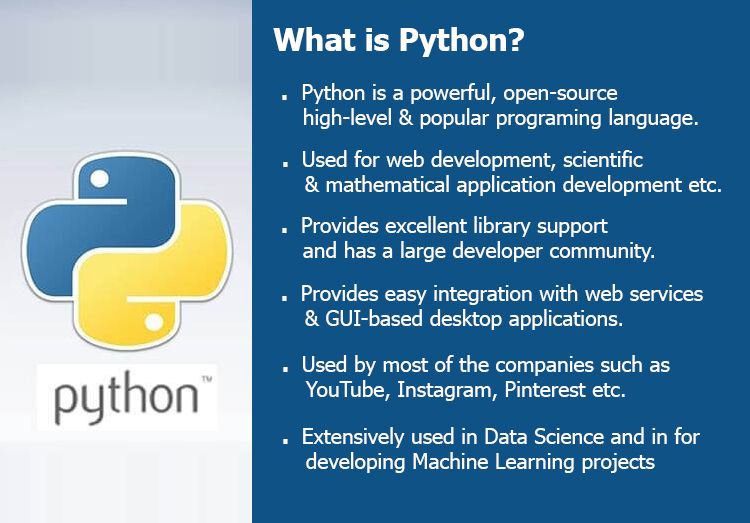
**What is Python?**

****

**Python History**

* **Python started in the late 1980s, created by Guido Van Rossum in December 1989 in the Netherlands.**
* **Python's first version was released in February 1991, and Python 1.0 came out in 1994 with features like lambda, map, filter, and reduce.**
* **Python 2.0 added features like list comprehensions and a garbage collection system.**
* **Python 3.0 was released on December 3, 2008, to fix major issues in the language.**
* **Python was inspired by other programming languages like ABC and Modula-3.**

**PythonVersion**

****

**Python Applications Area**

****

**Web and GUI Applications**: Python is used to develop web (Django, Flask) and desktop GUI applications (Tkinter, Kivy).

**Software Development**: It supports software development processes like build control, testing, and management.

**Scientific and Numeric Computing**: Python is popular for scientific and numeric tasks with libraries like SciPy and Pandas.

**Business and Multimedia Applications**: Python is used for business applications (ERP systems like Tryton) and multimedia apps (e.g., TimPlayer).

**Enterprise and CAD Applications**: Python develops enterprise apps (OpenERP) and 3D CAD tools like Fandango.

**Python Features**

****

**Python provides lots of features that are listed below.**

**1) Easy to Learn and Use**

**Python is easy to learn and use. It is developer-friendly and high level programming language.**

**2) Expressive Language**

**Python language is more expressive means that it is more understandable and readable.**

**3) Interpreted Language**

**Python is an interpreted language i.e. interpreter executes the code line by line at a time. This makes debugging easy and thus suitable for beginners.**

**4) Cross-platform Language**

**Python can run equally on different platforms such as Windows, Linux, Unix and Macintosh etc. So, we can say that Python is a portable language.**

**5) Free and Open Source**

**Python language is freely available at**[**official web address**](https://www.python.org/)**. The source-code is also available. Therefore it is open source.**

**6) Object-Oriented Language**

**Python supports object oriented language and concepts of classes and objects come into existence.**

**7) Extensible**

**It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our python code.**

**8) Large Standard Library**

**Python has a large and broad library and provides rich set of module and functions for rapid application development.**

**9) GUI Programming Support**

**Graphical user interfaces can be developed using Python.**

**10) Integrated**

**It can be easily integrated with languages like C, C++, JAVA etc.**

**Python Variables**

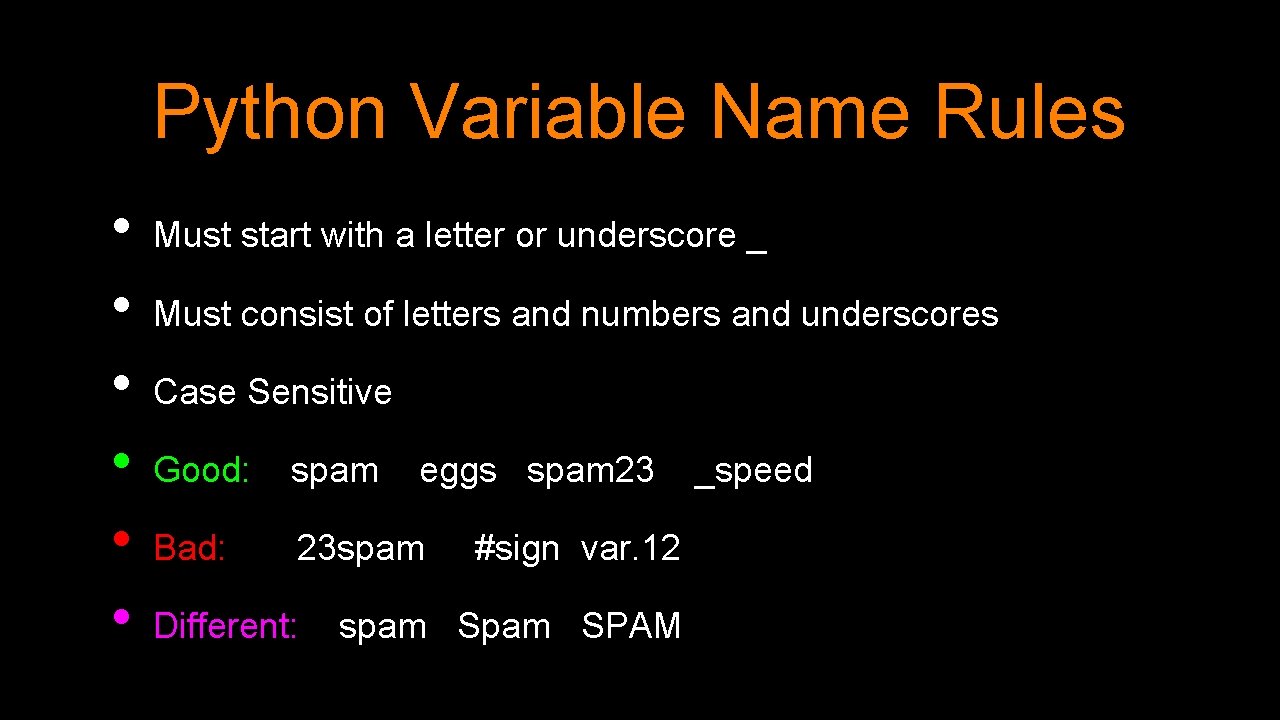
**Variable is a name that is used to refer to memory location. Python variable is also known as an identifier and used to hold value.**

**Rules for Naming Variables**

* **In Python, we don't need to specify the type of variable because Python is a type infer language and smart enough to get variable type.**
* **Variable names can be a group of both letters and digits, but they have to begin with a letter or an underscore.**
* **Variables are case sensitive.**
* **Spaces are not allowed**
* **No special characters are used except underscore.**

**Right : rate, rate\_of\_interest, Rate**

**Wrong: 1rate, rate of interest, rate#**

****

**Note - Variable name should not be a keyword.**

**Declaring Variable and Assigning Values**

**a=10**

**b=20.25**

**c=”india”**

**Multiple Assignment**

**1.Assigning single value to multiple variables**

**Example:**

**x=y=z=50**

**print(x)**

**print (y)**

**print (z)**

**Output:**

**50**

**50**

**2.Assigning multiple values to multiple variables:**

**Example:**

**a,b,c=5,10,15**

**print( a )**

**print (b)**

**print (c)**

**Output:**

**5**

**10**

**15**

**The values will be assigned in the order in which variables appears.**

**Comments**

**Compiler will ignore the statement inside the comments.**

**Python supports two types of comments:**

**1) Single lined comment:**

**In case user wants to specify a single line comment, then comment must start with #**

**Ex: # This is single line comment.**

**2) Multi lined Comment:**

**‘’’ ‘’’ Multi lined comment can be given inside triple quotes.**

**Example:**

**''' This**

**Is**

**Multiline comment'''**

**Example:**

**#single line comment**

**print ("Hello Python"  )**

**'''This is**

**multiline comment'''**

**Python Keywords**

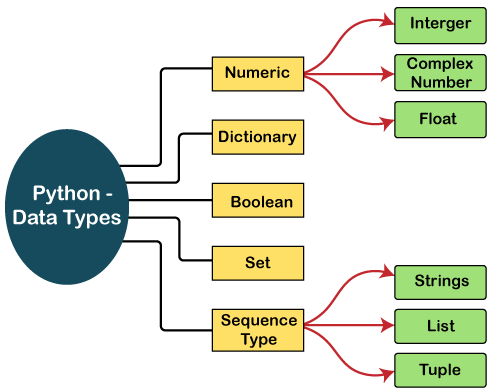
**Python Keywords are special reserved words which convey a special meaning to the compiler/interpreter. Each keyword have a special meaning and a specific operation. These keywords can't be used as variable.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **True** | **False** | **None** | **and** | **as** |
| **asset** | **def** | **class** | **continue** | **break** |
| **else** | **finally** | **elif** | **del** | **except** |
| **global** | **for** | **if** | **from** | **import** |
| **raise** | **try** | **or** | **return** | **pass** |
| **nonlocal** | **in** | **not** | **is** | **lambda** |

**Python Data Types**

**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:**

****

**Integer**

**Integer value can be any length such as integers 10, 2, 29, -20, -150 etc. Python has no restriction on the length of an integer. Its value belongs to int**

**Float**

**Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc. It is accurate upto 15 decimal points.**

**Complex**

**A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts, respectively. The complex numbers like 2.14j, 2.0 + 2.3j, etc.**

**Boolean**

**Boolean type provides two built-in values, True and False. These values are used to determine the given statement true or false. It denotes by the class bool. True can be represented by any non-zero value or 'T' whereas false can be represented by the 0 or 'F'.**

**Python Operators**

**Operators are particular symbols that are used to perform operations on operands. It returns result that can be used in application.**

**Example:**

**4 + 5 = 9**

**Here 4 and 5 are Operands and (+) , (=) signs are the operators. This expression produces the output 9.**

**Types of Operators**

**Python supports the following operators**

1. **Arithmetic Operators.**
2. **Relational Operators.**
3. **Logical Operators.**
4. **Assignment Operators.**
5. **Membership Operators.**
6. **Identity Operators.**
7. **Arithmetic Operators**

**The following table contains the arithmetic operators that are used to perform arithmetic operations.**

|  |  |
| --- | --- |
| **Operators** | **Description** |
| **+** | **To perform addition** |
| **-** | **To perform subtraction** |
| **\*** | **To perform multiplication** |
| **/** | **To perform division** |
| **//** | **Perform Floor division(gives integer value after division)** |
| **%** | **To return remainder after division(Modulus)** |
| **\*\*** | **Perform exponent(raise to power)** |

**Example:**

**a=10**

**b=3**

**c=a+b**

**d=a-b**

**e=a\*b**

**f=a/b**

**g=a//b**

**h=a%b**

**i=a\*\*b**

**print(c, d, e, f, g, h, i)**

**output:**

**13  7  30  3.3333333333333335  3  1  1000**

1. **Relational Operators**

**The following table contains the relational operators that are used to check relations.**

|  |  |
| --- | --- |
| **Operators** | **Description** |
| **<** | **Less than** |
| **>** | **Greater than** |
| **<=** | **Less than or equal to** |
| **>=** | **Greater than or equal to** |
| **==** | **Equal to** |
| **!=** | **Not equal to** |
|  |  |

**Example:**

**print(10<20) # True**

**print( 10>20 ) # False**

**print(10<=10) # True**

**print( 20>=15) # True**

**print(5==6) #False**

**print(5!=6) # True**

1. **Logical Operators**

**It joins two or more conditions.**

**The following table contains the arithmetic operators that are used to perform Logical operations.**

|  |  |
| --- | --- |
| **Operators** | **Description** |
| **And** | **Logical AND(When both conditions are true output will be true)** |
| **Or** | **Logical OR (If any one condition is true output will be true)** |
| **Not** | **Logical NOT(Compliment the condition i.e., reverse)** |

**Example**

**a=5>4 and 3>2**

**print (a )**

**b=5>4 or 3<2**

**print (b)**

**c=not(5>4)**

**print (c)**

**Output:**

**True**

**True**

**False**

1. **Assignment Operators**

**The following table contains the assignment operators that are used to assign values to the variables.**

|  |  |  |
| --- | --- | --- |
| **Operators** | **Description** |  |
| **=** | **Assignment** | **a=5** |
| **/=** | **Divide and Assign** | **a=a/5 or a/=5** |
| **+=** | **Add and assign** | **a=a+5 or a+=5** |
| **-=** | **Subtract and Assign** | **a=a-5 or a-=5** |
| **\*=** | **Multiply and assign** | **a=a\*5 or a\*=5** |
| **%=** | **Modulus and assign** | **a=a%5 or a%=5** |
| **\*\*=** | **Exponent and assign** | **a=a\*\*5 or a\*\*=5** |
| **//=** | **Floor division and assign** | **a=a//5 or a//=5** |

**Example:**

**c=10**

**print( c)**

**c+=5**

**print(c)**

**c-=5**

**print(c)**

**c\*=2**

**print( c )**

**c/=2**

**print( c  )**

**c%=3**

**print(c)**

**c=5**

**c\*\*=2**

**print( c  )**

**c//=2**

**print(c  )**

1. **Membership Operators**

**The following table contains the membership operators.**

|  |  |
| --- | --- |
| **Operators** | **Description** |
| **In** | **Returns true if a variable is in sequence of another variable, else false.** |
| **not In** | **Returns true if a variable is not in sequence of another variable, else false.** |

**Example:**

**a=10**

**list=[10,20,30,40,50]**

**if (a in list):**

**print ("a is in given list"  )**

**else:**

**print ("a is not in given list"  )**

**b=20**

**list=[10,20,30,40,50]**

**if(b not in list):**

**print ("b is not given in list"  )**

**else:**

**print ("b is given in list"  )**

**Output:**

**a is in given list**

**b is given in list**

1. **Identity Operators**

**The following table contains the identity operators.**

|  |  |
| --- | --- |
| **Operators** | **Description** |
| **Is** | **Returns true if identity of two operands are same, else false** |
| **is not** | **Returns true if identity of two operands are not same, else false.** |

**Example:**

**a=20**

**b=20**

**if( a is b):**

**print ('a,b have same identity' )**

**else:**

**print ( 'a, b are different' )**

**b=10**

**if( a is not b):**

**print ('a,b have different identity' )**

**else:**

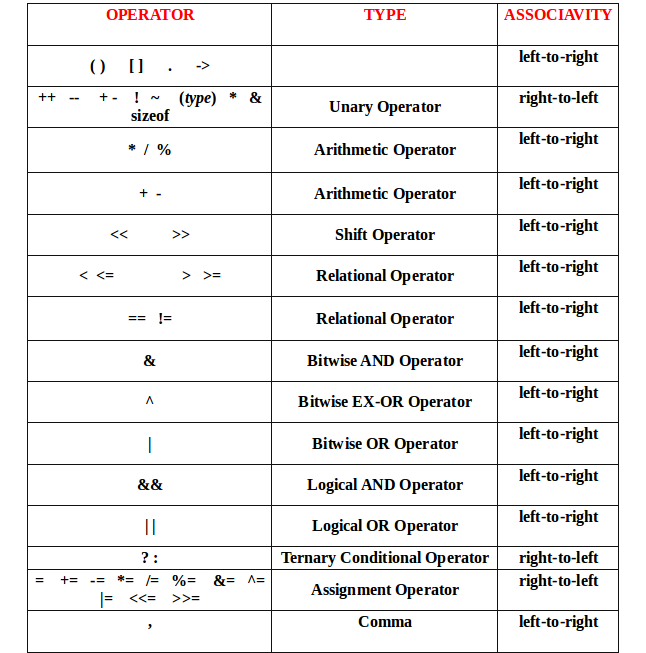
**print ('a,b have same identity')**

**Output**

**a,b have same identity**

**a,b have different identity**

**Python operator precedence and associavity**

****

**CONTROL STATEMENTS**

**Python If Statements**

**The Python if statement is a statement which is used to test specified condition. We can use if statement to perform conditional operations in our Python application.**

**There are various types of if statements in Python.**

* **if statement**
* **if-else statement**
* **if – elif statement**

**Python If Statement Syntax**

**if(condition):**

**statements**

**Example**

**a=5**

**if(a<10):**

**print('a is less than 10')**

**Python If Else Statements**

**The If statement is used to test specified condition and if the condition is true, if block executes, otherwise else block executes.**

**Python If Else Syntax**

**if(condition):**

**true statements**

**else:**

**False statements**

**Example**

**a=5**

**if(a<10):**

**print('a is less than 10 ')**

**else:**

**print('a is not less than 10 ')**

**Python if elif statement**

**In python, we can use if elif to check multiple conditions. Python provides elif keyword to make nested If statement.**

**Python if elif Syntax**

**If statement:**

**Body**

**elif statement:**

**Body**

**else:**

**Body**

**Example**

**a=5**

**if(a<10):**

**print('a is less than 10 ')**

**elif(a>10):**

**print('a is greater than 10 ')**

**else:**

**print('a is equal to 10')**

**# if and**

**m=30**

**p=40**

**c=50**

**if(m>=35 and p>=35 and c>=35):**

**print('pass')**

**else:**

**print('fail')**

**output:**

**fail**

**# if or**

**a=10**

**b=30**

**if(a<20 or b<20):**

**print("true")**

**else:**

**print('false')**

**output:**

**true**

|  |
| --- |
|  |

**LOOPS**

**For Loop**

**Python for loop is used to iterate the elements of a collection in the order that they appear. This collection can be a sequence(list or string).**

**Python For Loop Syntax**

**for <variable> in <sequence>:**

**Python For Loop Simple Example**

**for i in range(10):**

**print(i)**

**output:**

**0 1 2 3 4 5 6 7 8 9**

**for i in range (1,10):**

**print (i,end=’ ‘)**

**Output:**

**1 2 3 4 5 6 7 8 9**

**// step value**

**for i in range(1,20,2):**

**print(i, end=' ')**

**output:**

**1 3 5 7 9 11 13 15 17 19**

**# print values from 10 to 1**

**for i in range(10,0,-1):**

**print(i, end=' ')**

**output:**

**10 9 8 7 6 5 4 3 2 1**

**Python Nested For Loops**

**Loops defined within another Loop are called Nested Loops. Nested loops are used to iterate matrix elements or to perform complex computation.**

**When an outer loop contains an inner loop in its body it is called Nested Looping.**

**Python Nested For Loop Syntax**

**for  <expression>:**

**for <expression>:**

**Body**

**Example**

**for i in range(1,6):**

**for j in range(1,6):**

**print(j,end=’ ‘)**

**print()**

**Output:**

**1 2 3 4 5**

**1 2 3 4 5**

**1 2 3 4 5**

**1 2 3 4 5**

**1 2 3 4 5**

**for i in range(1,6):**

**for j in range(5,0,-1):**

**print(j,end=' ')**

**print()**

**Output:**

**5 4 3 2 1**

**5 4 3 2 1**

**5 4 3 2 1**

**5 4 3 2 1**

**5 4 3 2 1**

**Python While Loop**

**In Python, while loop is used to execute number of statements or body till the specified condition is true. Once the condition is false, the control will come out of the loop.**

**Python While Loop Syntax**

**while <expression>:**

**Body**

**Here, loop Body will execute till the expression passed is true. The Body may be a single statement or multiple statement.**

**Python While Loop Example**

**i=1**

**while(i<=10):**

**print(i,end=' ')**

**i=i+1**

**Output:**

**1 2 3 4 5 6 7 8 9 10**

**i=10**

**while(i>=1):**

**print(i,end=' ')**

**i=i-1**

**output:**

**10 9 8 7 6 5 4 3 2 1**

**Python Break**

**Break statement is a jump statement which is used to transfer execution control. It breaks the current execution and in case of inner loop, inner loop terminates immediately.**

**Example**

**for i in range(10):**

**if (i==4):**

**break**

**print (i)**

**Output:**

**0 1 2 3**

**Python Continue Statement**

**Python Continue Statement is a jump statement which is used to skip execution of current iteration. After skipping, loop continue with next iteration.**

**We can use continue statement with for as well as while loop in Python.**

**Example**

**for i in range(11):**

**if i==5 :**

**continue**

**print(i)**

**Output:**

**0**

**1**

**2**

**3**

**4**

**6**

**7**

**8**

**9**

**10**

**Python Pass**

**In Python, pass keyword is used to execute nothing; it means, when we don't want to execute code, the pass can be used to execute empty.**

