »

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!")
```

Try it Yourself »

Note: The else block will NOT be executed if the loop is stopped by a break statement.

Example

Break the loop when x is 3, and see what happens with the else block:

```
for x in range(6):
  if x == 3: break
  print(x)
else:
  print("Finally finished!")
```

Try it Yourself »

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)
```

Try it Yourself »

The pass Statement

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

Example

```
for x in [0, 1, 2]:
  pass
```

Try it Yourself »

Python While 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
```

Try it Yourself »

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
```

Try it Yourself »

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)
```

Try it Yourself »

The else Statement

With the `else` statement we can run a block of code once when the condition no longer is true:

Example

Print a message once the condition is false:

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

Try it Yourself »

Exercise:

Print `i` as long as `i` is less than 6.

```
i = 1
[      ] i < 6[      ]
  print(i)
  i += 1
```

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()
```

Try it Yourself »

Arguments

Information can be passed into functions as arguments.

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

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

Example

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

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

Try it Yourself »

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

Parameters or Arguments?

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

From a function's perspective:

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

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

Number of Arguments

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

Example

This function expects 2 arguments, and gets 2 arguments:

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

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

Try it Yourself »

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

Example

This function expects 2 arguments, but gets only 1:

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

my_function("Emil")
```

Try it Yourself »

Arbitrary Arguments, *args

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

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

Example

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

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

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

Try it Yourself »

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

Keyword Arguments

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

This way the order of the arguments does not matter.

Example

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

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

Try it Yourself »

The phrase *Keyword Arguments* are often shortened to *kwargs* in Python documentations.

Arbitrary Keyword Arguments, **kwargs

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

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

Example

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

```
def my_function(**kid):
  print("His last name is " + kid["lname"])

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

Try it Yourself »

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

Default Parameter Value

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

If we call the function without argument, 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")
```

Try it Yourself »

Passing a List as an Argument

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

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

Example

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

Try it Yourself »

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))
```

Try it Yourself »

The pass Statement

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

Example

```
def myfunction():
  pass
```

Try it Yourself »

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
    print("my result at this point is "+str(result))
    return result


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

Try it Yourself »

Exercise:

Create a function named `my_function`.

```
[    ]:
  print("Hello from a function")
```

Python Operators

Python Operators

Operators are used to perform operations on variables and values.

In the example below, we use the `+` operator to add together two values:

Example

```
print(10 + 5)
```

Run example »

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y

Python Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As

=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3

ADVERTISEMENT

Python Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y

<=	Less than or equal to	x <= y

Python Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x <

Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y

---

Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description	Example
&	AND	Sets each bit to 1 if both bits are 1	x & y
|	OR	Sets each bit to 1 if one of two bits is 1	x | y

Operator Precedence

Operator precedence describes the order in which operations are performed.

Example

Parentheses has the highest precedence, meaning that expressions inside parentheses must be evaluated first:

```
print((6 + 3) - (6 + 3))
```

Run example »

Example

Multiplication * has higher precedence than addition +, and therefor multiplications are evaluated before additions:

```
print(100 + 5 * 3)
```

Run example »

Example

Addition + and subtraction - has the same precedence, and therefor we evaluate the expression from left to right:

```
print(5 + 4 - 7 + 3)
```

Run example »

Exercise:

Multiply `10` with `5`, and print the result.

```
print(10        5)
```

Python Math

Python has a set of built-in math functions, including an extensive math module, that allows you to perform mathematical tasks on numbers.

Built-in Math Functions

The `min()` and `max()` functions can be used to find the lowest or highest value in an iterable:

Example

```
x = min(5, 10, 25)
y = max(5, 10, 25)

print(x)
print(y)
```

Try it Yourself »

The `abs()` function returns the absolute (positive) value of the specified number:

Example

```
x = abs(-7.25)

print(x)
```

Try it Yourself »

The `pow(x, y)` function returns the value of x to the power of y (x^y).

Example

Return the value of 4 to the power of 3 (same as 4 * 4 * 4):

```
x = pow(4, 3)

print(x)
```

Try it Yourself »

ADVERTISEMENT

The Math Module

Python has also a built-in module called `math`, which extends the list of mathematical functions.

To use it, you must import the `math` module:

```
import math
```

When you have imported the `math` module, you can start using methods and constants of the module.

The `math.sqrt()` method for example, returns the square root of a number:

Example

```
import math

x = math.sqrt(64)

print(x)
```

Try it Yourself »

The `math.ceil()` method rounds a number upwards to its nearest integer, and the `math.floor()` method rounds a number downwards to its nearest integer, and returns the result:

Example

```
import math

x = math.ceil(1.4)
y = math.floor(1.4)

print(x) # returns 2
print(y) # returns 1
```

Try it Yourself »

The `math.pi` constant, returns the value of PI (3.14...):

Example

```
import math

x = math.pi

print(x)
```

Try it Yourself »

Python Random Module

Python has a built-in module that you can use to make random numbers.

The `random` module has a set of methods:

Method	Description
randrange()	Returns a random number between the given range
randint()	Returns a random number between the given range
choice()	Returns a random element from the given sequence
random()	Returns a random float number between 0 and 1

```
import random

# Generate a random integer between 1 and 10 (inclusive)
random_number = random.randint(1, 10)
print("Random Number:", random_number)

# Create a list of items
my_list = ["apple", "banana", "cherry", "date", "elderberry"]

# Select a random item from the list
random_item = random.choice(my_list)
print("Random Item:", random_item)
```

Python Classes and Objects

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
```

Try it Yourself »

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)
```

Try it Yourself »

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)
```

Try it Yourself »

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

The __str__() Function

The __str__() function controls what should be returned when the class object is represented as a string.

If the __str__() function is not set, the string representation of the object is returned:

Example

The string representation of an object WITHOUT the __str__() function:

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

p1 = Person("John", 36)

print(p1)
```

Try it Yourself »

Example

The string representation of an object WITH the __str__() function:

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

  def __str__(self):
    return f"{self.name}({self.age})"

p1 = Person("John", 36)

print(p1)
```

Try it Yourself »

Object Methods

Objects can also contain methods. Methods in objects are functions that belong 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()
```

Try it Yourself »

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

Try it Yourself »

Modify Object Properties

You can modify properties on objects like this:

Example

Set the age of p1 to 40:

```
p1.age = 40
```

Try it Yourself »

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
```

Try it Yourself »

Delete Objects

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

Example

Delete the p1 object:

```
del p1
```

Try it Yourself »

The pass Statement

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

Example

```
class Person:
  pass
```

Try it Yourself »

Exercise:

Create a class named MyClass:

```
[ ] MyClass:
  x = 5
```

Python Inheritance

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

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()
```

Try it Yourself »

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()
```

Try it Yourself »

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)
```

Try it Yourself »

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.

Use the super() Function

Python also has a `super()` function that will make the child class inherit all the methods and properties from its parent:

Example

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

Try it Yourself »

By using the `super()` function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

Add Properties

Example

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

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019
```

Try it Yourself »

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):
    super().__init__(fname, lname)
    self.graduationyear = year

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

Try it Yourself »

Add Methods

Example

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

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

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

Try it Yourself »

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.

Exercise:

What is the correct syntax to create a class named Student that will inherit properties and methods from a class named Person?

```
class [      ]:
```

Python Polymorphism

The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

Function Polymorphism

An example of a Python function that can be used on different objects is the `len()` function.

String

For strings `len()` returns the number of characters:

Example

```
x = "Hello World!"

print(len(x))
```

Try it Yourself »

Tuple

For tuples `len()` returns the number of items in the tuple:

Example

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

print(len(mytuple))
```

Try it Yourself »

Dictionary

For dictionaries `len()` returns the number of key/value pairs in the dictionary:

Example

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

print(len(thisdict))
```

Try it Yourself »

Class Polymorphism

Polymorphism is often used in Class methods, where we can have multiple classes with the same method name.

For example, say we have three classes: `Car`, `Boat`, and `Plane`, and they all have a method called `move()`:

Example

Different classes with the same method:

```
class Car:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Drive!")