**Python Pass Syntax**

**pass**

**Python Pass Example**

**for i in range(5):**

**if i==3:**

**pass**

**print (i)**

**Output:**

**0**

**1**

**2**

**3**

**4**

**Sequences**

**Sequences allows you to store multiple values in an organized and efficient fashion.**

**Some basic sequence type classes in python are**

* **Lists**
* **Tuples**
* **Sets**
* **Dictionaries**

**Lists**

* **Python lists are the data structure that is capable of holding different type of data.**
* **Python lists are mutable i.e., Python will not create a new list if we modify an element in the list.**
* **A list can be composed by storing a sequence of different type of values separated by commas. It is enclosed between square([]) brackets.**
* **The elements are stored in the index basis with starting index as 0.**
* **It is a container that holds other objects in a given order. Different operation like insertion and deletion can be performed on lists.**

**# Accessing list values**

**li=[1,2,3,4,5,6,7,8,9,10]**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| **-10** | **-9** | **-8** | **-7** | **-6** | **-5** | **-4** | **-3** | **-2** | **-1** |

**print(li) # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]**

**print(li[0]) # 1**

**print(li[0],li[2]) # 1 3**

**print(li[0:4]) # [1, 2, 3, 4]**

**print(li[3:7]) # [4, 5, 6, 7]**

**print(li[0:]) # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]**

**print(li[:7]) # [1, 2, 3, 4, 5, 6, 7]**

**print(li[0:9:2]) # [1, 3, 5, 7, 9]**

**print(li[-1:-5:-1]) # [10, 9, 8, 7]**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| **-10** | **-9** | **-8** | **-7** | **-6** | **-5** | **-4** | **-3** | **-2** | **-1** |

**# list update**

**li=[1,2,3,4,5,6,7,8,9,10]**

**li[0]=100**

**print(li) #100,2,3,4,5,6,7,8,9,10**

**li[3:6]=300,400,500**

**print(li) # [100, 2, 3, 300, 400, 500, 7, 8, 9, 10]**

**# Adding two lists**

**l1=[1,2,3,4,5]**

**l2=[6,7,8,9,10]**

**l3=l1+l2**

**print(l3) # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]**

**#Replicating list - repeating**

**li=[1,2,3,4,5]**

**print(li\*3) # [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]**

**# append - adding a single element**

**li=[1,2,3,4,5]**

**li.append(6)**

**print(li) # [1, 2, 3, 4, 5, 6]**

**# Appending two elements**

**li=[1,2,3,4,5]**

**li.append(6,7)**

**print(li) # error**

**# Appending a list**

**li=[1,2,3,4,5]**

**li.append([6,7,8])**

**print(li) # [1, 2, 3, 4, 5, [6, 7, 8]]**

**# Extending a list**

**li=[1,2,3,4,5]**

**li.extend([6,7,8])**

**print(li) # [1, 2, 3, 4, 5, 6, 7, 8]**

**# Extend an element**

**li=[1,2,3,4,5]**

**li.extend(6)**

**print(li) # error**

**# Insert an element**

**li=[1,2,3,4,5]**

**li.insert(4,40)**

**print(li) # 1,2,3,4,40,5**

**# Remove an element**

**li=[10,20,30,40,50]**

**li.remove(30)**

**print(li) # [10, 20, 40, 50]**

**# Remove mulitple values**

**li=[10,20,30,40,50]**

**li.remove(30,40)**

**print(li) # error**

**# Deleting one or multiple values**

**li=[10,20,30,40,50,60,70,80,90,100]**

**del li[3]**

**print(li) # [10, 20, 30, 50, 60, 70, 80, 90, 100]**

**del li[3:6]**

**print(li) # [10, 20, 30, 80, 90, 100]**

**# Removing last element by pop method**

**li=[1,2,3,4,5,6,7,8,9,10]**

**li.pop()**

**print(li) # [1, 2, 3, 4, 5, 6, 7, 8, 9]**

**li.pop(3)**

**print(li) # [1, 2, 3, 5, 6, 7, 8, 9]**

**# Sorting a list**

**li=[15,23,34,14,5,36,17,38,79,10]**

**li.sort()**

**print(li) # [5, 10, 14, 15, 17, 23, 34, 36, 38, 79]**

**li.sort(reverse=True)**

**print(li) # [79, 38, 36, 34, 23, 17, 15, 14, 10, 5]**

**# Reverse a list**

**li=[1,2,3,4,5,6,7,8]**

**li.reverse()**

**print(li) # [8, 7, 6, 5, 4, 3, 2, 1]**

**# Count values in a list**

**li=[10,20,30,10,20,30,10,20,30]**

**print(li.count(10)) # 3**

**#Remove all items in a list**

**li=[10,20,30,10,20,30,10,20,30]**

**li.clear()**

**print(li) # []**

**# Index for a given value**

**li=[10,20,30,10,20,30,10,20,30,10,20,30]**

**print(li.index(20)) # 1**

**# list copy**

**l1=[1,2,3,4,5,6]**

**l2=l1**

**print(l2) # [1, 2, 3, 4, 5, 6]**

**# user input**

**li=[]**

**for i in range(1,6):**

**a=int(input('enter a value '))**

**li.append(a)**

**print(li)**

| **Operation** | **Description** |
| --- | --- |
| **x in l1** | **Check if the list l1 contains item x.** |
| **x not in l2** | **Check if list l1 does not contain item x.** |
| **l1 + l2** | **Concatenate the lists l1 and l2. Creates a new list containing the items from l1 and l2.** |
| **l1 \* 5** | **Repeat the list l1 5 times.** |
| **l1[i]** | **Get the item at index i. Example l1[2] is 30.** |
| **l1[i:j]** | **List slicing. Get the items from index i up to index j (excluding j) as a List. An example l1[0:2] is [10, 20]** |
| **l1[i:j:k]** | **List slicing with step. Returns a List with the items from index i up to index j taking every k-th item. An example l1[0:4:2] is [10, 30].** |
| **len(l1)** | **Returns a count of total items in a list.** |
| **l2.count(60)** | **Returns the number of times a particular item (60) appears in a list. The answer is 2.** |
| **l1.index(30)** | **Returns the index number of a particular item (30) in a list. The answer is 2.** |
| **l1.index(30, 2, 5)** | **Returns the index number of a particular item (30) in a list. But search Returns the item with maximum value from a list. The answer is 60 only from index number 2 to 5.** |
| **min(l1)** | **Returns the item with a minimum value from a list. The answer is 10.** |
| **max(l1)** | **Returns the item with maximum value from a list. The answer is 60.** |
| **l1.append(100)** | **Add item at the end of the list** |
| **l1.append([2, 5, 7])** | **Append the nested list at the end** |
| **l1[2] = 40** | **Modify the item present at index 2** |
| **l1.remove(40)** | **Removes the first occurrence of item 40 from the list.** |
| **pop(2)** | **Removes and returns the item at index 2 from the list.** |
| **l1.clear()** | **Make list empty** |
| **l3= l1.copy()** | **Copy l1 into l2** |

**TUPLE**

* **A tuple is a sequence of immutable objects, therefore tuple cannot be changed.**
* **The objects are enclosed within parenthesis and separated by comma.**
* **Tuple is similar to list. Only the difference is that list is enclosed between square bracket, tuple between parenthesis and List have mutable objects whereas Tuple have immutable objects.**

**Accessing Tuple Values**

**tu=(1,2,3,4,5,6,7,8,9,10)**

**print(tu[0]) # 1**

**print(tu[0],tu[3]) # 1 4**

**print(tu[3:7]) # (4, 5, 6, 7)**

**print(tu[-4:-1]) # (7, 8, 9)**

**print(tu[-1:-5:-1]) # (10, 9, 8, 7)**

**# Adding two Tuples**

**tu1=(1,2,3,4,5)**

**tu2=(5,6,7,8,9)**

**tu3=tu1+tu2**

**print(tu3) # (1, 2, 3, 4, 5, 5, 6, 7, 8, 9)**

**# Replicating Tuple - Repeating**

**tu=(1,2,3,4,5)**

**print(tu\*2) # (1, 2, 3, 4, 5, 1, 2, 3, 4, 5)**

**# Updating tuple - not possible because tuple values cannot be modified**

**tu=(1,2,3,4,5)**

**tu[0]=100**

**print(tu) # error**

**# sort, reverse, append, insert, remove**

**tu=(1,2,3,4,5)**

**tu.sort() # error**

**tu.reverse() # error**

**tu.append(5) # error**

**tu.insert(30,3) # error**

**tu.remove(3) # error**

**SETS**

* **The set in python can be defined as the unordered collection of various items enclosed within the curly braces.**
* **The elements of the set cannot be duplicate. The elements of the python set must be immutable.**
* **Unlike other collections in python, there is no index attached to the elements of the set, i.e., we cannot directly access any element of the set by the index.**
* **However, we can print them all together or we can get the list of elements by looping through the set.**

**set1 = {1, 2, 3}**

**print(set1) # {1, 2, 3}**

**# set of mixed datatypes**

**set1 = {1.0, "Hello", (1, 2, 3)}**

**print(set1) # {1.0, (1, 2, 3), 'Hello'}**

**# set cannot have duplicates**

**set1 = {1, 2, 3, 4, 3, 2}**

**print(set1) # {1, 2, 3, 4}**

**# we can make set from a list**

**set1 = set([1, 2, 3, 2])**

**print(set1) # {1, 2, 3}**

**# creating an empty set**

**a = {}**

**a = set()**

**# add an element**

**set1={1,3}**

**set1.add(2)**

**print(set1) # {1, 2, 3}**

**# add multiple elements**

**set1={1,2}**

**set1.update([2, 3, 4]) # {1, 2, 3, 4}**

**print(set1)**

**# add list and set**

**set1.update([4, 5], {1, 6, 8})**

**print(set1) # {1, 2, 3, 4, 5, 6, 8}**

**# Removing elements from a set**

**# discard an element**

**set1={1,3,4,5,6}**

**set1.discard(4)**

**print(set1) # {1, 3, 5, 6}**

**# remove an element**

**set1={1,3,5,6}**

**set1.remove(6)**

**print(set1) # {1, 3, 5}**

**# discard an element**

**set1={1,2,3,5}**

**set1.discard(2)**

**print(set1) # {1, 3, 5}**

**discard() vs remove()**

**set1.discard(2) # displays no error**

**set1.remove(2) # displays error**

**Python Set Operations**

1. **Union**
2. **Intersection**
3. **Difference**
4. **Intersection\_update**

**# Set union method**

**A = {1, 2, 3, 4, 5}**

**B = {4, 5, 6, 7, 8}**

**print(A.union(B) ) # {1, 2, 3, 4, 5, 6, 7, 8}**

**# use | operator**

**print(A | B) # {1, 2, 3, 4, 5, 6, 7, 8}**

**# Set Intersection**

**A = {1, 2, 3, 4, 5}**

**B = {4, 5, 6, 7, 8}**

**# intersection method**

**print(A.intersection(B)) # {4, 5}**

**print(B.intersection(A)) # {4, 5}**

**# use & operator**

**A = {1, 2, 3, 4, 5}**

**B = {4, 5, 6, 7, 8}**

**print(A & B) # {4, 5}**

**Set Difference**

**# difference method**

**print(A.difference(B) ) # {1, 2, 3}**

**print(B.difference(A) ) # {8, 6, 7}**

**# use - operator on B**

**B – A # {8, 6, 7}**

**Set Symmetric Difference**

**A = {1, 2, 3, 4, 5}**

**B = {4, 5, 6, 7, 8}**

**# use symmetric\_difference function on A**

**print(A.symmetric\_difference(B)) # {1, 2, 3, 6, 7, 8}**

**# use symmetric\_difference function on B**

**B.symmetric\_difference(A) # {1, 2, 3, 6, 7, 8}**

**# use ^ operator**

**print(A ^ B) # {1, 2, 3, 6, 7, 8}**

**Set intersection\_update()**

**A = {1, 2, 3, 4, 5}**

**B = {4, 5, 6, 7, 8}**

**A.intersection\_update(B)**

**print(A) # A = {4,5}**

**Set issubset()**

**A = {1, 2, 3}**

**B = {1, 2, 3, 4, 5}**

**C = {1, 2, 4, 5}**

**print(A.issubset(B)) # True**

**print(B.issubset(A)) # False**

**print(A.issubset(C)) # False**

**print(C.issubset(B)) # True**

**Set issuperset()**

**A = {1, 2, 3, 4, 5}**

**B = {1, 2, 3}**

**C = {1, 2, 3}**

**print(A.issuperset(B)) # Returns True**

**print(B.issuperset(A)) # Returns False**

**print(C.issuperset(B)) # Returns True**

**DICTIONARY**

**Python Dictionary**

* **Dictionary is an unordered set of key and value pair.**
* **It is an container that contains data, enclosed within curly braces.**
* **The pair i.e., key and value is known as item.**
* **The key passed in the item must be unique.**
* **The key and the value is separated by a colon(:).**
* **This pair is known as item. Items are separated from each other by a comma(,).**

**# dictionary with keys and values**

**di = {1: 'apple', 2: 'ball'}**

**# empty dictionary**

**di = {}**

**# dictionary with mixed keys**

**di = {'name': 'John', 1: [2, 4, 3]}**

**# using dict()**

**di = dict({1:'apple', 2:'ball'})**

**# from sequence having each item as a pair**

**di = dict([(1,'apple'), (2,'ball')])**

**# Accessing Elements from Dictionary**

**di = {'name': 'Jack', 'age': 26}**

**print(di['name']) # Jack**

**print(di['age']) # 26**

**# Trying to access keys which doesn't exist throws error**

**print(di['address']) # KeyError: 'address'**

**Changing and Adding Dictionary elements**

**di = {'name': 'Jack', 'age': 26}**

**# update value**

**di['age'] = 27**

**print(di) # {'age': 27, 'name': 'Jack'}**

**# add item**

**di['address'] = 'Downtown'**

**print(di) # {'address': 'Downtown', 'age': 27, 'name': 'Jack'}**

**# Removing elements from a dictionary**

**squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}**

**# remove a particular item, returns its value**

**print(squares.pop(4))**

**print(squares) # {1: 1, 2: 4, 3: 9, 5: 25}**

**# remove an arbitrary item, return (key,value)**

**print(squares.popitem())**

**print(squares) # Output: {1: 1, 2: 4, 3: 9}**

**# remove all items**

**squares.clear()**

**print(squares) Output: {}**

**# delete the dictionary itself**

**del squares**

**print(squares) # Throws Error**

**# fromkeys**

**marks = {}.fromkeys(['Math', 'English', 'Science'], 0)**

**print(marks) # Output: {'English': 0, 'Math': 0, 'Science': 0}**

**Python Dictionary get()**

**person = {'name': 'Phill', 'age': 22}**

**print('Name: ', person.get('name')) # Name: Phill**

**print('Age: ', person.get('age')) # Age: 22**

**# value is not provided**

**print('Salary: ', person.get('salary')) # Salary: None**

**Python Dictionary update()**

**d = {1: "one", 2: "three"}**

**d1 = {2: "two"}**

**d.update(d1)**

**print(d) # {1: 'one', 2: 'two'}**

**d1 = {3: "three"}**

**d.update(d1)**

**print(d) # {1: 'one', 2: 'two', 3: 'three'}**

| **Operations** | **Description** |
| --- | --- |
| **dict({'a': 10, 'b': 20})** | **Create a dictionary using a dict() constructor.** |
| **d2 = {}** | **Create an empty dictionary.** |
| **d1.get('a')** | **Retrieve value using the key name a.** |
| **d1.keys()** | **Returns a list of keys present in the dictionary.** |
| **d1.values()** | **Returns a list with all the values in the dictionary.** |
| **d1.items()** | **Returns a list of all the items in the dictionary with each key-value pair inside a tuple.** |
| **len(d1)** | **Returns number of items in a dictionary.** |
| **d1['d'] = 40** | **Update dictionary by adding a new key.** |
| **d1.update({'e': 50, 'f': 60})** | **Add multiple keys to the dictionary.** |
| **d1.setdefault('g', 70)** | **Set the default value if a key doesn’t exist.** |
| **d1['b'] = 100** | **Modify the values of the existing key.** |
| **d1.pop('b')** | **Remove the key b from the dictionary.** |
| **d1.popitem()** | **Remove any random item from a dictionary.** |
| **d1.clear()** | **Removes all items from the dictionary.** |
| **'key' in d1.keys()** | **Check if a key exists in a dictionary.** |
| **d1.update(d2)** | **Add all items of dictionary d2 into d1.** |
| **d3= {\*\*d1, \*\*d2}** | **Join two dictionaries.** |
| **d2 = d1.copy()** | **Copy dictionary d1 into d2.** |
| **max(d1)** | **Returns the key with the maximum value in the dictionary d1** |
| **min(d1)** | **Returns the key with the minimum value in the dictionary d1** |

**Python Strings**

**A string is a sequence of characters.**

**st1 = 'Hello'**

**print(st1) # Hello**

**st1= "Hello"**

**print(st1) # Hello**

**st1= '''Hello'''**

**print(st1) # Hello**

**# triple quotes string can extend multiple lines**

**my\_string = """Hello, welcome to**

**the world of Python"""**

**print(my\_string)**

**Output**

**Hello, welcome to**

**the world of Python**

**#Accessing string characters in Python**

**str = 'program'**

**print(str) # program**

**#first character**

**print('str[0] = ', str[0]) # str[0] = p**

**#last character**

**print('str[-1] = ', str[-1]) # str[-1] = m**

**#slicing 2nd to 5th character**

**print('str[1:5] = ', str[1:5]) # str[1:5] = rogr**

**#slicing 6th to 2nd last character**

**print('str[5:1:-1] = ', str[5:1:-1]) # str[5:-2] = argo**

**# index must be in range**

**str [15] # IndexError: string index out of range**

**How to change or delete a string**

**str = 'program'**

**str[5] = 'a' #TypeError: 'str' object does not support item assignment**

**# We cannot delete or remove characters from a string. But deleting the string entirely is possible using the del keyword.**

**del str**

**Two or More StringsConcatenation of**

**str1 = 'Hello'**

**str2 ='World!'**

**# using +**

**print(str1 + str2) # HelloWorld!**

**# using \***

**print(str1 \* 3) # HelloHelloHello**

**String Membership Test**

**print('a' in 'program')**

**True**

**print('at' not in 'battle')**

**False**

**# Python String Methods**

**# Python String capitalize()**

**str1 = "python is AWesome."**

**str2 = str1.capitalize()**

**print(str2) # Python is awesome**

**Python String swapcase()**

**str1 = "THIS SHOULD ALL BE LOWERCASE."**

**print(str1.swapcase()) # this should all be lowercase.**

**str2 = "this should all be uppercase."**

**print(str2.swapcase()) # THIS SHOULD ALL BE UPPERCASE.**

**str3 = "ThIs ShOuLd Be MiXeD cAsEd."**

**print(str3.swapcase()) # tHiS sHoUlD bE mIxEd CaSeD.**

**Python String title()**

**text = 'My favorite number is 25.'**

**print(text.title()) # My Favorite Number Is 25.**

**Python String lower()**

**string = "THIS SHOULD BE LOWERCASE!"**

**print(string.lower()) # this should be lowercase!**

**# Python String isalnum()**

**name = "M234onica"**

**print(name.isalnum()) # True**

**# contains whitespace**

**name = "M3onica Gell22er "**

**print(name.isspace()) # False**

**Working of isalnum()**

**name = "M0n1caG3ll3r"**

**if name.isalnum() == True:**

**print("All characters of string (name) are alphanumeric.")**

**else:**

**print("All characters are not alphanumeric.")**

**Output**

**All characters of string (name) are alphanumeric.**

**Python String isdigit()**

**s = "28212"**

**print(s.isdigit()) # True**

**# contains alphabets and spaces**

**s = "Mo3 nicaG el l22er"**

**print(s.isdigit()) # False**

**Python String islower()**

**s = 'this is good'**

**print(s.islower()) # True**

**s = 'th!s is a1so g00d'**

**print(s.islower()) # True**

**s = 'this is Not good'**

**print(s.islower()) # False**

**How to use islower() in a program?**

**s = 'this is good'**

**if s.islower() == True:**

**print('Does not contain uppercase letter.')**

**else:**

**print('Contains uppercase letter.')**

**output**

**Does not contain uppercase letter.**

**s = 'this is Good'**

**if s.islower() == True:**

**print('Does not contain uppercase letter.')**

**else:**

**print('Contains uppercase letter.')**

**Output**

**Contains uppercase letter.**

**Python String isnumeric()**

**s = '1242323'**

**print(s.isnumeric()) # True**

**Python String isspace()**

**s = ' \t'**

**print(s.isspace()) # True**

**s = ' a '**

**print(s.isspace()) # False**

**Python String join()**

**# .join() with lists**

**numList = ['1', '2', '3', '4']**

**separator = '+ '**

**print(separator.join(numList)) # 1+ 2+ 3+ 4**

**# .join() with tuples**

**numTuple = ('1', '2', '3', '4')**

**separator = ', '**

**print(separator.join(numTuple)) # 1, 2, 3, 4**

**s1 = 'abc'**

**s2 = '123'**

**# each element of s2 is separated by s1**

**# '1'+ 'abc'+ '2'+ 'abc'+ '3'**

**print('s1.join(s2):', s1.join(s2)) # s1.join(s2): 1abc2abc3**

**# each element of s1 is separated by s2**

**# 'a'+ '123'+ 'b'+ '123'+ 'b'**

**print('s2.join(s1):', s2.join(s1)) # s2.join(s1): a123b123c**

**The join() method with sets**

**# .join() with sets**

**test = {'2', '1', '3'}**

**s = ', '**

**print(s.join(test)) # 2, 3, 1**

**test = {'Python', 'Java', 'Ruby'}**

**s = '->->'**

**print(s.join(test)) # Python->->Ruby->->Java**

**# .join() with dictionaries**

**test = {'mat': 1, 'that': 2}**

**s = '->'**

**# joins the keys only**

**print(s.join(test)) # mat->that**

**test = {1: 'mat', 2: 'that'}**

**s = ', '**

**print(s.join(test)) # # this gives error since key isn't string**

**Python String find()**

**find() With No start and end Argument**

**quote = 'Let it be, let it be, let it be'**

**result = quote.find('let it')**

**print("Substring 'let it':", result) # Substring 'let it': 11**

**# find returns -1 if substring not found**

**result = quote.find('small')**

**print("Substring 'small ':", result) # Substring 'small ': -1**

**# How to use find()**

**if (quote.find('be,') != -1):**

**print("Contains substring 'be,'")**

**else:**

**print("Doesn't contain substring")**

**Output**

**Contains substring 'be,'**

**Python String replace()**

**Using replace()**

**song = 'cold, cold heart'**

**print(song.replace('cold', 'hurt')) # hurt, hurt heart**

**song = 'Let it be, let it be, let it be, let it be'**

**# replacing only two occurences of 'let'**

**print(song.replace('let', "don't let", 2)) # Let it be, don't let it be, don't let it be, let it be**

**String replace()**

**song = 'cold, cold heart'**

**replaced\_song = song.replace('o', 'e')**

**# The original string is unchanged**

**print('Original string:', song) # Original string: cold, cold heart**

**print('Replaced string:', replaced\_song) # Replaced string: celd, celd heart**

**Python List Comprehension**

## List Comprehension vs For Loop in Python

#### Example 1: Iterating through a string Using for Loop

**li = []**

**for i in 'human':**

**li.append(i)**

**print(li)**

**output will be:**

**['h', 'u', 'm', 'a', 'n']**

### Example 2: Iterating through a string Using List Comprehension

**li = [ i for i in 'human' ]**

**print( li)**

**the output will be:**

**['h', 'u', 'm', 'a', 'n']**

## Conditionals in List Comprehension

#### Example : Using if with List Comprehension

**number\_list = [ x for x in range(20) if x % 2 == 0]**

**print(number\_list)**

**output:**

**[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]**

#### Example : Nested IF with List Comprehension

**num\_list = [y for y in range(100) if y % 2 == 0 if y % 5 == 0]**

**print(num\_list)**

**output:**

**[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]**

#### Example : if...else With List Comprehension

**obj = ["Even" if i%2==0 else "Odd" for i in range(10)]**

**print(obj)**

**output:**

**['Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd']**

## Nested Loops in List Comprehension

#### Example : Transpose of a Matrix using List Comprehension

**matrix = [[1, 2], [3,4], [5,6], [7,8]]**

**transpose = [[row[i] for row in matrix] for i in range(2)]**

**print (transpose)**

**output:**

**[[1, 3, 5, 7], [2, 4, 6, 8]]**

**FUNCTIONS**

* **A Function is a self block of code.**
* **A Function can be called as a section of a program that is written once and can be executed whenever required in the program, thus making code reusability.**
* **A Function is a subprogram that works on data and produce some output.**

**Defining a Function:**

**A Function defined in Python should follow the following format:**

1. **Keyword def is used to start the Function Definition. Def specifies the starting of Function block.**
2. **def is followed by function-name followed by parenthesis.**
3. **Parameters are passed inside the parenthesis. At the end a colon is marked.**

**def add():**

**a=10**

**b=20**

**c=a+b**

**print(c)**

**add()**

**print('hello')**

**add()**

**Output:**

**30**

**hello**

**30**

**def add():**

**a=10**

**b=20**

**c=a+b**

**print(c)**

**def sub():**

**a=20**

**b=10**

**c=a-b**

**print(c)**

**add()**

**print('hello')**

**sub()**

**add()**

**# Function with 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.**

**def hi(name):**

**print('hi',name, 'how are you')**

**hi('amar')**

**hi('kumar')**

**hi('ram')**

**output:**

**hi amar how are you**

**hi kumar how are you**

**hi ram how are you**

**Example:**

**def add(a,b):**

**c=a+b**

**print(c)**

**add(5,8)**

**print('hello ')**

**add(4,6)**

**Output:**

**13**

**hello**

**10**

**#Function with return value**

* **The**[**Python**](https://www.javatpoint.com/python-tutorial)**return statement is used to return a value from a function. The user can only use the return statement in a function.**
* **It cannot be used outside of the Python function. A return statement includes the return keyword and the value that will be returned after that.**

**Example:**

**def add(a,b):**

**c=a+b**

**return c**

**x=add(5,8)**

**y=add(4,6)**

**print(x+y)**

**Output:**

**23**

**Types of Arguments**

**1) Positional argument**

**2) Keyword argument**

**3) Default argument.**

1. **Positional Arguments**

**def add(a,b):**

**print(a+b)**

**add(10,20)**

**print('one')**

**add(20,30)**

**Output:**

**30**

**one**

**50**

**2. Keyword Arguments**

**When we call a function with some values, these values get assigned to the arguments according to their position.**

**def add(a,b):**

**print(a+b)**

**add(b=10,a=20)**

**print('one')**

**add(20,30)**

**Output:**

**30**

**one**

**50**

**3. Default arguments**

* **Function arguments can have default values in Python.**
* **We can provide a default value to an argument by using the assignment operator (=).**

**def add(a,b=5):**

**print(a+b)**

**add(20)**

**add(10,20)**

**add(b=10,a=20)**

**Output:**

**25**

**30**

**30**

**Example:**

**def add(a,b):**

**c=a+b**

**print(c)**

**def sub(a,b):**

**c=a-b**

**print(c)**

**def mul(a,b):**

**c=a\*b**

**print(c)**

**add(10,20)**

**sub(40,25)**

**mul(10,4)**

**Anonymous Function or Lambda Functions**

**Python allows us to not declare the function in the standard manner, i.e., by using the def keyword. Rather, the anonymous functions are declared by using lambda keyword.**

**Example 1**

**x=lambda a:a\*a**

**print(x(5))**

**Output:**

**25**

**Example 2**

**x=lambda a,b:a+b**

**print(x(10,20))**

**output:**

**30**

**Example 3**

**x=lambda a,b,c:a+b+c**

**print(x(10,20,30))**

**output:**

**60**

**Filter Function:**

* **A lambda function is a small, anonymous function that is defined using the** lambda **keyword, followed by a list of arguments, a colon, and an expression.**
* **Lambda functions can have any number of arguments but only a single expression. The expression is evaluated and returned when the lambda function is called.**
* **Lambda functions are typically used for short operations that can be defined in a single line and do not require a full function definition.**

**Use of lambda function with filter**

**a= [1,2,3,4,10,123,22]**

**Oddlist = list(filter(lambda x:(x%3 == 0),a))**

**print(Oddlist)**

**Output:**

**[3, 123]**

**Map Function:**

**The map() function executes a specified function for each item in an iterable. The item is sent to the function as a parameter.**

**Use of lambda function with map**

**a= [1,2,3,4,10,22,123]**

**new\_list = list(map(lambda x:x\*3,a))**

**print(new\_list)**

**Output:**

**[3, 6, 9, 12, 30, 66, 369]**

**\*args and \*\*kwargs in Python**

**In Python, we can pass a variable number of arguments to a function using special symbols. There are two special symbols:**

1. **\*args (Non Keyword Arguments)**
2. **\*\*kwargs (Keyword Arguments)**
3. **\*args(Non Keyword Arguments)**

**we are not sure about the number of arguments that can be passed to a function. Python has \*args which allow us to pass the variable number of non keyword arguments to function.**

**In the function, we should use an asterisk \* before the parameter name to pass variable length arguments.The arguments are passed as a tuple and these passed arguments make tuple inside the function with same name as the parameter excluding asterisk \*.**

**Using \*args to pass the variable length arguments to the function**

**def adder(\*num):**

**sum = 0**

**for n in num:**

**sum = sum + n**

**print("Sum:",sum)**

**adder(3,5)**

**adder(4,5,6,7)**

**adder(1,2,3,5,6)**

**output**

**Sum: 8**

**Sum: 22**

**Sum: 17**

1. **\*\*args(Keyword Arguments)**

**In the function, we use the double asterisk \*\* before the parameter name to denote this type of argument. The arguments are passed as a dictionary and these arguments make a dictionary inside function with name same as the parameter excluding double asterisk \*\*.**

**Using \*\*kwargs to pass the variable keyword arguments to the function**

**def intro(\*\*data):**

**print("\nData type of argument:",type(data))**

**for key, value in data.items():**

**print("{} is {}".format(key,value))**

**intro(Firstname="Sita", Lastname="Sharma", Age=22, Phone=1234567890)**

**intro(Firstname="John", Lastname="Wood", Email="johnwood@nomail.com", Country="Wakanda", Age=25, Phone=9876543210)**

**Output :**

**Data type of argument: <class 'dict'>**

**Firstname is Sita**

**Lastname is Sharma**

**Age is 22**

**Phone is 1234567890**

**Data type of argument: <class 'dict'>**

**Firstname is John**

**Lastname is Wood**

**Email is johnwood@nomail.com**

**Country is Wakanda**

**Age is 25**

**Phone is 9876543210**

**Modules**

**A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code.**

1. **Using import statement**
2. **Importing multiple modules**
3. **Using from.. import statement**
4. **import whole module**
5. **Using import statement:**

**def add(a,b):**

**c=a+b**

**print (c)**

**Save the file by the name addition.py. To import this file "import" statement is used.**

**import addition**

**addition.add(10,20)**

**addition.add(30,40)**

**Output:**

**30**

**70**

1. **Importing multiple modules:**

**1) msg.py:**

**def msg\_method():**

**print ("Today the weather is rainy"  )**

**2) display.py:**

**def display\_method():**

**print ("The weather is Sunny"  )**

**3) multiimport.py:**

**import msg,display**

**msg.msg\_method()**

**display.display\_method()**

**Output:**

**Today the weather is rainy**

**The weather is Sunny**

1. **Using from.. import statement:**

**1) area.py**

**def circle(r):**

**print (3.14\*r\*r)**

**def square(l):**

**print (l\*l)**

**def rectangle(l,b):**

**print (l\*b)**

**def triangle(b,h):**

**print (0.5\*b\*h )**

**2) area1.py**

**from area import square,rectangle**

**square(10)**

**rectangle(2,5)**

**Output:**

**100**

**4. To import whole module:**

**1) area.py**

**def circle(r):**

**print (3.14\*r\*r )**

**def square(l):**

**print (l\*l )**

**return**

**def rectangle(l,b):**

**print (l\*b)**

**def triangle(b,h):**

**print (0.5\*b\*h )**

**2) area1.py**

**from area import  \***

**square(10)**

**rectangle(2,5)**

**circle(5)**

**triangle(10,20)**

**Output:**

**100**

**10**

**78.5**

**100.0**

# What is if \_\_name\_\_ == \_\_main\_\_

* **Python files are called modules and they are identified by the .py file extension. A module can define functions, classes, and variables.**
* **So when the interpreter runs a module, the \_\_name\_\_ variable will be set as  \_\_main\_\_ if the module that is being run is the main program.**
* **But if the code is importing the module from another module, then the \_\_name\_\_  variable will be set to that module’s name.**

**File: a.py**

**def add(a,b):**

**c=a+b**

**print(c)**

**if \_\_name\_\_=="\_\_main\_\_":**

**add(10,20)**

**File: b.py**

**import a**

**a.add(20,30)**

**output:**

**50**

**File: a.py**

**def add(a,b):**

**c=a+b**

**print(c)**

**add(10,20)**

**File: b.py**

**import a**

**a.add(20,30)**

**output:**

**30**

**50**

**Global Variables, Local Variables & Non Local Variables**

1. **Global Variables**

**In Python, a variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.**

## Local Variables

**A variable declared inside the function's body or in the local scope is known as a local variable.**

## Nonlocal Variables

**Nonlocal variables are used in nested functions whose local scope is not defined. This means that the variable can be neither in the local nor the global scope.**

### Example 1: Create a Global Variable

**x = 10**

**def foo():**

**print("x inside:", x)**

**foo()**

**print("x outside:", x)**

**Output**

**x inside: 10**

**x outside: 10**

**x = 10**

**def foo():**

**x=20**

**print("x inside:", x)**

**foo()**

**print("x outside:", x)**

**Output**

**x inside: 20**

**x outside: 10**

**x = 10**

**def foo():**

**global x**

**x=20**

**print("x inside:", x)**

**foo()**

**print("x outside:", x)**

**output:**

**x inside: 20**

**x outside: 20**

**Non Local Variable**

**def outer():**

**x = 10**

**def inner():**

**x = 20**

**print("inner:", x)**

**inner()**

**print("outer:", x)**

**outer()**

**Output:**

**inner: 20**

**outer: 10**

**def outer():**

**x = 10**

**def inner():**

**nonlocal x**

**x = 20**

**print("inner:", x)**

**inner()**

**print("outer:", x)**

**outer()**

**Output:**

**inner: 20**

**outer: 20**

**Regular Expressions**

**Python RegEx**

**A Regular Expression (RegEx) is a sequence of characters that defines a search pattern. For example,**

**^a...s$**

**The above code defines a RegEx pattern. The pattern is: any five letter string starting with a and ending with s.**

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| **^a...s$** | **Abs** | **No match** |
| **Alias** | **Match** |
| **Abyss** | **No Match** |
| **Alias** | **No match** |
| **An abacus** | **No match** |

**Python has a module named re to work with RegEx. Here's an example:**

**import re**

**pattern = '^a...s$'**

**test\_string = 'abyss'**

**result = re.search(pattern, test\_string)**

**if result:**

**print("Search successful.")**

**else:**

**print("Search unsuccessful.")**

## Specify Pattern Using RegEx

**To specify regular expressions, metacharacters are used. In the above example, ^ and $ are metacharacters.**

### MetaCharacters

**Metacharacters are characters that are interpreted in a special way by a RegEx engine. Here's a list of metacharacters:**

**[] . ^ $ \* + ? {} () | \**

**[] - Square brackets**

**Square brackets specifies a set of characters you wish to match.**

|  |  |  |
| --- | --- | --- |
| Expression | String | Matched? |
| [abc] | **A** | **1 match** |
| **Ac** | **2 matches** |
| **Hey Jude** | **No match** |
| **abc de ca** | **5 matches** |

**Here, [abc] will match if the string you are trying to match contains any of the a, b or c.**

**You can also specify a range of characters using - inside square brackets.**

**[a-e] is the same as [abcde].**

**[1-4] is the same as [1234].**

**[0-39] is the same as [01239].**

**You can complement (invert) the character set by using caret ^ symbol at the start of a square-bracket.**

**[^abc] means any character except a or b or c.**

**[^0-9] means any non-digit character.**

**. – Period.**

**A period matches any single character (except newline '\n').**

|  |  |  |
| --- | --- | --- |
| Expression | String | Matched? |
| .. | **A** | **No match** |
| **Ac** | **1 match** |
| **Acd** | **1 match** |
| **Acde** | **2 matches (contains**  **4 characters)** |

**import re**

**pattern = 'A..c'**

**test\_string = 'Axyc'**

**result = re.search(pattern, test\_string)**

**if result:**

**print("Search successful.")**

**else:**

**print("Search unsuccessful.")**

**output:**

**Search successful.**

**^ - Caret**

**The caret symbol ^ is used to check if a string starts with a certain character.**

|  |  |  |
| --- | --- | --- |
| Expression | String | Matched? |
| ^a | **A** | **1 match** |
| **Abc** | **1 match** |
| **Bac** | **No match** |
| ^ab | **Abc** | **1 match** |
| **Acb** | **No match (starts with a but not followed by b)** |

**import re**

**pattern = '^a'**

**test\_string = 'abc'**

**result = re.search(pattern, test\_string)**

**if result:**

**print("Search successful.")**

**else:**

**print("Search unsuccessful.")**

**output:**

**Search successful.**

**$ - Dollar**

**The dollar symbol $ is used to check if a string ends with a certain character.**

|  |  |  |
| --- | --- | --- |
| Expression | String | Matched? |
| a$ | **A** | **1 match** |
| **Formula** | **1 match** |
| **Cab** | **No match** |

**import re**

**pattern = 'a$'**

**test\_string = 'Formula'**

**result = re.search(pattern, test\_string)**

**if result:**

**print("Search successful.")**

**else:**

**print("Search unsuccessful.")**

**output:**

**Search successful.**

**\* - Star**

**The star symbol \* matches zero or more occurrences of the pattern left to it.**

|  |  |  |
| --- | --- | --- |
| Expression | String | Matched? |
| ma\*n | **mn** | **1 match** |
| **man** | **1 match** |
| **maaan** | **1 match** |
| **main** | **No match (a is not followed by n)** |
| **woman** | **1 match** |

**import re**

**pattern = 'ma\*n'**

**test\_string = 'man'**

**result = re.search(pattern, test\_string)**

**if result:**

**print("Search successful.")**

**else:**

**print("Search unsuccessful.")**

**output:**

**Search successful.**

**+ - Plus**

**The plus symbol + matches one or more occurrences of the pattern left to it.**

|  |  |  |
| --- | --- | --- |
| Expression | String | Matched? |
| ma+n | Mn | No match (no a character) |
| Man | 1 match |
| maaan | 1 match |
| Main | No match (a is not followed by n) |
| woman | 1 match |

**import re**

**pattern = 'ma+n'**

**test\_string = 'man'**

**result = re.search(pattern, test\_string)**

**if result:**

**print("Search successful.")**

**else:**

**print("Search unsuccessful.")**

**output:**

**Search successful.**

**? - Question Mark**

**The question mark symbol ? matches zero or one occurrence of the pattern left to it.**

|  |  |  |
| --- | --- | --- |
| Expression | String | Matched? |
| ma?n | Mn | 1 match |
| Man | 1 match |
| Maaan | No match (more than one a character) |
| Main | No match (a is not followed by n) |
| Woman | 1 match |

**import re**

**pattern = 'ma?n'**

**test\_string = 'mn'**

**result = re.search(pattern, test\_string)**

**if result:**

**print("Search successful.")**

**else:**

**print("Search unsuccessful.")**

**output:**

**Search successful.**

**{} - Braces**

**Consider this code: {n,m}. This means at least n, and at most m repetitions of the pattern left to it.**

|  |  |  |
| --- | --- | --- |
| Expression | String | Matched? |
| a{2,3} | abc dat | No match |
| abc daat | 1 match (at daat) |
| aabc daaat | 2 matches (at aabc and daaat) |
| aabc daaaat | 2 matches (at aabc and daaaat) |

**Let's try one more example. This RegEx [0-9]{2, 4} matches at least 2 digits but not more than 4 digits**

|  |  |  |
| --- | --- | --- |
| Expression | String | Matched? |
| [0-9]{2,4} | ab123csde | 1 match (match at ab123csde) |
| 12 and 345673 | 2 matches (at 12 and 345673) |
| 1 and 2 | No match |