class Boat:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Sail!")
```

```
class Plane:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang")       #Create a Car class
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat class
plane1 = Plane("Boeing", "747")     #Create a Plane class

for x in (car1, boat1, plane1):
  x.move()
```

Try it Yourself »

Look at the for loop at the end. Because of polymorphism we can execute the same method for all three classes.

Inheritance Class Polymorphism

What about classes with child classes with the same name? Can we use polymorphism there?

Yes. If we use the example above and make a parent class called `Vehicle`, and make `Car`, `Boat`, `Plane` child classes of `Vehicle`, the child classes inherits the `Vehicle` methods, but can override them:

Example

Create a class called `Vehicle` and make `Car`, `Boat`, `Plane` child classes of `Vehicle`:

```
class Vehicle:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
```

```
    print("Move!")

class Car(Vehicle):
  pass

class Boat(Vehicle):
  def move(self):
    print("Sail!")

class Plane(Vehicle):
  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang") #Create a Car object
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object
plane1 = Plane("Boeing", "747") #Create a Plane object

for x in (car1, boat1, plane1):
  print(x.brand)
  print(x.model)
  x.move()
```

Try it Yourself »

Child classes inherits the properties and methods from the parent class.

In the example above you can see that the `Car` class is empty, but it inherits `brand`, `model`, and `move()` from `Vehicle`.

The `Boat` and `Plane` classes also inherit `brand`, `model`, and `move()` from `Vehicle`, but they both override the `move()` method.

Because of polymorphism we can execute the same method for all classes.

Definition of Abstraction

Abstraction is an OOP concept that focuses only on relevant data of an object. It hides the background details and emphasizes the essential data points for reducing the complexity and increase efficiency. It generally retains only information which is most relevant for that specific process. Abstraction method mainly focusses on the idea instead of actual functioning.

Definition of Encapsulation

Encapsulation is a method of making a complex system easier to handle for end users. The user need not worry about internal details and complexities of the system. Encapsulation is a process of wrapping the data and the code, that operate on the data into a single entity. You can assume it as a protective wrapper that stops random access of code defined outside that wrapper.

S.No	Abstraction	Encapsulation
1.	It is the process of gaining information.	It is a method that helps wrap up data into a single module.
2.	The problems in this technique are solved at the interface level.	Problems in encapsulation are solved at the implementation level.
3.	It helps hide the unwanted details/information.	It helps hide data using a single entity, or using a unit with the help of method that helps protect the information.
4.	It can be implemented using abstract classes and interfaces.	It can be implemented using access modifiers like public, private and protected.
5.	The complexities of the implementation are hidden using interface and abstract class.	The data is hidden using methods such as getters and setters.
6.	Abstraction can be performed using objects that are encapsulated within a single module.	Objects in encapsulation don't need to be in abstraction.

Conclusion

The most significant difference between the two is that data abstraction is a method which helps to hide the unwanted data from the user, while data encapsulation is a method which helps to hide data using a single entity.

NumPy Tutorial

NumPy is a Python library.

NumPy is used for working with arrays.

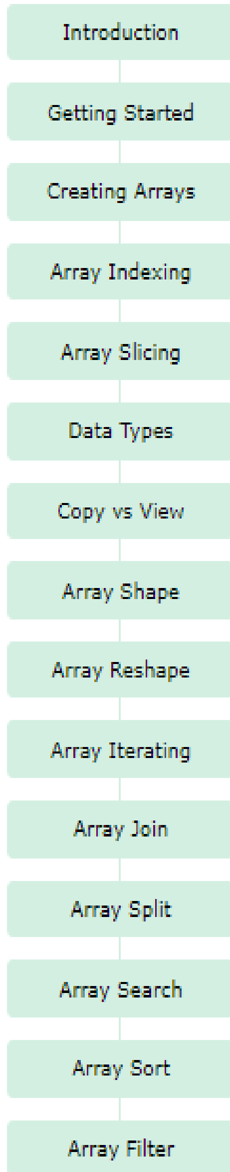
NumPy is short for "Numerical Python".

Learning by Reading

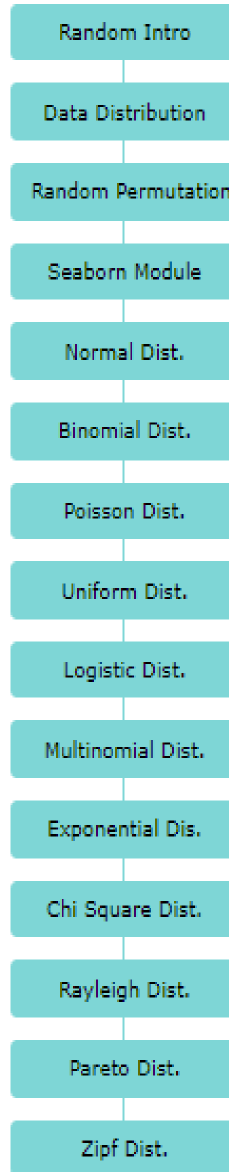
We have created 43 tutorial pages for you to learn more about NumPy.

Starting with a basic introduction and ends up with creating and plotting random data sets, and working with NumPy functions:

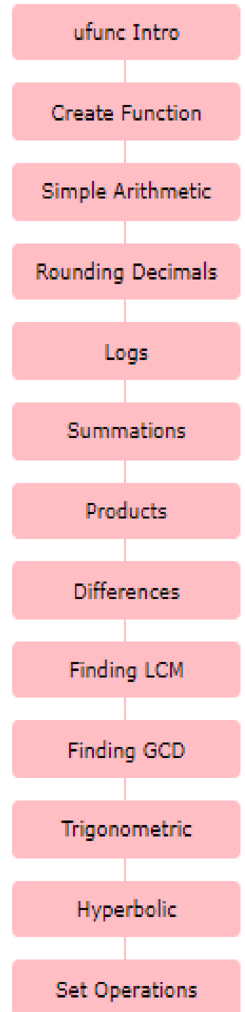
Basic



Random



ufunc



NumPy Introduction

What is NumPy?

NumPy is a Python library used for working with arrays.

It also has functions for working in domain of linear algebra, fourier transform, and matrices.

NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.

NumPy stands for Numerical Python.

Why Use NumPy?

In Python we have lists that serve the purpose of arrays, but they are slow to process.

NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.

The array object in NumPy is called `ndarray`, it provides a lot of supporting functions that make working with `ndarray` very easy.

Arrays are very frequently used in data science, where speed and resources are very important.

Data Science: is a branch of computer science where we study how to store, use and analyze data for deriving information from it.

Why is NumPy Faster Than Lists?

NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.

This behavior is called locality of reference in computer science.

This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

Which Language is NumPy written in?

NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.

Where is the NumPy Codebase?

The source code for NumPy is located at this github repository https://github.com/numpy/numpy

github: enables many people to work on the same codebase.

NumPy Getting Started

Installation of NumPy

If you have Python and PIP already installed on a system, then installation of NumPy is very easy.

Install it using this command:

```
C:\Users\Your Name>pip install numpy
```

If this command fails, then use a python distribution that already has NumPy installed like, Anaconda, Spyder etc.

Import NumPy

Once NumPy is installed, import it in your applications by adding the `import` keyword:

```
import numpy
```

Now NumPy is imported and ready to use.

Example

```
import numpy

arr = numpy.array([1, 2, 3, 4, 5])

print(arr)
```

Try it Yourself »

NumPy as np

NumPy is usually imported under the `np` alias.

alias: In Python alias are an alternate name for referring to the same thing.

Create an alias with the `as` keyword while importing:

```
import numpy as np
```

Now the NumPy package can be referred to as `np` instead of `numpy`.

Example

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)
```

Try it Yourself »

Checking NumPy Version

The version string is stored under `__version__` attribute.

Example

```
import numpy as np

print(np.__version__)
```

NumPy Creating Arrays

Create a NumPy ndarray Object

NumPy is used to work with arrays. The array object in NumPy is called `ndarray`.

We can create a NumPy `ndarray` object by using the `array()` function.

Example

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)

print(type(arr))
```

Try it Yourself »

type(): This built-in Python function tells us the type of the object passed to it. Like in above code it shows that `arr` is `numpy.ndarray` type.

To create an `ndarray`, we can pass a list, tuple or any array-like object into the `array()` method, and it will be converted into an `ndarray`:

Example

Use a tuple to create a NumPy array:

```
import numpy as np

arr = np.array((1, 2, 3, 4, 5))

print(arr)
```

Try it Yourself »

Dimensions in Arrays

A dimension in arrays is one level of array depth (nested arrays).

nested array: are arrays that have arrays as their elements.

ADVERTISEMENT

0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

Example

Create a 0-D array with value 42

```
import numpy as np

arr = np.array(42)
```

```
print(arr)
```

Try it Yourself »

1-D Arrays

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.

These are the most common and basic arrays.

Example

Create a 1-D array containing the values 1,2,3,4,5:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)
```

Try it Yourself »

2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array.