**import re**

**pattern = 'a{2,3}'**

**test\_string = 'abc daat'**

**result = re.search(pattern, test\_string)**

**if result:**

**print("Search successful.")**

**else:**

**print("Search unsuccessful.")**

**output:**

**Search successful.**

**| - Alternation**

**Vertical bar | is used for alternation (or operator).**

|  |  |  |
| --- | --- | --- |
| Expression | String | Matched? |
| a|b | Cde | No match |
| Ade | 1 match (match at ade) |
| acdbea | 3 matches (at acdbea) |

**Here, a|b match any string that contains either a or b**

**import re**

**pattern = 'a|b'**

**test\_string = 'ade'**

**result = re.search(pattern, test\_string)**

**if result:**

**print("Search successful.")**

**else:**

**print("Search unsuccessful.")**

**Output**

**Search successful.**

**() - Group**

**Parentheses () is used to group sub-patterns. For example, (a|b|c)xz match any string that matches either a or b or c followed by xz**

|  |  |  |
| --- | --- | --- |
| Expression | String | Matched? |
| (a|b|c)xz | ab xz | No match |
| Abxz | 1 match (match at abxz) |
| axz cabxz | 2 matches (at axzbc cabxz) |

**import re**

**pattern = '(a|b|c)xz'**

**test\_string = 'axz'**

**result = re.search(pattern, test\_string)**

**if result:**

**print("Search successful.")**

**else:**

**print("Search unsuccessful.")**

**Output**

**Search successful.**

**\ - Backslash**

**Backlash \ is used to escape various characters including all metacharacters. For example,**

**\$a match if a string contains $ followed by a. Here, $ is not interpreted by a RegEx engine in a special way.**

**If you are unsure if a character has special meaning or not, you can put \ in front of it. This makes sure the character is not treated in a special way.**

**import re**

**string = 'hello 12 hi 89. Howdy 34'**

**pattern = '\d+'**

**result = re.findall(pattern, string)**

**print(result)**

**Output**

**['12', '89', '34']**

**Special Sequences**

**Special sequences make commonly used patterns easier to write. Here's a list of special sequences:**

**\d - Matches any decimal digit. Equivalent to [0-9]**

|  |  |  |
| --- | --- | --- |
| Expression | String | Matched? |
| \d | 12abc3 | 3 matches (at 12abc3) |
| Python | No match |

**\D - Matches any non-decimal digit. Equivalent to [^0-9]**

|  |  |  |
| --- | --- | --- |
| Expression | String | Matched? |
| \D | 1ab34"50 | 3 matches (at 1ab34"50) |
| 1345 | No match |

**\s - Matches where a string contains any whitespace character. Equivalent to [ \t\n\r\f\v].**

|  |  |  |
| --- | --- | --- |
| Expression | String | Matched? |
| \s | Python RegEx | 1 match |
| PythonRegEx | No match |

**\S - Matches where a string contains any non-whitespace character. Equivalent to [^ \t\n\r\f\v].**

| Expression | String | Matched? |
| --- | --- | --- |
| \S | a b | 2 matches (at a b) |
|  | No match |

**\w - Matches any alphanumeric character (digits and alphabets). Equivalent to [a-zA-Z0-9\_]. By the way, underscore \_ is also considered an alphanumeric character.**

| Expression | String | Matched? |
| --- | --- | --- |
| \w | 12&": ;c | 3 matches (at 12&": ;c) |
| %"> ! | No match |

**\W - Matches any non-alphanumeric character. Equivalent to [^a-zA-Z0-9\_]**

|  |  |  |
| --- | --- | --- |
| Expression | String | Matched? |
| \W | 1a2%c | 1 match (at 1a2%c) |
| Python | No match |

## Python RegEx Functions

**Python has a module named re to work with regular expressions. To use it, we need to import the module.**

**import re**

**The module defines several functions and constants to work with RegEx.**

## re.findall()

**The re.findall() method returns a list of strings containing all matches.**

### Example:

**# Program to extract numbers from a string**

**import re**

**string = 'hello 12 hi 89. Howdy 34'**

**pattern = '\d+'**

**result = re.findall(pattern, string)**

**print(result)**

**Output**

**['12', '89', '34']**

**If the pattern is no found, re.findall() returns an empty list.**

## 

## re.split()

**The re.split method splits the string where there is a match and returns a list of strings where the splits have occurred.**

### Example :

**import re**

**string = 'Twelve:12 Eighty nine:89.'**

**pattern = '\d+'**

**result = re.split(pattern, string)**

**print(result)**

**Output**

**['Twelve:', ' Eighty nine:', '.']**

**If the pattern is no found, re.split() returns a list containing an empty string.**

**You can pass maxsplit argument to the re.split() method. It's the maximum number of splits that will occur.**

**import re**

**string = 'Twelve:12 Eighty nine:89 Nine:9.'**

**pattern = '\d+'**

**# maxsplit = 1**

**# split only at the first occurrence**

**result = re.split(pattern, string, 1)**

**print(result)**

**Output**

**['Twelve:', ' Eighty nine:89 Nine:9.']**

**By the way, the default value of maxsplit is 0; meaning all possible splits.**

## re.sub()

**The syntax of re.sub() is:**

**re.sub(pattern, replace, string)**

**The method returns a string where matched occurrences are replaced with the content of replace variable.**

### Example:

**# Program to remove all whitespaces**

**import re**

**string = 'abc 12 de 23 f45 6'**

**pattern = '\s+'**

**replace = ''**

**new\_string = re.sub(pattern, replace, string)**

**print(new\_string)**

**Output**

**abc12de23f456**

**If the pattern is not found, re.sub() returns the original string.**

**You can pass count as a fourth parameter to the re.sub() method. If omited, it results to 0. This will replace all occurrences.**

**import re**

**# multiline string**

**string = 'abc 12\**

**de 23 \n f45 6'**

**# matches all whitespace characters**

**pattern = '\s+'**

**replace = ''**

**new\_string = re.sub(r'\s+', replace, string, 1)**

**print(new\_string)**

**Output**

**# abc12de 23**

**# f45 6**

## re.subn()

**The re.subn() is similar to re.sub() expect it returns a tuple of 2 items containing the new string and the number of substitutions made.**

### Example :

**# Program to remove all whitespaces**

**import re**

**# multiline string**

**string = 'abc 12de 23 f45 6'**

**# matches all whitespace characters**

**pattern = '\s+'**

**# empty string**

**replace = ''**

**new\_string = re.subn(pattern, replace, string)**

**print(new\_string)**

**Output**

**('abc12de23f456', 4)**

## re.search()

**The re.search() method takes two arguments: a pattern and a string. The method looks for the first location where the RegEx pattern produces a match with the string.**

**If the search is successful, re.search() returns a match object; if not, it returns None.**

**match = re.search(pattern, str)**

### re.match()

**import re**

**pattern = '^a...s$'**

**test\_string = 'abyss'**

**result = re.search(pattern, test\_string)**

**if result:**

**print("Search successful.")**

**else:**

**print("Search unsuccessful.")**

### Example :

**import re**

**string = "Python is fun"**

**# check if 'Python' is at the beginning**

**match = re.match('Python', string)**

**if match:**

**print("pattern found inside the string")**

**else:**

**print("pattern not found")**

**Output**

**pattern found inside the string**

**Here, match contains a match object.**

**OOPs - Object Oriented Programming Language**

**Python is an Object-Oriented Programming language, so everything in Python is treated as an object. An object is a real-life entity. It is the collection of various data and functions that operate on those data.**

**Class:**

**The class is a user-defined data structure that binds the data members and methods into a single unit. Class is a blueprint or code template for object creation. Using a class, you can create as many objects as you want.**

**Object:**

**An object is an instance of a class. It is a collection of attributes (variables) and methods. We use the object of a class to perform actions.**

**Objects have two characteristics:**

**They have states and behaviors (object has attributes and methods attached to it) Attributes represent its state, and methods represent its behavior. Using its methods, we can modify its state.**

**Every object has the following property.**

* **Identity: Every object must be uniquely identified.**
* **State: An object has an attribute that represents a state of an object, and it also reflects the property of an object.**
* **Behavior: An object has methods that represent its behavior.**

**Constructor: A constructor is a special method used to create and initialize an object of a**[**class**](https://pynative.com/python-classes-and-objects/)**. This method is defined in the class.**

**class Cat:**

**def eat(self):**

**print('eating')**

**def sleep(self):**

**print('sleeping')**

**c1=Cat()**

**c1.eat()**

**c1.sleep()**

**c2=Cat()**

**c2.eat()**

**c2.sleep()**

**output:**

**eating**

**sleeping**

**eating**

**sleeping**

**Example**

**class A:**

**def one(self):**

**print('this is function one')**

**def two(self):**

**print('this is function two')**

**a1=A()**

**a1.one()**

**a1.two()**

**Output:**

**this is function one**

**this is function two**

**Example**

**class A:**

**def one(self):**

**print('this is function one')**

**def two(self):**

**print('this is function two')**

**a1=A()**

**a2=A()**

**a1.one()**

**a1.two()**

**a2.one()**

**a2.two()**

**Output:**

**this is function one**

**this is function two**

**this is function one**

**this is function two**

**class Student:**

**def read(self):**

**print('Student is reading')**

**def eat(self):**

**print('student is eating')**

**s1=Student()**

**s1.read()**

**s1.eat()**

**Output:**

**Student is reading**

**student is eating**

**class Cat:**

**def insert (self,name):**

**self.name=name**

**def eat(self):**

**print(self.name, ' is eating')**

**def sleep(self):**

**print(self.name, 'is sleeping')**

**c1=Cat()**

**c1.insert('Amy')**

**c1.eat()**

**c1.sleep()**

**c2.insert('Tamy')**

**c2.eat()**

**c2.sleep()**

**output:**

**Amy is eating**

**Amy is sleeping**

**Tamy is eating**

**Tamy is sleeping**

**class Student:**

**def insert(self,name,id,age):**

**self.name=name**

**self.id=id**

**self.age=age**

**def display(self):**

**print(self.name,self.id,self.age)**

**s1=Student()**

**s2=Student()**

**s1.insert("ram",1,12)**

**s2.insert("kumar",2,13)**

**s1.display()**

**s2.display()**

**class Student:**

**def \_\_init\_\_(self,name,id,age):**

**self.name=name**

**self.id=id**

**self.age=age**

**def display(self):**

**print(self.name,self.id,self.age)**

**s1=Student("ram",1,12)**

**s2=Student("kumar",2,13)**

**s1.display()**

**s2.display()**

**class Cat:**

**def \_\_init\_\_(self,name):**

**self.name=name**

**def eat(self):**

**print(self.name, ' is eating')**

**def sleep(self):**

**print(self.name, 'is sleeping')**

**c1=Cat('Amy')**

**c1.eat()**

**c1.sleep()**

**c2=Cat('Tamy')**

**c2.eat()**

**c2.sleep()**

**output:**

**Amy is eating**

**Amy is sleeping**

**Tamy is eating**

**Tamy is sleeping**

**Variables – Instance Variables, Class Variables**

**Instance variables: The instance variables are attributes attached to an instance of a class. We define instance variables in the constructor ( the \_\_init\_\_() method of a class).**

**Class Variables: A class variable is a variable that is declared inside of class, but outside of any instance method or \_\_init\_\_() method**

**Methods: Instance Method, Class Method, Static Method**

**Instance method: Used to access or modify the object state. If we use instance variables inside a method, such methods are called instance methods.**

**Class method: Used to access or modify the class state. In method implementation, if we use only class variables, then such type of methods we should declare as a class method.**

**Static method: It is a general utility method that performs a task in isolation. Inside this method, we don’t use instance or class variable because this static method doesn’t have access to the class attributes.**

**Example for Instance Variables**

**class Person:**

**def \_\_init\_\_(self, name, age, profession): # constructor**

**# data members (instance variables)**

**self.name = name**

**self.age = age**

**self.profession = profession**

**# Behavior (instance methods)**

**def show(self):**

**print('Name:', self.name, ' Age:', self.age, ' Profession:', self.profession)**

**# Behavior (instance methods)**

**def work(self):**

**print(self.name, 'working as a', self.profession)**

**# create object of a class**

**p1 = Person('Ram', 20, 'Software Engineer')**

**# call methods**

**p1.show()**

**p1.work()**

**Output:**

**Name: Ram Age: 20 Profession: Software Engineer**

**Ram working as a Software Engineer**

**Example - 2**

**class Student:**

**def \_\_init\_\_(self, name, rollno, branch): # constructor**

**# data members (instance variables)**

**self.name = name**

**self.rollno = rollno**

**self.branch = branch**

**# Behavior (instance methods)**

**def show(self):**

**print('Name:', self.name, ' rollno:', self.rollno, ' branch:', self.branch)**

**# Behavior (instance methods)**

**def work(self):**

**print(self.name, 'Studying in ', self.branch)**

**# create object of a class**

**s1 = Student('Ram', 2, 'ECE')**

**s2 = Student('Arun', 7, 'Mech')**

**# call methods**

**s1.show()**

**s1.work()**

**s2.show()**

**s2.work()**

**Output:**

**Name: Ram rollno: 2 branch: ECE**

**Ram Studying in ECE**

**Name: Arun rollno: 7 branch: Mech**

**Arun Studying in Mech**

**Example for Class Variables**

**class Student:**

**#class variable**

**city="vizag"**

**def \_\_init\_\_(self,id,name,age):**

**self.id=id**

**self.name=name**

**self.age=age**

**def display(self):**

**print(self.id,self.name,self.age,Student.city)**

**s1=Student(1,'ram',20)**

**s1=Student(2,'ramana',21)**

**s1.display()**

**Output**

**1 ram 20 vizag**

**2 ramana 21 vizag**

**Example for Instance Method**

**class Student:**

**# constructor**

**def \_\_init\_\_(self, name, age):**

**# Instance variable**

**self.name = name**

**self.age = age**

**# instance method access instance variable**

**def show(self):**

**print('Name:', self.name, 'Age:', self.age)**

**# create first object**

**print('First Student')**

**emma = Student("Jessa", 14)**

**# call instance method**

**emma.show()**

**# create second object**

**print('Second Student')**

**kelly = Student("Kelly", 16)**

**# call instance method**

**kelly.show()**

**Output:**

**First Student**

**Name: Jessa Age: 14**

**Second Student**

**Name: Kelly Age: 16**

**Example for Class Method**

**class Student:**

**city="Vizag"**

**@classmethod**

**def change(cls):**

**cls.city="Hyderabad"**

**def \_\_init\_\_(self,id, name):**

**self.id=id**

**self.name=name**

**def display(self):**

**print(self.id,self.name,Student.city)**

**a1=Student(1,"Ram")**

**a2=Student(2,"Shyam")**

**a1.display()**

**a2.display()**

**Student.change()**

**a1.display()**

**Example for Static Method**

**class Employee:**

**@staticmethod**

**def sample(x):**

**print('Inside static method', x)**

**# call static method**

**Employee.sample(10)**

**# can be called using object**

**emp = Employee()**

**emp.sample(10)**

**What is a Constructor?**

**Constructor: A constructor is a special method used to create and initialize an object of a**[**class**](https://pynative.com/python-classes-and-objects/)**. This method is defined in the class.**

**Types of Constructor**

**In Python, we have the following three types of constructors.**

* **Default Constructor**
* **Non-parametrized constructor**
* **Parameterized constructor**

**Default Constructor**

**Python will provide a default constructor if no constructor is defined. Python adds a default constructor when we do not include the constructor in the class or forget to declare it. It does not perform any task but initializes the objects. It is an empty constructor without a body**

**Non-Parametrized Constructor**

* **A constructor without any arguments is called a non-parameterized constructor. This type of constructor is used to initialize each object with default values.**
* **This constructor doesn’t accept the arguments during object creation. Instead, it initializes every object with the same set of values.**
* **A constructor without any arguments is called a non-parameterized constructor. This type of constructor is used to initialize each object with default values.**
* **This constructor doesn’t accept the arguments during object creation. Instead, it initializes every object with the same set of values.**

**Parameterized Constructor**

* **A constructor with defined parameters or arguments is called a parameterized constructor. We can pass different values to each object at the time of creation using a parameterized constructor.**
* **The first parameter to constructor is self that is a reference to the being constructed, and the rest of the arguments are provided by the programmer. A parameterized constructor can have any number of arguments.**

**Self Keyword in Python**

* **As you all know, the class contains instance variables and methods. Whenever we define instance methods for a class, we use self as the first parameter. Using self, we can access the**[**instance variable**](https://pynative.com/python-instance-variables/)**and instance method of the object.**
* **The first argument self refers to the current object.**
* **Whenever we call an instance method through an object, the Python compiler implicitly passes object reference as the first argument commonly known as self.**

**Docstring: It is the first string inside the class and has a brief description of the class. Although not mandatory, this is highly recommended.**

**Create a Class in Python**

**In Python, class is defined by using the class keyword. The syntax to create a class is given below.**

**Syntax**

**class class\_name:**

**'''This is a docstring. I have created a new class'''**

**<statement 1>**

**<statement 2>**

**.**

**.**

**<statement N>**

**class\_name: It is the name of the class**

**Docstring: It is the first string inside the class and has a brief description of the class. Although not mandatory, this is highly recommended.**

**statements: Attributes and methods**

**Inheritance in Python**

**The process of inheriting the properties of the parent class into a child class is called inheritance. The existing class is called a base class or parent class and the new class is called a subclass or child class or derived class.**

**Types Of Inheritance**

**In Python, based upon the number of child and parent classes involved, there are five types of inheritance. The type of inheritance are listed below:**

* **Single inheritance**
* **Multilevel inheritance**
* **Hierarchical Inheritance**
* **Multiple Inheritance**
* **Hybrid Inheritance**

**Single Inheritance**

**In single inheritance, a child class inherits from a single-parent class. Here is one child class and one parent class.**

**Multilevel inheritance**

**In multilevel inheritance, a class inherits from a child class or derived class. Suppose three classes A, B, C. A is the superclass, B is the child class of A, C is the child class of B. In other words, we can say a chain of classes is called multilevel inheritance.**

**Hierarchical Inheritance**

**In Hierarchical inheritance, more than one child class is derived from a single parent class. In other words, we can say one parent class and multiple child classes.**

**Multiple Inheritance**

**In multiple inheritance, one child class can inherit from multiple parent classes. So here is one child class and multiple parent classes.**

**Hybrid Inheritance**

**When inheritance is consists of multiple types or a combination of different inheritance is called hybrid inheritance.**

|  |  |  |
| --- | --- | --- |
| **Python Single Inheritance**  **Single Level Inheritance** | **Python Multilevel Inheritance**  **Multi Level Inheritance** | **Python hierarchical inheritance**  **Hierarchical Inheritance** |
| **Python Multiple Inheritance**  **Multiple Inheritance** | **Python hybrid inheritance**  **Hybrid Inheritance** |  |