These are often used to represent matrix or 2nd order tensors.

NumPy has a whole sub module dedicated towards matrix operations called `numpy.mat`

Example

Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6:

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

print(arr)
```

Try it Yourself »

Check Number of Dimensions?

NumPy Arrays provides the `ndim` attribute that returns an integer that tells us how many dimensions the array have.

Example

Check how many dimensions the arrays have:

```
import numpy as np

a = np.array(42)
b = np.array([1, 2, 3, 4, 5])

print(a.ndim)
print(b.ndim)
```

Higher Dimensional Arrays

An array can have any number of dimensions.

When the array is created, you can define the number of dimensions by using the `ndmin` argument.

Example

Create an array with 2 dimensions and verify that it has 5 dimensions:

```
import numpy as np

arr = np.array([1, 2, 3, 4], ndmin=2)
#arr = np.array([[1, 2, 3, 4]])
print(arr)
print('number of dimensions :', arr.ndim)
```

Try it Yourself »

Exercise:

Insert the correct method for creating a NumPy array.

```
arr = np.      ([1, 2, 3, 4, 5])
```

NumPy Array Indexing

Access Array Elements

Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

Example

Get the first element from the following array:

```
import numpy as np

arr = np.array([1, 2, 3, 4])
```

```
print(arr[0])
```

Try it Yourself »

Example

Get the second element from the following array.

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr[1])
```

Try it Yourself »

Example

Get third and fourth elements from the following array and add them.

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr[2] + arr[3])
```

Try it Yourself »

ADVERTISEMENT

Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.

Example

Access the element on the first row, second column:

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('2nd element on 1st row: ', arr[1][1]
```

Try it Yourself »

Example

Access the element on the 2nd row, 5th column:

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('5th element on 2nd row: ', arr[1][4])
```

Try it Yourself »

Machine Learning

Machine Learning is making the computer learn from studying data and statistics.

Machine Learning is a step into the direction of artificial intelligence (AI).

Machine Learning is a program that analyses data and learns to predict the outcome.

Data Set

In the mind of a computer, a data set is any collection of data. It can be anything from an array to a complete database.

Example of an array:

```
[99,86,87,88,111,86,103,87,94,78,77,85,86]
```

What is Train/Test

Train/Test is a method to measure the accuracy of your model.

It is called Train/Test because you split the data set into two sets: a training set and a testing set.

80% for training, and 20% for testing.

You *train* the model using the training set.

You *test* the model using the testing set.

Train the model means *create* the model.

Test the model means test the accuracy of the model.

The `DecisionTreeClassifier` works based on the decision tree algorithm, a supervised learning algorithm used for classification tasks. Here's a simplified explanation of how the algorithm works:

1. Decision Tree Structure:

A decision tree is a hierarchical structure where each node represents a decision based on a specific feature. The tree is built recursively by splitting the data at each node based on the feature that provides the best separation according to a certain criterion.

2. Splitting Criteria:

The decision tree algorithm selects the best feature to split the data at each node. The selection is based on a splitting criterion, often using metrics like Gini impurity, entropy, or information gain. These criteria measure the homogeneity of the target variable within each subset created by the split.

MACHINE LEARNING CODE

```
import numpy as np

from sklearn.tree import DecisionTreeClassifier

# Input data

X = np.array([[33, 65, 77, 22, 11, 99, 44, 66, 87]])

y = np.array(["fail", "pass", "pass", "fail", "fail", "pass", "fail", "pass", "pass"])

# Transpose X to have samples along the rows

X = X.T  # or X = np.transpose(X)

# Create a DecisionTreeClassifier

dt_classifier = DecisionTreeClassifier(random_state=42)

# Train the classifier on the training data

dt_classifier.fit(X, y)

# Make predictions on the testing data (single sample)

new_data_point = np.array([[55]])  # Transpose if necessary

predictions = dt_classifier.predict(new_data_point)

print(predictions)
```