**Example for Single Level Inheritance**

**class Vehicle:**

**def Vehicle\_info(self):**

**print('Inside Vehicle class')**

**class Car(Vehicle):**

**def car\_info(self):**

**print('Inside Car class')**

**car = Car()**

**car.Vehicle\_info()**

**car.car\_info()**

**Output**

**Inside Vehicle class**

**Inside Car class**

**Example for Multi Level Inheritance**

**class Vehicle:**

**def Vehicle\_info(self):**

**print('Inside Vehicle class')**

**class Car(Vehicle):**

**def car\_info(self):**

**print('Inside Car class')**

**class SportsCar(Car):**

**def sports\_car\_info(self):**

**print('Inside SportsCar class')**

**car1 = SportsCar()**

**car1.Vehicle\_info()**

**car1.car\_info()**

**car1.sports\_car\_info()**

**Output**

**Inside Vehicle class**

**Inside Car class**

**Inside SportsCar class**

**Example for Hierarchial Inheritance**

**class Vehicle:**

**def info(self):**

**print("This is Vehicle")**

**class Car(Vehicle):**

**def car\_info(self, name):**

**print("Car name is:", name)**

**class Truck(Vehicle):**

**def truck\_info(self, name):**

**print("Truck name is:", name)**

**obj1 = Car()**

**obj1.info()**

**obj1.car\_info('BMW')**

**obj2 = Truck()**

**obj2.info()**

**obj2.truck\_info('Ford')**

**Output**

**This is Vehicle**

**Car name is: BMW**

**This is Vehicle**

**Truck name is: Ford**

**Example for Multiple Inheritance**

**class Person:**

**def person\_info(self, name, age):**

**print('Inside Person class')**

**print('Name:', name, 'Age:', age)**

**class Company:**

**def company\_info(self, company\_name, location):**

**print('Inside Company class')**

**print('Name:', company\_name, 'location:', location)**

**class Employee(Person, Company):**

**def Employee\_info(self, salary, skill):**

**print('Inside Employee class')**

**print('Salary:', salary, 'Skill:', skill)**

**emp = Employee()**

**emp.person\_info('Jessa', 28)**

**emp.company\_info('Google', 'Atlanta')**

**emp.Employee\_info(12000, 'Machine Learning')**

**Output**

**Inside Person class**

**Name: Jessa Age: 28**

**Inside Company class**

**Name: Google location: Atlanta**

**Inside Employee class**

**Salary: 12000 Skill: Machine Learning**

**Example for Hybrid Inheritance**

**class Vehicle:**

**def vehicle\_info(self):**

**print("Inside Vehicle class")**

**class Car(Vehicle):**

**def car\_info(self):**

**print("Inside Car class")**

**class Truck(Vehicle):**

**def truck\_info(self):**

**print("Inside Truck class")**

**class SportsCar(Car, Vehicle):**

**def sports\_car\_info(self):**

**print("Inside SportsCar class")**

**s\_car = SportsCar()**

**s\_car.vehicle\_info()**

**s\_car.car\_info()**

**s\_car.sports\_car\_info()**

**Python super() function**

**When a class inherits all properties and behavior from the parent class is called inheritance. In such a case, the inherited class is a subclass and the latter class is the parent class.**

**In child class, we can refer to parent class by using the super() function. The super function returns a temporary object of the parent class that allows us to call a parent class method inside a child class method.**

**Benefits of using the super() function.**

* **We are not required to remember or specify the parent class name to access its methods.**
* **We can use the super() function in both single and multiple inheritances.**
* **The super() function support code reusability as there is no need to write the entire function**

**class Company:**

**def company\_name(self):**

**return 'Google'**

**class Employee(Company):**

**def info(self):**

**# Calling the superclass method using super()function**

**c\_name = super().company\_name()**

**print("Jessa works at", c\_name)**

**# Creating object of child class**

**emp = Employee()**

**emp.info()**

**Output:**

**Jessa works at Google**

**Polymorphism in Python**

**Polymorphism in Python is the ability of an**[**object**](https://pynative.com/python-classes-and-objects/)**to take many forms. In simple words, polymorphism allows us to perform the same action in many different ways.**

1. **Method overriding**
2. **Method overloading**
3. **Operator overloading**
4. **Method Overriding**

**Polymorphism allows us to defines methods in the child**[**class**](https://pynative.com/python-classes-and-objects/)**that have the same name as the methods in the parent class. This process of re-implementing the inherited method in the child class is known as Method Overriding.**

1. **Method Overloading**

**The process of calling the same method with different parameters is known as method overloading. Python does not support method overloading. Python considers only the latest defined method even if you overload the method. Python will raise a TypeError if you overload the method.**

1. **Operator Overloading in Python**

**Operator overloading means changing the default behavior of an**[**operator**](https://pynative.com/python-operators/)**depending on the operands (values) that we use. In other words, we can use the same operator for multiple purposes.**

**Example for Method Overriding - 1**

**class A:**

**def one(self):**

**print('this is one')**

**class B(A):**

**def one(self):**

**print('this is two')**

**b1=B()**

**b1.one()**

**Example for Method Overriding - 2**

**class Vehicle:**

**def \_\_init\_\_(self, name, color, price):**

**self.name = name**

**self.color = color**

**self.price = price**

**def show(self):**

**print('Details:', self.name, self.color, self.price)**

**def max\_speed(self):**

**print('Vehicle max speed is 150')**

**def change\_gear(self):**

**print('Vehicle change 6 gear')**

**class Car(Vehicle):**

**def max\_speed(self):**

**print('Car max speed is 240')**

**def change\_gear(self):**

**print('Car change 7 gear')**

**car = Car('Car x1', 'Red', 20000)**

**car.show()**

**car.max\_speed()**

**car.change\_gear()**

**vehicle = Vehicle('Truck x1', 'white', 75000)**

**vehicle.show()**

**vehicle.max\_speed()**

**vehicle.change\_gear()**

**Output:**

**Details: Car x1 Red 20000**

**Car max speed is 240**

**Car change 7 gear**

**Details: Truck x1 white 75000**

**Vehicle max speed is 150**

**Vehicle change 6 gear**

**Example for Method Overloading**

**def addition(a, b):**

**c = a + b**

**print(c)**

**def addition(a, b, c):**

**d = a + b + c**

**print(d)**

**# addition(4, 5)**

**# This line will call the second product method**

**addition(3, 7, 5)**

**def adder(\*num):**

**sum = 0**

**for n in num:**

**sum = sum + n**

**print("Sum:",sum)**

**adder(3,5)**

**adder(4,5,6,7)**

**adder(1,2,3,5,6)**

**Example for Method Overloading**

**class Shape:**

**def area(self, a, b=0):**

**if b > 0:**

**print('Area of Rectangle is:', a \* b)**

**else:**

**print('Area of Square is:', a \*\* 2)**

**square = Shape()**

**square.area(5)**

**rectangle = Shape()**

**rectangle.area(5, 3)**

**Output:**

**Area of Square is: 25**

**Area of Rectangle is: 15**

**Overloading + operator for custom objects**

**Suppose we have two objects, and we want to add these two objects with a binary + operator. However, it will throw an error if we perform addition because the compiler doesn’t add two objects. See the following example for more details.**

**Example for Operator Overloading which is not possible**

**class Book:**

**def \_\_init\_\_(self, pages):**

**self.pages = pages**

**# creating two objects**

**b1 = Book(400)**

**b2 = Book(300)**

**# add two objects**

**print(b1 + b2)**

**Output**

**TypeError: unsupported operand type(s) for +: 'Book' and 'Book'**

**We can overload + operator to work with custom objects also. Python provides some special or magic function that is automatically invoked when associated with that particular operator.**

**For example, when we use the + operator, the magic method \_\_add\_\_() is automatically invoked. Internally + operator is implemented by using \_\_add\_\_() method. We have to override this method in our class if you want to add two custom objects.**

**Example for Operator Overloading - 1**

**class Book:**

**def \_\_init\_\_(self, pages):**

**self.pages = pages**

**# Overloading + operator with magic method**

**def \_\_add\_\_(self, b2):**

**return self.pages + b2.pages**

**b1 = Book(400)**

**b2 = Book(300)**

**print("Total number of pages: ", b1 + b2)**

**Output**

**Total number of pages: 700**

**Overloading the \* Operator**

**Example for Operator Overloading - 2**

**class Employee:**

**def \_\_init\_\_(self, name, salary):**

**self.name = name**

**self.salary = salary**

**def \_\_mul\_\_(self, timesheet):**

**print('Worked for', timesheet.days, 'days')**

**return self.salary \* timesheet.days**

**class TimeSheet:**

**def \_\_init\_\_(self, name, days):**

**self.name = name**

**self.days = days**

**emp = Employee("Jessa", 800)**

**timesheet = TimeSheet("Jessa", 50)**

**print("salary is: ", emp \* timesheet)**

**Output**

**Wroked for 50 days**

**salary is: 40000**

**Magic Methods**

**In Python, there are different magic methods available to perform overloading operations. The below table shows the magic methods names to overload the mathematical operator, assignment operator, and relational operators in Python.**

| **Operator Name** | **Symbol** | **Magic method** |
| --- | --- | --- |
| **Addition** | **+** | **\_\_add\_\_(self, other)** |
| **Subtraction** | **-** | **\_\_sub\_\_(self, other)** |
| **Multiplication** | **\*** | **\_\_mul\_\_(self, other)** |
| **Division** | **/** | **\_\_div\_\_(self, other)** |
| **Floor Division** | **//** | **\_\_floordiv\_\_(self,other)** |
| **Modulus** | **%** | **\_\_mod\_\_(self, other)** |
| **Power** | **\*\*** | **\_\_pow\_\_(self, other)** |
| **Increment** | **+=** | **\_\_iadd\_\_(self, other)** |
| **Decrement** | **-=** | **\_\_isub\_\_(self, other)** |
| **Product** | **\*=** | **\_\_imul\_\_(self, other)** |
| **Division** | **/+** | **\_\_idiv\_\_(self, other)** |
| **Modulus** | **%=** | **\_\_imod\_\_(self, other)** |
| **Power** | **\*\*=** | **\_\_ipow\_\_(self, other)** |
| **Less than** | **<** | **\_\_lt\_\_(self, other)** |
| **Greater than** | **>** | **\_\_gt\_\_(self, other)** |
| **Less than or equal to** | **<=** | **\_\_le\_\_(self, other)** |
| **Greater than or equal to** | **>=** | **\_\_ge\_\_(self, other)** |
| **Equal to** | **==** | **\_\_eq\_\_(self, other)** |
| **Not equal** | **!=** | **\_\_ne\_\_(self, other)** |

**ABSTRACTION**

**What is Abstraction?**

**Hiding internal details and showing only functionality.**

**Abstract Classes in Python**

* **An abstract class can be considered as a blueprint for other classes, allows you to create a set of methods that must be created within any child classes built from your abstract class.**
* **A class which contains one or more abstract methods is called an abstract class. An abstract method is a method that has declaration but not has any implementation.**
* **Abstract classes are not able to instantiated and it needs subclasses to provide implementations for those abstract methods which are defined in abstract classes. While we are designing large functional units we use an abstract class.**
* **When we want to provide a common implemented functionality for all implementations of a component, we use an abstract class.**
* **Abstract classes allow partially to implement classes when it completely implements all methods in a class, then it is called interface.**

**class A():**

**def one(self):**

**pass**

**class B(A):**

**def one(self):**

**print('this is one')**

**a1=A()**

**a1.one()**

**b1=B()**

**b1.one()**

**from abc import ABC, abstractmethod**

**class A(ABC):**

**@abstractmethod**

**def one(self):**

**pass**

**class B(A):**

**def one(self):**

**print('this is one')**

**a1=A()**

**a1.one()**

**b1=B()**

**b1.one()**

|  |
| --- |
| **from abc import ABC, abstractmethod**    **class Polygon(ABC):**    **@abstractmethod**  **def noofsides(self):**  **pass**    **class Triangle(Polygon):**    **def noofsides(self):**  **print("I have 3 sides")**    **class Pentagon(Polygon):**    **def noofsides(self):**  **print("I have 5 sides")**    **class Hexagon(Polygon):**    **def noofsides(self):**  **print("I have 6 sides")**    **class Quadrilateral(Polygon):**    **def noofsides(self):**  **print("I have 4 sides")**    **R = Triangle()**  **R.noofsides()**    **K = Quadrilateral()**  **K.noofsides()**    **R = Pentagon()**  **R.noofsides()**    **K = Hexagon()**  **K.noofsides()**  **Output:**  **I have 3 sides**  **I have 4 sides**  **I have 5 sides**  **I have 6 sides** |

|  |
| --- |
| **from abc import ABC, abstractmethod**  **class Animal(ABC):**  **@abstractmethod**  **def move(self):**  **pass**    **class Human(Animal):**    **def move(self):**  **print("I can walk and run")**    **class Snake(Animal):**    **def move(self):**  **print("I can crawl")**    **class Dog(Animal):**    **def move(self):**  **print("I can bark")**    **class Lion(Animal):**    **def move(self):**  **print("I can roar")**    **# Driver code**  **R = Human()**  **R.move()**    **K = Snake()**  **K.move()**    **R = Dog()**  **R.move()**    **K = Lion()**  **K.move()**  **Output:**  **I can walk and run**  **I can crawl**  **I can bark**  **I can roar** |

**Encapsulation in Python**

## What is Encapsulation in Python?

**Encapsulation in Python describes the concept of bundling data and methods within a single unit. So, for example, when you create a class, it means you are implementing encapsulation. A class is an example of encapsulation as it binds all the data members (instance variables) and methods into a single unit.**

## Example:

**class Employee:**

**def \_\_init\_\_(self, name, salary, project):**

**self.name = name**

**self.salary = salary**

**self.project = project**

**def show(self):**

**print("Name: ", self.name, 'Salary:', self.salary)**

**def work(self):**

**print(self.name, 'is working on', self.project)**

**emp = Employee('Jessa', 8000, 'NLP')**

**emp.show()**

**emp.work()**

**Output:**

**Name: Jessa Salary: 8000**

**Jessa is working on NLP**

## Access Modifiers in Python

**Encapsulation can be achieved by declaring the data members and methods of a class either as private or protected. But In Python, we don’t have direct access modifiers like public, private, and protected. We can achieve this by using single underscore and double underscores.**

**Access modifiers limit access to the variables and methods of a class. Python provides three types of access modifiers private, public, and protected.**

* **Public Member: Accessible anywhere from outside of class.**
* **Private Member: Accessible within the class**
* **Protected Member: Accessible within the class and its sub-classes**

## Public Member

**Public data members are accessible within and outside of a class. All member variables of the class are by default public.**

## Example for Public Member

**class Employee:**

**# constructor**

**def \_\_init\_\_(self, name, salary):**

**# public data members**

**self.name = name**

**self.salary = salary**

**# public instance methods**

**def show(self):**

**# accessing public data member**

**print("Name: ", self.name, 'Salary:', self.salary)**

**# creating object of a class**

**emp = Employee('Jessa', 10000)**

**# accessing public data members**

**print("Name: ", emp.name, 'Salary:', emp.salary)**

**# calling public method of the class**

**emp.show()**

[**Run**](https://pynative.com/online-python-code-editor-to-execute-python-code/)

**Output**

**Name: Jessa Salary: 10000**

**Name: Jessa Salary: 10000**

## Private Member

**We can protect variables in the class by marking them private. To define a private variable add two underscores as a prefix at the start of a variable name.**

**Private members are accessible only within the class, and we can’t access them directly from the class objects.**

## Example for Private Member

**class A:**

**def \_\_init\_\_(self,x):**

**self.\_\_x =x**

**def show(self):**

**print(self.\_\_x)**

**a1=A(10)**

**print(a1.\_\_x)**

**a1.show()**

**output:**

**print(a1.\_\_x)**

**AttributeError: 'A' object has no attribute '\_\_x'**

**In the above example, the salary is a private variable. As you know, we can’t access the private variable from the outside of that class.**

**We can access private members from outside of a class using the following two approaches**

* **Create public method to access private members**
* **Use name mangling**

**Access Private member outside of a class using an instance method**

**class A:**

**def \_\_init\_\_(self,salary):**

**self.\_\_salary=salary**

**def show(self):**

**print(self.\_\_salary)**

**a1=A(10000)**

**a1.show()**

**output:**

**10000**

## Name Mangling to access private members

**We can directly access private and protected variables from outside of a class through name mangling. The name mangling is created on an identifier by adding two leading underscores and one trailing underscore, like this \_classname\_\_dataMember, where classname is the current class, and data member is the private variable name.**

## Example: Access private member

**class A:**

**def \_\_init\_\_(self, name, salary):**

**self.name = name**

**self.\_\_salary = salary**

**a1 = A('Ram', 10000)**

**print('Name:', a1.name)**

**print('Salary:', a1.\_A\_\_salary)**

**Output**

**Name: Ram**

**Salary: 10000**

## Protected Member

**Protected members are accessible within the class and also available to its sub-classes. To define a protected member, prefix the member name with a single underscore \_.**

**Protected data members are used when you implement inheritance and want to allow data members access to only child classes.**

## Example for Protected member

**class A:**

**def \_\_init\_\_(self,salary):**

**self.\_salary=salary**

**class B(A):**

**def display(self):**

**print(self.\_salary)**

**b1=B(20000)**

**b1.display()**

**Output:**

**20000**

## Getters and Setters in Python

**To implement proper encapsulation in Python, we need to use setters and getters. The primary purpose of using getters and setters in object-oriented programs is to ensure data encapsulation. Use the getter method to access data members and the setter methods to modify the data members.**

**In Python, private variables are not hidden fields like in other programming languages. The getters and setters methods are often used when:**

* **When we want to avoid direct access to private variables**
* **To add validation logic for setting a value**

## Example

**class Student:**

**def \_\_init\_\_(self, name, age):**

**# private member**

**self.name = name**

**self.\_\_age = age**

**# getter method**

**def get\_age(self):**

**return self.\_\_age**

**# setter method**

**def set\_age(self, age):**

**self.\_\_age = age**

**stud = Student('Jessa', 14)**

**# retrieving age using getter**

**print('Name:', stud.name, stud.get\_age())**

**# changing age using setter**

**stud.set\_age(16)**

**# retrieving age using getter**

**print('Name:', stud.name, stud.get\_age())**

**Output**

**Name: Jessa 14**

**Name: Jessa 16**

**Python Exception**

* **An exception is a Python object that represents an error. Python provides a way to handle the exception so that the code can be executed without any interruption.**
* **An exception can be defined as an unusual condition in a program resulting in the interruption in the flow of the program.**
* **Whenever an exception occurs, the program stops the execution, and thus the further code is not executed. Therefore, an exception is the run-time errors that are unable to handle to Python script. An exception is a Python object that represents an error**
* **Python provides a way to handle the exception so that the code can be executed without any interruption. If we do not handle the exception, the interpreter doesn't execute all the code that exists after the exception.**

**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.**

**Without try except**

**a=int(input('Enter a value'))**

**b=int(input('Enter b value'))**

**c=a/b**

**print(c)**

**print('normal execution')**

**print('normal execution')**

**output:**

**Enter a value: 10**

**Enter b value: 0**

**Traceback (most recent call last):**

**File "main.py", line 3, in <module>**

**c=a/b**

**ZeroDivisionError: division by zero**

**With try catch**

**a=int(input('enter a value'))**

**b=int(input('enter b value'))**

**try:**

**c=a/b**

**print(c)**

**except:**

**print('not divisible by 0')**

**print('normal execution')**

**print('normal execution')**

**output:**

**enter a value10**

**enter b value0**

**not divisibly by 0**

**normal execution**

**normal execution**

**Name error**

**try:**

**print(a)**

**except NameError:**

**print("Name error")**

**print('hi')**

**print("hello")**

**Type error**

**a='india'**

**b=20**

**try:**

**c=a+b**

**print(c)**

**except:**

**print("type error")**

**Try Except Else**

**try:**

**a=10/0**

**print (a)**

**except ArithmeticError:**

**print( "This statement is raising an exception")**

**else:**

**print ("Welcome" )**

**Output:**

**This statement is raising an exception**

## Else

**You can use the else keyword to define a block of code to be executed if no errors were raised:**

**try:**

**print("Hello")**

**except:**

**print("Something went wrong")**

**else:**

**print("Nothing went wrong")**

**Hello  
Nothing went wrong**

**Finally Block:**

**In case if there is any code which the user want to be executed, whether exception occurs or not then that code can be placed inside the finally block. Finally block will always be executed irrespective of the exception.**

**try:**

**a=10/0**

**print ("Exception not occurred")**

**except ArithmeticError:**

**print( "This statement is raising an exception")**

**finally:**

**print ("Code to be executed")**

**Output:**

**Code to be executed**

**Traceback (most recent call last):**

**File "C:/Python27/noexception.py", line 2, in <module>**

**a=10/0;**

**ZeroDivisionError: integer division or modulo by zero**

## 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")**

**output:**

**Variable x is not defined**

**x=[1,2,3,4]**

**try:**

**print(x[7])**

**except NameError:**

**print("Variable x is not defined")**

**except:**

**print("Something else went wrong")**

**Raising Exceptions in Python**

**In Python programming, exceptions are raised when errors occur at runtime. We can also manually raise exceptions using the raise keyword.**

**It causes an exception to be generated explicitly. Built-in errors are raised implicitly.**

**try:**

**x=int(input('Enter a number upto 100: '))**

**if x > 100:**

**raise ValueError(x)**

**except ValueError:**

**print(x, "is out of allowed range")**

**else:**

**print(x, "is within the allowed range")**

**Output**

**Enter a number upto 100: 200**

**200 is out of allowed range**

**Enter a number upto 100: 50**

**50 is within the allowed range**

## Custom and User-defined Exceptions

**Sometimes we have to define and raise exceptions explicitly to indicate that something goes wrong. Such a type of exception is called a user-defined exception or customized exception.**

**The user can define custom exceptions by creating a new class. This new exception class has to derive either directly or indirectly from the built-in class Exception. In Python, most of the built-in exceptions also derived from the Exception class.**

**class Error(Exception):**

**pass**

**class ValueTooSmallError(Error):**

**pass**

**class ValueTooLargeError(Error):**

**pass**

**try:**

**num = int(input("Enter any value in 10 to 50 range: "))**

**if num < 10:**

**raise ValueTooSmallError**

**elif num > 50:**

**raise ValueTooLargeError**

**except ValueTooSmallError:**

**print("Value is below range..try again")**

**except ValueTooLargeError:**

**print("value out of range...try again")**

**print("Great! value in correct range.")**

**Built-in Exceptions**

**The table below shows built-in exceptions that are usually raised in Python:**

|  |  |
| --- | --- |
| **Exception** | **Description** |
| **ArithmeticError** | **Raised when an error occurs in numeric calculations** |
| **AssertionError** | **Raised when an assert statement fails** |
| **AttributeError** | **Raised when attribute reference or assignment fails** |
| **Exception** | **Base class for all exceptions** |
| **EOFError** | **Raised when the input() method hits an "end of file" condition (EOF)** |
| **FloatingPointError** | **Raised when a floating point calculation fails** |
| **GeneratorExit** | **Raised when a generator is closed (with the close() method)** |
| **ImportError** | **Raised when an imported module does not exist** |
| **IndentationError** | **Raised when indendation is not correct** |
| **IndexError** | **Raised when an index of a sequence does not exist** |
| **KeyError** | **Raised when a key does not exist in a dictionary** |
| **KeyboardInterrupt** | **Raised when the user presses Ctrl+c, Ctrl+z or Delete** |
| **LookupError** | **Raised when errors raised cant be found** |
| **MemoryError** | **Raised when a program runs out of memory** |
| **NameError** | **Raised when a variable does not exist** |
| **NotImplementedError** | **Raised when an abstract method requires an inherited class to override the method** |
| **OSError** | **Raised when a system related operation causes an error** |
| **OverflowError** | **Raised when the result of a numeric calculation is too large** |
| **ReferenceError** | **Raised when a weak reference object does not exist** |
| **RuntimeError** | **Raised when an error occurs that do not belong to any specific expections** |
| **StopIteration** | **Raised when the next() method of an iterator has no further values** |
| **SyntaxError** | **Raised when a syntax error occurs** |
| **TabError** | **Raised when indentation consists of tabs or spaces** |
| **SystemError** | **Raised when a system error occurs** |
| **SystemExit** | **Raised when the sys.exit() function is called** |
| **TypeError** | **Raised when two different types are combined** |
| **UnboundLocalError** | **Raised when a local variable is referenced before assignment** |
| **UnicodeError** | **Raised when a unicode problem occurs** |
| **UnicodeEncodeError** | **Raised when a unicode encoding problem occurs** |
| **UnicodeDecodeError** | **Raised when a unicode decoding problem occurs** |
| **UnicodeTranslateError** | **Raised when a unicode translation problem occurs** |
| **ValueError** | **Raised when there is a wrong value in a specified data type** |
| **ZeroDivisionError** | **Raised when the second operator in a division is zero** |

**Multi threading**

**Multithreading is defined as the ability of a processor to execute multiple threads concurrently.**

## Why use Multithreading?

**Multithreading allows you to break down an application into multiple sub-tasks and run these tasks simultaneously. If you use multithreading properly, your application speed, performance, and rendering can all be improved.**

**class Hello:**

**def one(self):**

**for i in range(5):**

**print('Hello', end=' ')**

**class Hi:**

**def two(self):**

**for i in range(5):**

**print('hai', end=' ')**

**t1=Hello()**

**t2=Hi()**

**t1.one()**

**t2.two()**

**output:**

**Hello Hello Hello Hello Hello hai hai hai hai hai**

**Multi Threading example**

**from threading import \***

**class Hello(Thread):**

**def run(self):**

**for i in range(5):**

**print('Hello', end=' ')**

**class Hi(Thread):**

**def run(self):**

**for i in range(5):**

**print('hai', end=' ')**

**t1=Hello()**

**t2=Hi()**

**t1.start()**

**t2.start()**

**output:**

**Hello haiHello Hello Hello Hello hai hai hai hai**

**Thread Synchronization**

**Many Threads trying to access the same object can lead to problem like making data inconsistent or getting unexpected output. So when a thread is already accessing an object, preventing any other thread accessing the same object is called Thread synchronization.**

**The object on which the threads are synchronized is called Synchronized Object or Mutually Exclusive Lock(mutex).**

**Thread Synchronization is recommended when multiple threads are acting on the same thread simultaneously.**

**Following techniques to do Thread Synchronization**

1. **Using locks**
2. **Using RLock**

**Locks**

**Locks are typically used to synchronize access to a shared resource. Lock can be used to lock the object in which the thread is acting. A Lock has only two states, locked and unlocked. It is created in the unlocked state.**

**Acquire( )**

**This method is used to change the state to locked and return immediately. When the state is locked, acquire() blocks until a call to release() in another thread changes it to unlocked, the acquire() call resets it to locked and returns.**

**Release()**

**This method is used to release a lock(). This can be called from any thread, not only the thread which has acquired the lock.**

**from threading import Thread, current\_thread**

**class Flight:**

**def \_\_init\_\_(self, available\_seat):**

**self.available\_seat=available\_seat**

**def reserve(self,need\_seat):**

**print('Available seats: ', self.available\_seat)**

**if(self.available\_seat>=need\_seat):**

**name=current\_thread().name**

**print(f'{need\_seat} seat is alloted for {name}')**

**self.available\_seat-=need\_seat**

**else:**

**print('Sorry! All seats has alloted')**

**f=Flight(1)**

**t1=Thread(target=f.reserve, args=(1,), name='Rahul')**

**t2=Thread(target=f.reserve, args=(1,), name='Sonam')**

**t3=Thread(target=f.reserve, args=(1,), name='Vikram')**

**t1.start()**

**t2.start()**

**t3.start()**

**from threading import \***

**class Flight:**

**def \_\_init\_\_(self, available\_seat):**

**self.available\_seat=available\_seat**

**self.l = Lock()**

**def reserve(self,need\_seat):**

**self.l.acquire()**

**print('Available seats: ', self.available\_seat)**

**if(self.available\_seat>=need\_seat):**

**name=current\_thread().name**

**print(f'{need\_seat} seat is alloted for {name}')**

**self.available\_seat-=need\_seat**

**else:**

**print('Sorry! All seats has alloted')**

**self.l.release()**

**f=Flight(1)**

**t1=Thread(target=f.reserve, args=(1,), name='Rahul')**

**t2=Thread(target=f.reserve, args=(1,), name='Sonam')**

**t3=Thread(target=f.reserve, args=(1,), name='Vikram')**

**t1.start()**

**t2.start()**

**t3.start()**

**FILES**

**Files are named locations on disk to store related information. They are used to permanently store data in a non-volatile memory (e.g. hard disk).**

**File operation takes place in the following order:**

* **Open a file**
* **Read or write (perform operation)**
* **Close the file**

1. **Opening Files in Python**

**Python has a built-in open() function to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.**

**f = open("test.txt") # open file in current directory**

**f = open("C:/Python38/README.txt") # specifying full path**

**We can also specify if we want to open the file in text mode or binary mode.**

* **The default is reading in text mode. In this mode, we get strings when reading from the file.**
* **On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like images or executable files.**

**f = open("test.txt") # equivalent to 'r' or 'rt'**

**f = open("test.txt",'w') # write in text mode**

**f = open("img.bmp",'r+b') # read and write in binary mode**

|  |  |
| --- | --- |
| Mode | Description |
| r | **Opens a file for reading. (default)** |
| w | **Opens a file for writing. Creates a new file if it does not exist or truncates the file if it exists.** |
| x | **Opens a file for exclusive creation. If the file already exists, the operation fails.** |
| a | **Opens a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.** |
| t | **Opens in text mode. (default)** |
| b | **Opens in binary mode.** |
| + | **Opens a file for updating (reading and writing)** |

1. **Closing Files in Python**

**When we are done with performing operations on the file, we need to properly close the file.**

**Closing a file will free up the resources that were tied with the file. It is done using the close() method available in Python.**

**Python has a garbage collector to clean up unreferenced objects but we must not rely on it to close the file.**

**f = open("test.txt")**

**# perform file operations**

**f.close()**

1. **Writing to Files in Python**

**In order to write into a file in Python, we need to open it in write w, append a or exclusive creation x mode.**

**We need to be careful with the w mode, as it will overwrite into the file if it already exists. Due to this, all the previous data are erased.**

**Writing a string or sequence of bytes (for binary files) is done using the write() method. This method returns the number of characters written to the file.**

**with open("test.txt",'w') as f:**

**f.write("my first file\n")**

**f.write("This file\n\n")**

**f.write("contains three lines\n")**

**This program will create a new file named test.txt in the current directory if it does not exist. If it does exist, it is overwritten.**

**We must include the newline characters ourselves to distinguish the different lines.**

1. **Reading Files in Python**

**To read a file in Python, we must open the file in reading r mode.**

**There are various methods available for this purpose. We can use the read(size) method to read in the size number of data. If the size parameter is not specified, it reads and returns up to the end of the file.**

**We can read the text.txt file we wrote in the above section in the following way:**

**f = open("test.txt",'r')**

**f.read(4) # read the first 4 data**

**'This'**

**f.read(4) # read the next 4 data**

**' is '**

**f.read() # read in the rest till end of file**

**'my first file\nThis file\ncontains three lines\n'**

**We can change our current file cursor (position) using the seek() method. Similarly, the tell() method returns our current position (in number of bytes).**

**f.tell() # get the current file position**

**56**

**f.seek(0) # bring file cursor to initial position**

**0**

**print(f.read()) # read the entire file**

**Alternatively, we can use the readline() method to read individual lines of a file. This method reads a file till the newline, including the newline character.**

**f.readline()**

**'This is my first file\n'**

**f=open("sample.txt",'w')**

**f.write("hello world ")**

**f=open("sample.txt",'a')**

**f.write("adding new text ")**

**f.close()**

**f=open("sample.txt",'r')**

**print(f.read())**

**Decorators**

* **Decorators allow you to make simple modifications to callable objects like**[**functions**](https://www.learnpython.org/en/Functions)**,**[**methods, or classes**](https://www.learnpython.org/en/Classes%20and%20Objects)**.**

#### A decorator takes in a function, adds some functionality and returns it.

**Creating Decorators**

**let's go ahead and create a simple decorator that will convert a sentence to uppercase. We do this by defining a wrapper inside an enclosed function. As you can see it very similar to the function inside another function that we created earlier.**

**def a():**

**return 'hello there'**

**def uppercase(a):**

**def wrapper():**

**make\_uppercase = a().upper()**

**return make\_uppercase**

**return wrapper**

**decorate = uppercase(a)**

**print(decorate())**

**output:**

**HELLO THERE**

**However, Python provides a much easier way for us to apply decorators. We simply use the @ symbol before the function we'd like to decorate.**

**@uppercase\_decorator**

**def say\_hi():**

**return 'hello there'**

**say\_hi()**

**Output:**

**'HELLO THERE'**

**Applying Multiple Decorators to a Single Function**

**We can use multiple decorators to a single function. However, the decorators will be applied in the order that we've called them. Below we'll define another decorator that splits the sentence into a list. We'll then apply the uppercase\_decorator and split\_string decorator to a single function.**

**def uppercase\_decorator(function):**

**def wrapper():**

**func = function()**

**make\_uppercase = func.title()**

**return make\_uppercase**

**return wrapper**

**def split\_string(function):**

**def wrapper():**

**func = function()**

**splitted\_string = func.split()**

**return splitted\_string**

**return wrapper**

**@uppercase\_decorator**

**def say\_hi():**

**return 'hi there'**

**@split\_string**

**@uppercase\_decorator**

**def say\_hello():**

**return 'hello there'**

**print(say\_hi())**

**print(say\_hello())**

**output:**

**Hi There**

**['Hello', 'There']**

**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\_\_().**

**my\_list = [4, 7, 0, 3] # get an iterator using iter()**

**my\_iter = iter(my\_list) # iterate through it using next()**

**print(next(my\_iter)) # Output: 4**

**print(next(my\_iter)) # Output: 7**

**# next(obj) is same as obj.\_\_next\_\_()**

**print(my\_iter.\_\_next\_\_()) # Output: 0**

**print(my\_iter.\_\_next\_\_()) # Output: 3**

**# This will raise error, no items left**

**next(my\_iter) # error**

**Example**

**mystr = "banana"**

**myit = iter(mystr)**

**print(next(myit))**

**print(next(myit))**

**print(next(myit))**

**print(next(myit))**

**print(next(myit))**

**print(next(myit))**

**Output:**

**b**

**a**

**n**

**a**

**n**

**a**

**Building Custom Iterators**

**class PowTwo:**

**def \_\_init\_\_(self, max=0):**

**self.max = max**

**def \_\_iter\_\_(self):**

**self.n = 0**

**return self**

**def \_\_next\_\_(self):**

**if self.n <= self.max:**

**result = 2 \*\* self.n**

**self.n += 1**

**return result**

**else:**

**raise StopIteration**

**numbers = PowTwo(3)**

**i = iter(numbers)**

**print(next(i))**

**print(next(i))**

**print(next(i))**

**print(next(i))**

**print(next(i))**

**Output**

**1**

**2**

**4**

**8**

**Traceback (most recent call last):**

**File "/home/bsoyuj/Desktop/Untitled-1.py", line 32, in <module>**

**print(next(i))**

**File "<string>", line 18, in \_\_next\_\_**

**raise StopIteration**

**StopIteration**

**Generators**

**Generator function is a function which returns generator-iterator with the help of Yield Keyword.**

**def fib(mymax):**

**a,b=0,1**

**while true:**

**c=a+b**

**if(c< mymax) :**

**yield c**

**a=b**

**b=c**

**else:**

**break**

**gen=mymax(10)**

**print(next(gen))**

**print(next(gen))**

**print(next(gen))**

**print(next(gen))**

**print(next(gen))**

**print(next(gen)) # stop iteration**

## Numpy – Numerical Python

## NumPy can be used to perform a wide variety of mathematical operations on arrays. It adds powerful data structures to Python that guarantee efficient calculations with arrays and matrices

**a=[1,2,3,4]**

**b=[5,6,7,8]**

**c=a+b**

**print(c)**

**output:**

**[1, 2, 3, 4, 5, 6, 7, 8]**

**import numpy as np**

**a=[1,2,3,4]**

**b=[5,6,7,8]**

**a=np.array(a)**

**b=np.array(b)**

**c=a+b**

**print(c)**

**output:**

**[ 6 8 10 12]**

## Creating a numpy array

**import numpy as np**

**list1 = [0,1,2,3,4]**

**arr1d = np.array(list1)**

**print(arr1d)**

**output:**

**[0 1 2 3 4]**

**Create a 2d array from a list of lists**

**import numpy as np**

**list2 = [[0,1,2], [3,4,5], [6,7,8]]**

**arr2d = np.array(list2)**

**print(arr2d)**

**Output:**

**[[0 1 2]**

**[3 4 5]**

**[6 7 8]]**

**Create a float 2d array**

**import numpy as np**

**list2 = [[0,1,2], [3,4,5], [6,7,8]]**

**arr2d\_f = np.array(list2, dtype='float')**

**print(arr2d\_f)**

**[[0. 1. 2.]**

**[3. 4. 5.]**

**[6. 7. 8.]]**

**Create an object array to hold numbers as well as strings**

**import numpy as np**

**arr1d\_obj = np.array([1, 'a'], dtype='object')**

**print(arr1d\_obj)**

**output:**

**[1 'a']**

**# Create a boolean array**

**import numpy as np**

**arr2d\_b = np.array([1, 0, 10], dtype='bool')**

**print(arr2d\_b)**

**output:**

**[ True False True]**

**# Create a 2d array with 3 rows and 4 columns**

**import numpy as np**

**list2 = [[1, 2, 3, 4],[3, 4, 5, 6], [5, 6, 7, 8]]**

**arr2 = np.array(list2, dtype='float')**

**print(arr2)**

**Output:**

**[[1. 2. 3. 4.]**

**[3. 4. 5. 6.]**

**[5. 6. 7. 8.]]**

**import numpy as np**

**list2 = [[1, 2, 3, 4],[3, 4, 5, 6], [5, 6, 7, 8]]**

**arr2 = np.array(list2, dtype='float')**

**print('Shape: ', arr2.shape) # Shape: (3, 4)**

**print('Datatype: ', arr2.dtype) # Datatype: float64**

**print('Size: ', arr2.size) # Size: 12**

**print('Num Dimensions: ', arr2.ndim) #> Num Dimensions: 2**

**Extract the first 2 rows and columns**

**import numpy as np**

**list1 = [[1, 2, 3, 4],[3, 4, 5, 6], [5, 6, 7, 8]]**

**arr1d = np.array(list1)**

**print(arr1d[:2, :2])**

**output:**

**[[1 2]**

**[3 4]]**

**Get the boolean output by applying the condition to each element.**

**import numpy as np**

**list1 = [[1, 2, 3, 4],[3, 4, 5, 6], [5, 6, 7, 8]]**

**arr2d = np.array(list1)**

**b = arr2d > 4**

**print(b)**

**Output:**

**[[False False False False]**

**[False False True True]**

**[ True True True True]]**

**Reverse only the row positions**

**import numpy as np**

**list1 = [[1, 2, 3, 4],[3, 4, 5, 6], [5, 6, 7, 8]]**

**arr2d = np.array(list1)**

**print(arr2d[::-1, ])**

**output:**

**[[5 6 7 8]**

**[3 4 5 6]**

**[1 2 3 4]]**

**Reverse the row and column positions**

**import numpy as np**

**list1 = [[1, 2, 3, 4],[3, 4, 5, 6], [5, 6, 7, 8]]**

**arr2d = np.array(list1)**

**print(arr2d[::-1, ::-1])**

**output:**

**[[8 7 6 5]**

**[6 5 4 3]**

**[4 3 2 1]]**

**mean, max and min**

**import numpy as np**

**list1 = [[1, 2, 3, 4],[3, 4, 5, 6], [5, 6, 7, 8]]**

**arr2d = np.array(list1)**

**print("Mean value is: ", arr2d.mean())**

**print("Max value is: ", arr2d.max())**

**print("Min value is: ", arr2d.min())**

**output:**

**Mean value is: 4.5**

**Max value is: 8**

**Min value is: 1**

## Creating sequences, repetitions and random numbers using numpy?

**The np**.**arange function comes handy to create customised number sequences as ndarray.**

**# Lower limit is 0 be default**

**import numpy as np**

**print(np.arange(5)) # [0 1 2 3 4]**

**# 0 to 9**

**import numpy as np**

**print(np.arange(0, 10)) # [0 1 2 3 4 5 6 7 8 9]**

**# 0 to 9 with step of 2**

**import numpy as np**

**print(np.arange(0, 10, 2)) #[0 2 4 6 8]**

**# 10 to 1, decreasing order**

**import numpy as np**

**print(np.arange(10, 0, -1)) #[10 9 8 7 6 5 4 3 2 1]**

**#np**.**linspace**

**import numpy as np**

**print(np.linspace(start=1, stop=50, num=10, dtype=int))**

**The np**.**zeros and np**.**ones functions lets you create arrays of desired shape where all the items are either 0’s or 1’s.**

**import numpy as np**

**print(np.zeros([2,2]))**

**output:**

**[[0. 0.]**

**[0. 0.]]**

**import numpy as np**

**print(np.ones([2,2]))**

**output:**

**[[1. 1.]**

**[1. 1.]]**

## Creating repeating sequences

**np**.**tile will repeat a whole list or array n times. Whereas, np**.**repeat repeats each item n times.**

**a = [1,2,3]**

**# Repeat whole of 'a' two times**

**print('Tile: ', np.tile(a, 2)) # Tile: [1 2 3 1 2 3]**

**# Repeat each element of 'a' two times**

**print('Repeat: ', np.repeat(a, 2)) # Repeat: [1 1 2 2 3 3]**

**What is Pandas?**

* **Pandas is a Python library used for working with data sets.**
* **It has functions for analyzing, cleaning, exploring, and manipulating data.**
* **The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.**

**Why Use Pandas?**

* **Pandas allows us to analyze big data and make conclusions based on statistical theories.**
* **Pandas can clean messy data sets, and make them readable and relevant.**
* **Relevant data is very important in data science.**
* **Data Science: is a branch of computer science where we study how to store, use and analyze data for deriving information from it.**

**What Can Pandas Do?**

**Pandas gives you answers about the data. Like:**

* **Is there a correlation between two or more columns?**
* **What is average value?**
* **Max value?**
* **Min value?**

**Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called cleaning the data.**

****

**What is a DataFrame?**

**A Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns.**

**Example**

**import pandas**

**mydataset = {**

**'cars': ["BMW", "Volvo", "Ford"],**

**'passings': [3, 7, 2]**

**}**

**myvar = pandas.DataFrame(mydataset)**

**print(myvar)**

**output:**

**cars passings**

**0 BMW 3**

**1 Volvo 7**

**2 Ford 2**

**Pandas as pd**

**Pandas is usually imported under the pd alias.**

**Example**

**import pandas**

**mydataset = {**

**'cars': ["BMW", "Volvo", "Ford"],**

**'passings': [3, 7, 2]**

**}**

**myvar = pandas.DataFrame(mydataset)**

**print(myvar)**

**output:**

**cars passings**

**0 BMW 3**

**1 Volvo 7**

**2 Ford 2**

**What is a Series?**

* **A Pandas Series is like a column in a table.**
* **It is a one-dimensional array holding data of any type.**

**Example**

**Create a simple Pandas Series from a list:**

**import pandas as pd**

**a = [1, 7, 2]**

**myvar = pd.Series(a)**

**print(myvar)**

**output**

**0 1**

**1 7**

**2 2**

**Labels**

**If nothing else is specified, the values are labeled with their index number. First value has index 0, second value has index 1 etc.**

**Create Labels**

**With the index argument, you can name your own labels.**

**Example**

**import pandas as pd**

**a = [1, 7, 2]**

**myvar = pd.Series(a, index = ["x", "y", "z"])**

**print(myvar)**

**output:**

**x 1**

**y 7**

**z 2**

**dtype: int64**

**Key/Value Objects as Series**

**You can also use a key/value object, like a dictionary, when creating a Series.**

**Create a simple Pandas Series from a dictionary:**

**import pandas as pd  
calories = {"day1": 420, "day2": 380, "day3": 390}  
myvar = pd.Series(calories)  
print(myvar)**

**output:**

**day1 420**

**day2 380**

**day3 390**

**dtype: int64**

**import pandas as pd  
data = {  
  "calories": [420, 380, 390],  
  "duration": [50, 40, 45]  
}  
df = pd.DataFrame(data)  
print(df)**

**output:**

**calories duration**

**0 420 50**

**1 380 40**

**2 390 45**

**Locate Row**

**As you can see from the result above, the DataFrame is like a table with rows and columns.**

**Pandas use the loc attribute to return one or more specified row(s)**

**import pandas as pd**

**data = {**

**"calories": [420, 380, 390],**

**"duration": [50, 40, 45]**

**}**

**df = pd.DataFrame(data)**

**print(df.loc[0])**

**output:**

**calories 420**

**duration 50**

**Name: 0, dtype: int64**

**Example**

**Return row 0 and 1:**

**import pandas as pd**

**data = {**

**"calories": [420, 380, 390],**

**"duration": [50, 40, 45]**

**}**

**df = pd.DataFrame(data)**

**print(df.loc[[0, 1]])**

**output:**

**calories duration**

**0 420 50**

**1 380 40**

**Named Indexes**

**With the index argument, you can name your own indexes.**

**Example**

**Add a list of names to give each row a name:**

**import pandas as pd  
  
data = {  
  "calories": [420, 380, 390],  
  "duration": [50, 40, 45]  
}  
df = pd.DataFrame(data, index = ["day1", "day2", "day3"])  
print(df)**

**Result**

**calories duration**

**day1 420 50**

**day2 380 40**

**day3 390 45**

**Locate Named Indexes**

**Use the named index in the loc attribute to return the specified row(s).**

**Example**

**Return "day2":**

**#refer to the named index:  
print(df.loc["day2"])**

**Result**

**calories 380**

**duration 40**

**Name: 0, dtype: int64**

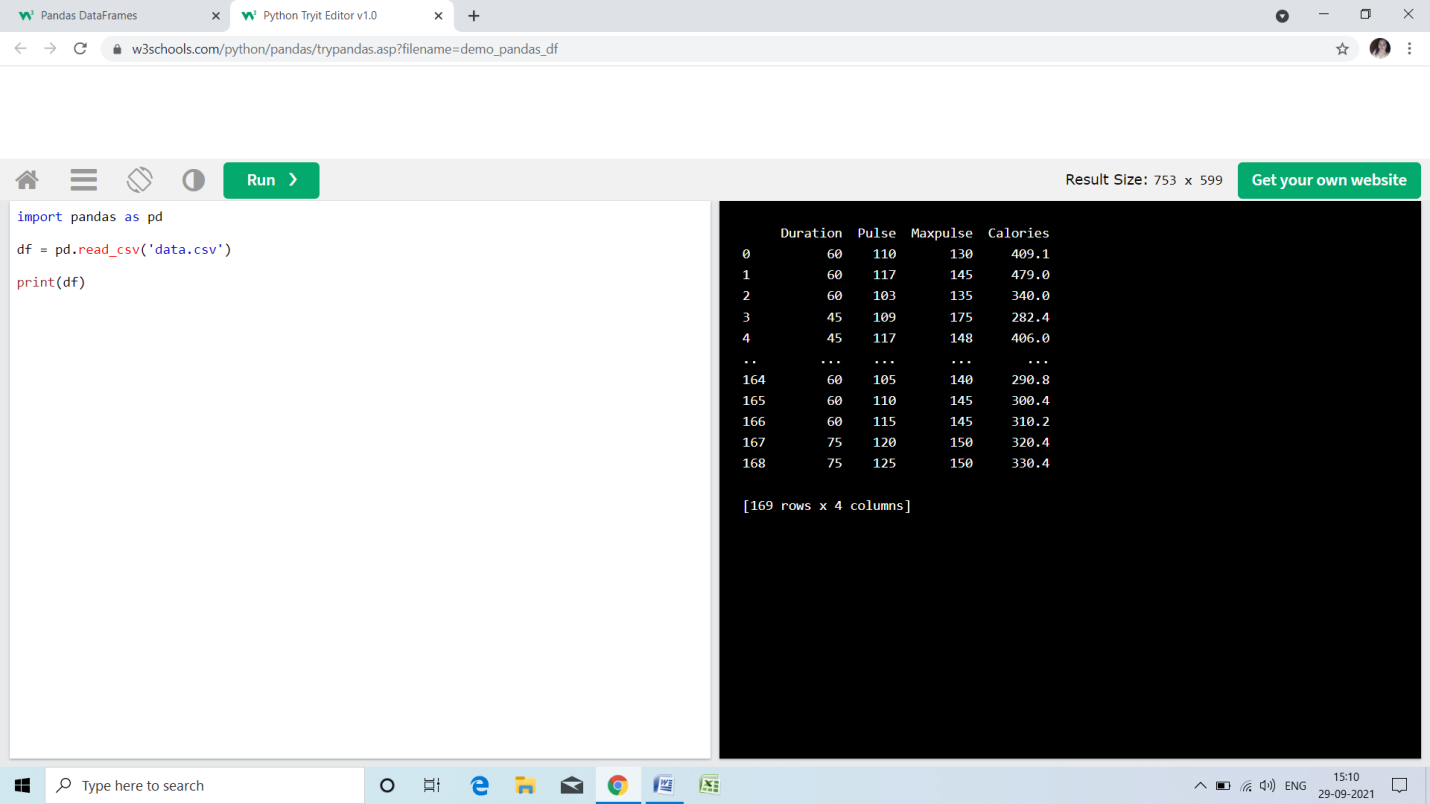
**Load Files Into a DataFrame**

* **If your data sets are stored in a file, Pandas can load them into a DataFrame.**

**Example**

* **Load a comma separated file (CSV file) into a DataFrame:**

**import pandas as pd  
df = pd.read\_csv('data.csv')  
print(df)**

****

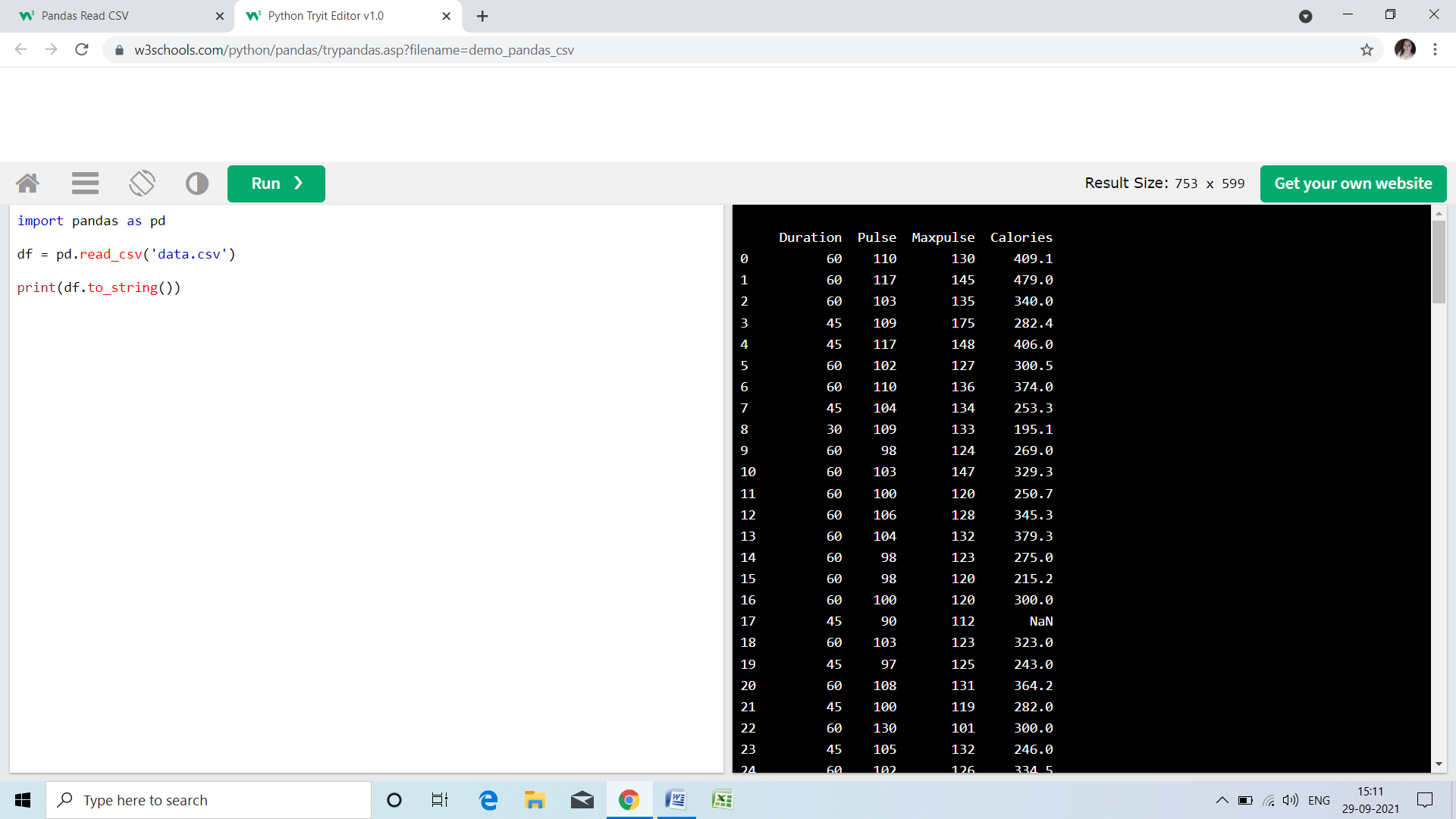
**Read CSV Files**

* **A simple way to store big data sets is to use CSV files (comma separated files).**
* **CSV files contains plain text and is a well know format that can be read by everyone including Pandas.**
* [**Download data.csv**](https://www.w3schools.com/python/pandas/data.csv)**. or**[**Open data.csv**](https://www.w3schools.com/python/pandas/data.csv.txt)

**Example**

**Load the CSV into a DataFrame:**

**import pandas as pd  
df = pd.read\_csv('data.csv')  
print(df.to\_string())**

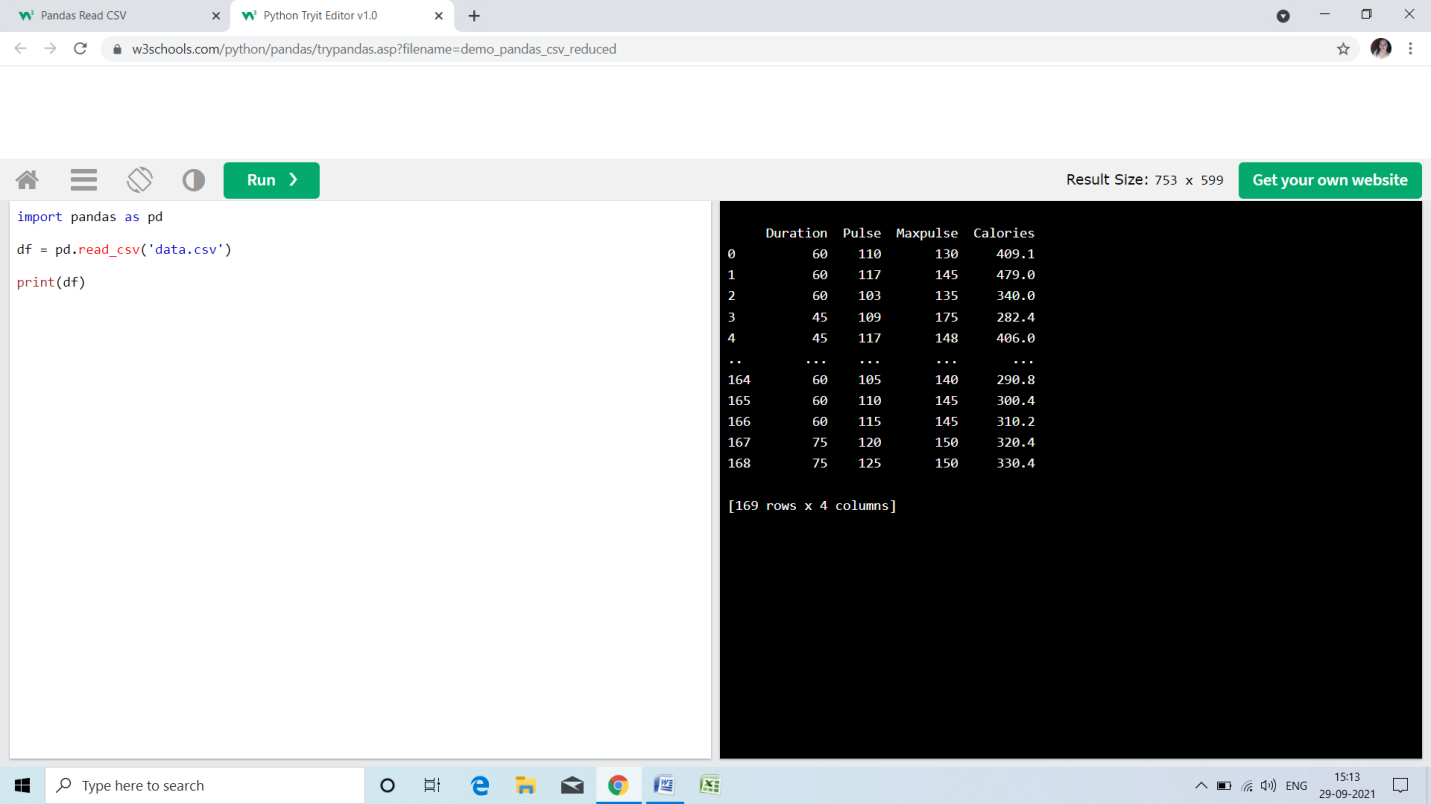
****

**By default, when you print a DataFrame, you will only get the first 5 rows, and the last 5 rows:**

**Example**

**Print a reduced sample:**

**import pandas as pd  
df = pd.read\_csv('data.csv')  
print(df)**

****

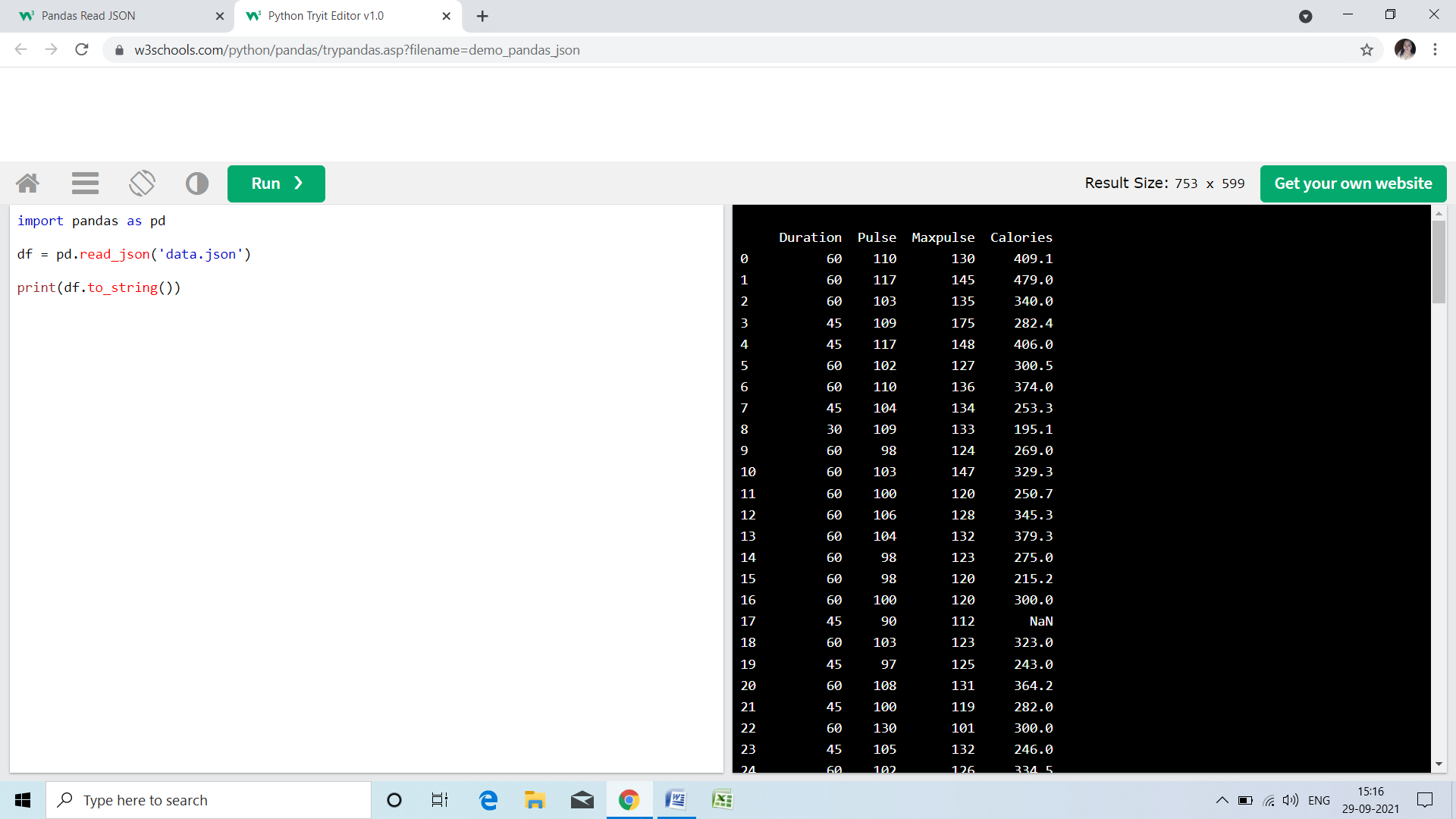
**Read JSON**

* **Big data sets are often stored, or extracted as JSON.**
* **JSON is plain text, but has the format of an object, and is well known in the world of programming, including Pandas.**
* **In our examples we will be using a JSON file called 'data.json'.**

**Example**

**Load the JSON file into a DataFrame:**

**import pandas as pd  
df = pd.read\_json('data.json')  
print(df.to\_string())**

****

**Dictionary as JSON**

**JSON = Python Dictionary**

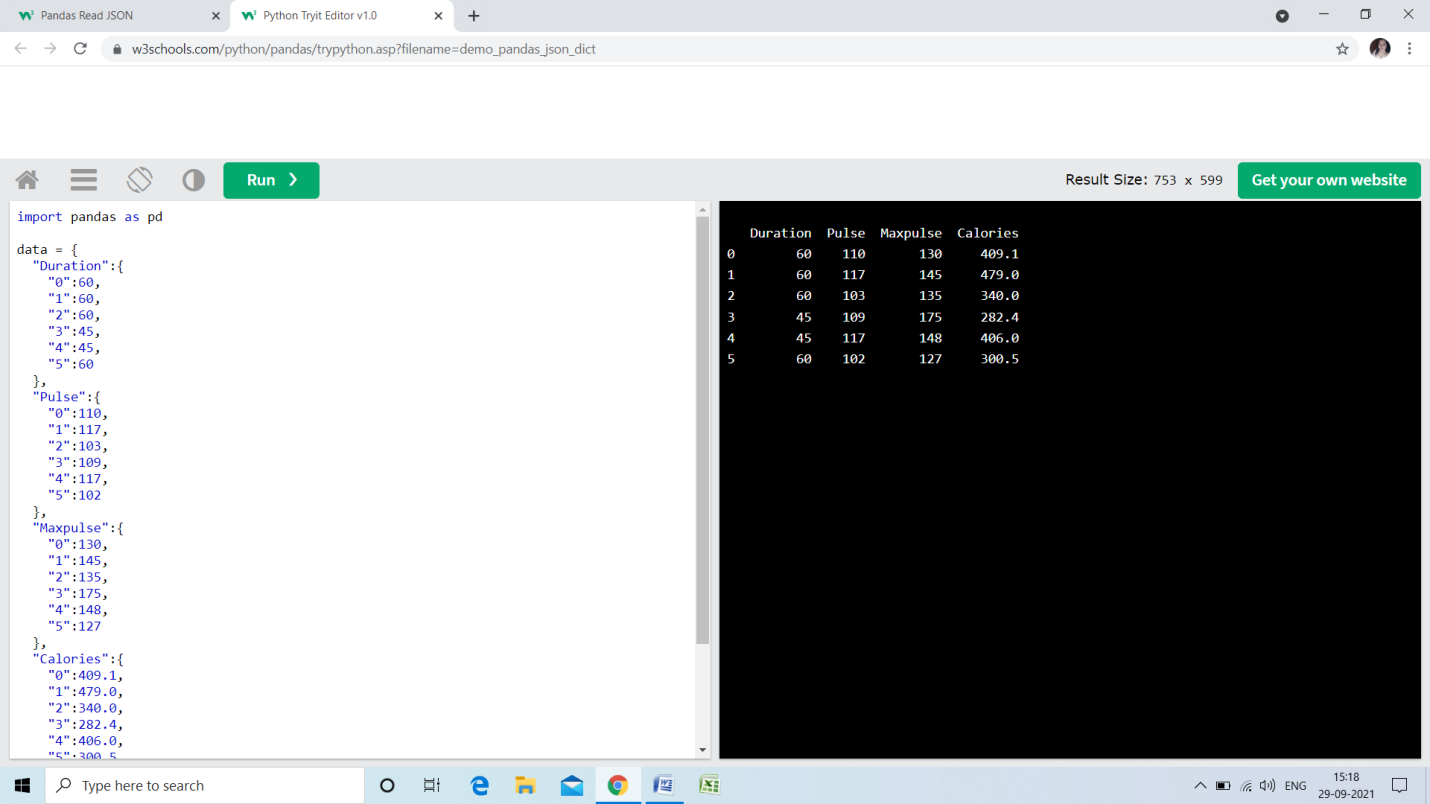
**JSON objects have the same format as Python dictionaries.**

**If your JSON code is not in a file, but in a Python Dictionary, you can load it into a DataFrame directly:**

**Example**

**Load a Python Dictionary into a DataFrame:**

**import pandas as pd  
data = {  
  "Duration":{  
    "0":60,  
    "1":60,  
    "2":60,  
    "3":45,  
    "4":45,  
    "5":60  
  },  
  "Pulse":{  
    "0":110,  
    "1":117,  
    "2":103,  
    "3":109,  
    "4":117,  
    "5":102  
  },  
  "Maxpulse":{  
    "0":130,  
    "1":145,  
    "2":135,  
    "3":175,  
    "4":148,  
    "5":127  
  },  
  "Calories":{  
    "0":409,  
    "1":479,  
    "2":340,  
    "3":282,  
    "4":406,  
    "5":300  
  }  
}  
df = pd.DataFrame(data)  
print(df)**

****

**Viewing the Data**

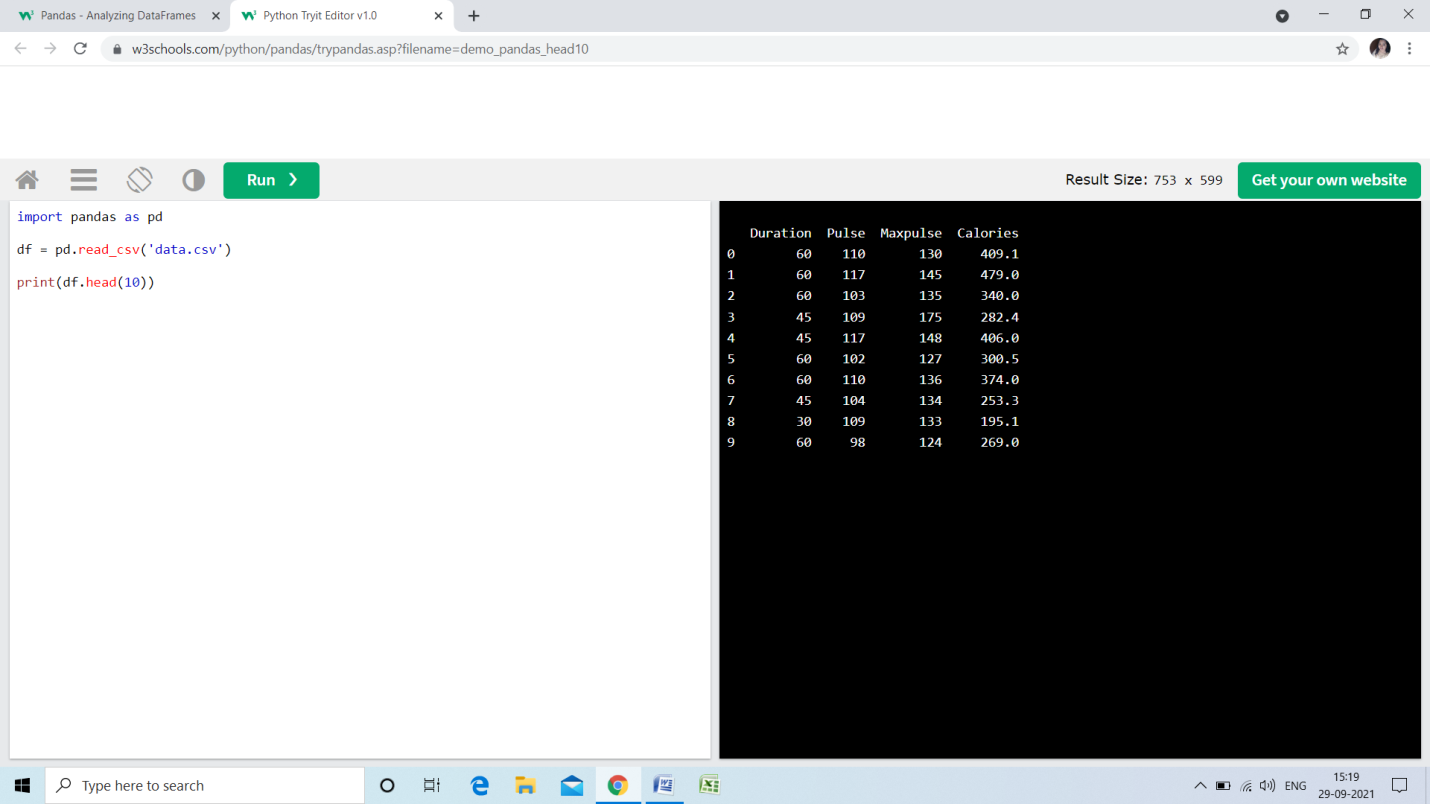
**One of the most used method for getting a quick overview of the DataFrame, is the head() method.**

**The head() method returns the headers and a specified number of rows, starting from the top.**

**Example**

**Get a quick overview by printing the first 10 rows of the DataFrame:**

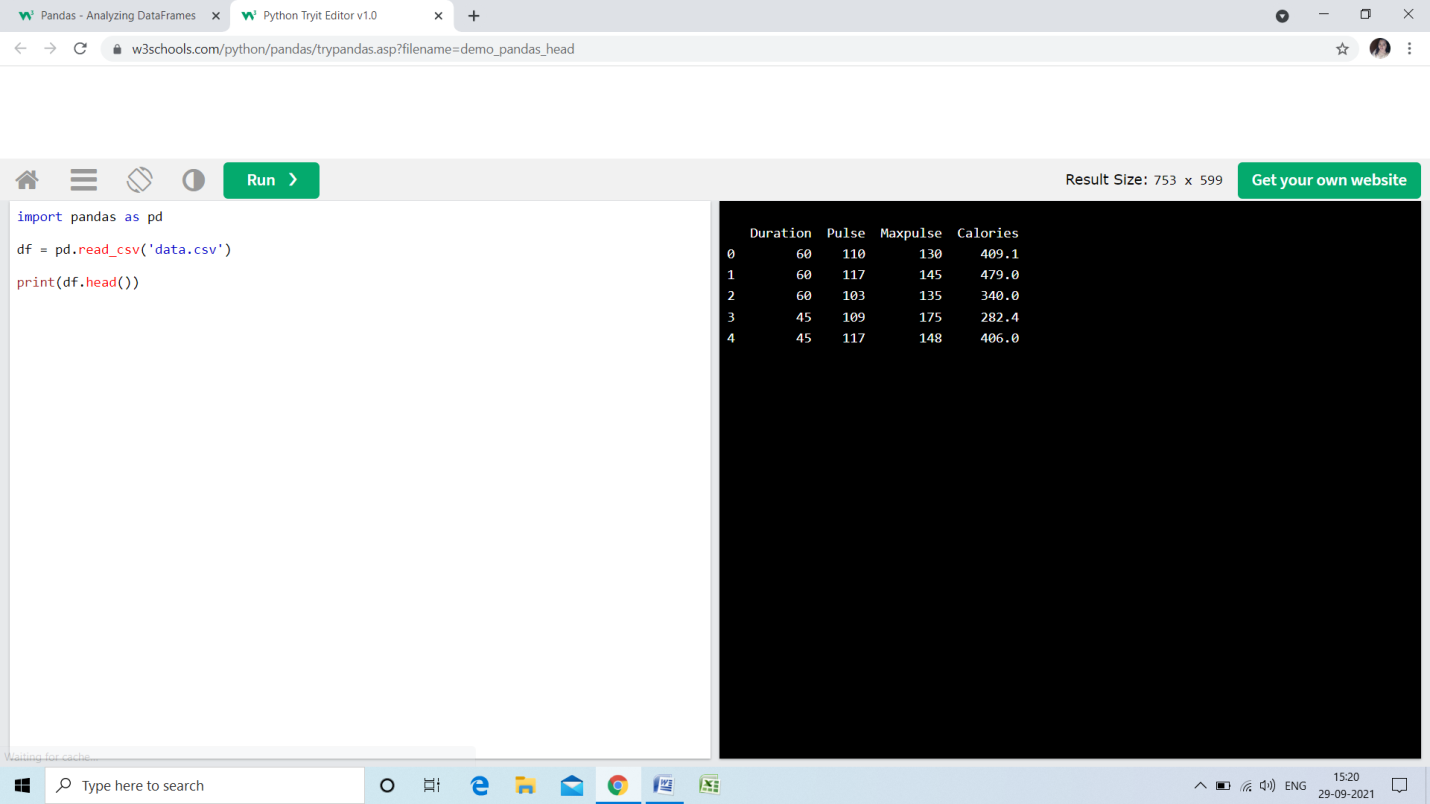
**import pandas as pd  
df = pd.read\_csv('data.csv')  
print(df.head(10))**

****

**Example**

* **Print the first 5 rows of the DataFrame:**

**import pandas as pd  
df = pd.read\_csv('data.csv')  
print(df.head())**

****

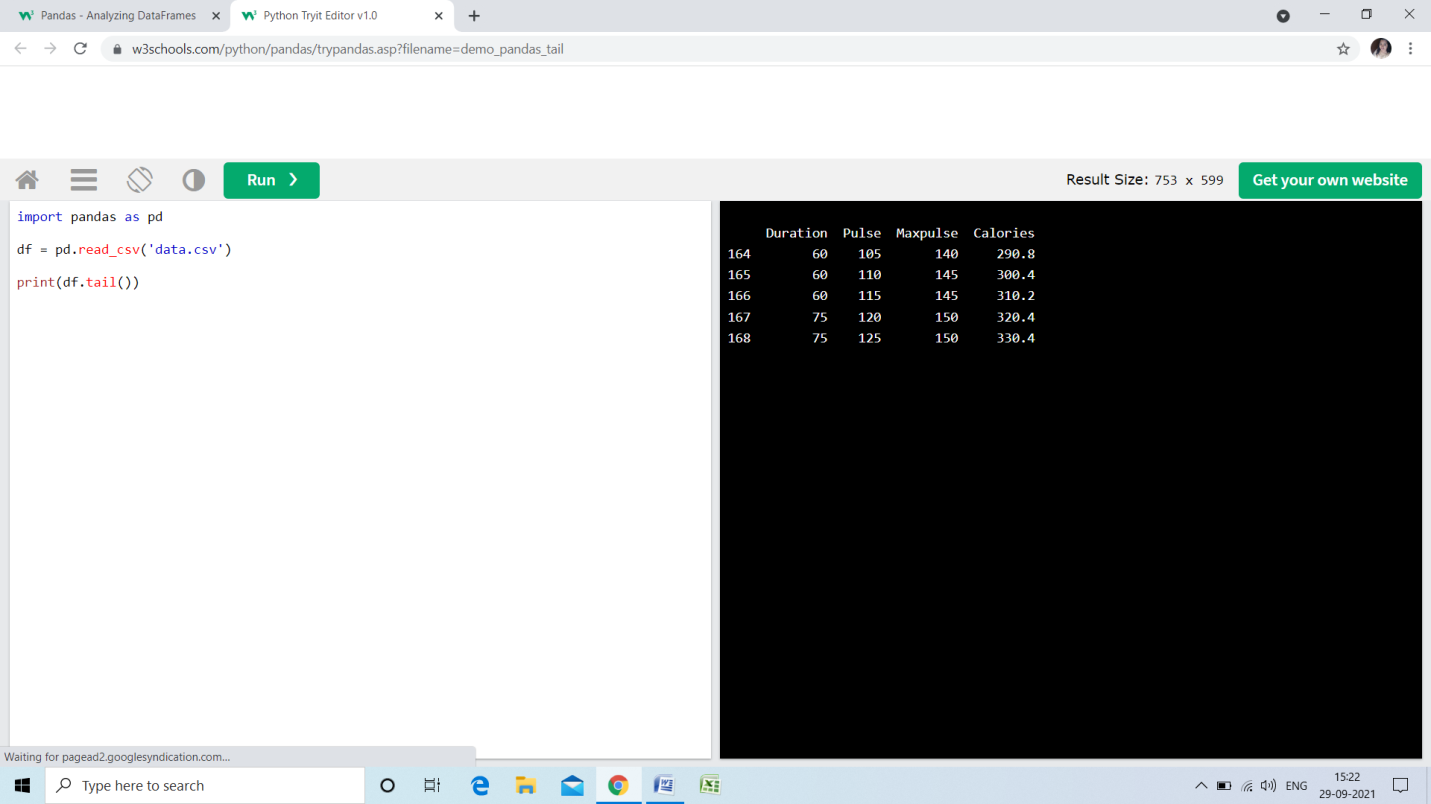
**There is also a tail() method for viewing the *last* rows of the DataFrame.**

**The tail() method returns the headers and a specified number of rows, starting from the bottom.**

**Example**

**Print the last 5 rows of the DataFrame:**

**print(df.tail())**

****

**Info About the Data**

**The DataFrames object has a method called info(), that gives you more information about the data set.**

**Example**

**Print information about the data:**

**print(df.info())**

**Result**

**<class 'pandas.core.frame.DataFrame'>**

**RangeIndex: 169 entries, 0 to 168**

**Data columns (total 4 columns):**

**# Column Non-Null Count Dtype**

**--- ------ -------------- -----**

**0 Duration 169 non-null int64**

**1 Pulse 169 non-null int64**

**2 Maxpulse 169 non-null int64**

**3 Calories 164 non-null float64**

**dtypes: float64(1), int64(3)**

**memory usage: 5.4 KB**

**None**

**Result Explained**

**The result tells us there are 169 rows and 4 columns:**

**RangeIndex: 169 entries, 0 to 168**

**Data columns (total 4 columns):**

**And the name of each column, with the data type:**

**# Column Non-Null Count Dtype**

**--- ------ -------------- -----**

**0 Duration 169 non-null int64**

**1 Pulse 169 non-null int64**

**2 Maxpulse 169 non-null int64**

**3 Calories 164 non-null float64**

**Null Values**

* **The info() method also tells us how many Non-Null values there are present in each column, and in our data set it seems like there are 164 of 169 Non-Null values in the "Calories" column.**
* **Which means that there are 5 rows with no value at all, in the "Calories" column, for whatever reason.**
* **Empty values, or Null values, can be bad when analyzing data, and you should consider removing rows with empty values. This is a step towards what is called *cleaning data*, and you will learn more about that in the next chapters.**

**Data Cleaning**

**Data cleaning means fixing bad data in your data set.**

* **Bad data could be:**
* **Empty cells**
* **Data in wrong format**
* **Wrong data**
* **Duplicates**

**Our Data Set**

**Duration Date Pulse Maxpulse Calories**

**0 60 '2020/12/01' 110 130 409.1**

**1 60 '2020/12/02' 117 145 479.0**

**2 60 '2020/12/03' 103 135 340.0**

**3 45 '2020/12/04' 109 175 282.4**

**4 45 '2020/12/05' 117 148 406.0**

**5 60 '2020/12/06' 102 127 300.0**

**6 60 '2020/12/07' 110 136 374.0**

**7 450 '2020/12/08' 104 134 253.3**

**8 30 '2020/12/09' 109 133 195.1**

**9 60 '2020/12/10' 98 124 269.0**

**10 60 '2020/12/11' 103 147 329.3**

**11 60 '2020/12/12' 100 120 250.7**

**12 60 '2020/12/12' 100 120 250.7**

**13 60 '2020/12/13' 106 128 345.3**

**14 60 '2020/12/14' 104 132 379.3**

**15 60 '2020/12/15' 98 123 275.0**

**16 60 '2020/12/16' 98 120 215.2**

**17 60 '2020/12/17' 100 120 300.0**

**18 45 '2020/12/18' 90 112 NaN**

**19 60 '2020/12/19' 103 123 323.0**

**20 45 '2020/12/20' 97 125 243.0**

**21 60 '2020/12/21' 108 131 364.2**

**22 45 NaN 100 119 282.0**

**23 60 '2020/12/23' 130 101 300.0**

**24 45 '2020/12/24' 105 132 246.0**

**25 60 '2020/12/25' 102 126 334.5**

**26 60 2020/12/26 100 120 250.0**

**27 60 '2020/12/27' 92 118 241.0**

**28 60 '2020/12/28' 103 132 NaN**

**29 60 '2020/12/29' 100 132 280.0**

**30 60 '2020/12/30' 102 129 380.3**

**31 60 '2020/12/31' 92 115 243.0**

* **The data set contains some empty cells ("Date" in row 22, and "Calories" in row 18 and 28).**
* **The data set contains wrong format ("Date" in row 26).**
* **The data set contains wrong data ("Duration" in row 7).**
* **The data set contains duplicates (row 11 and 12).**

**Empty Cells**

**Empty cells can potentially give you a wrong result when you analyze data.**

**Remove Rows**

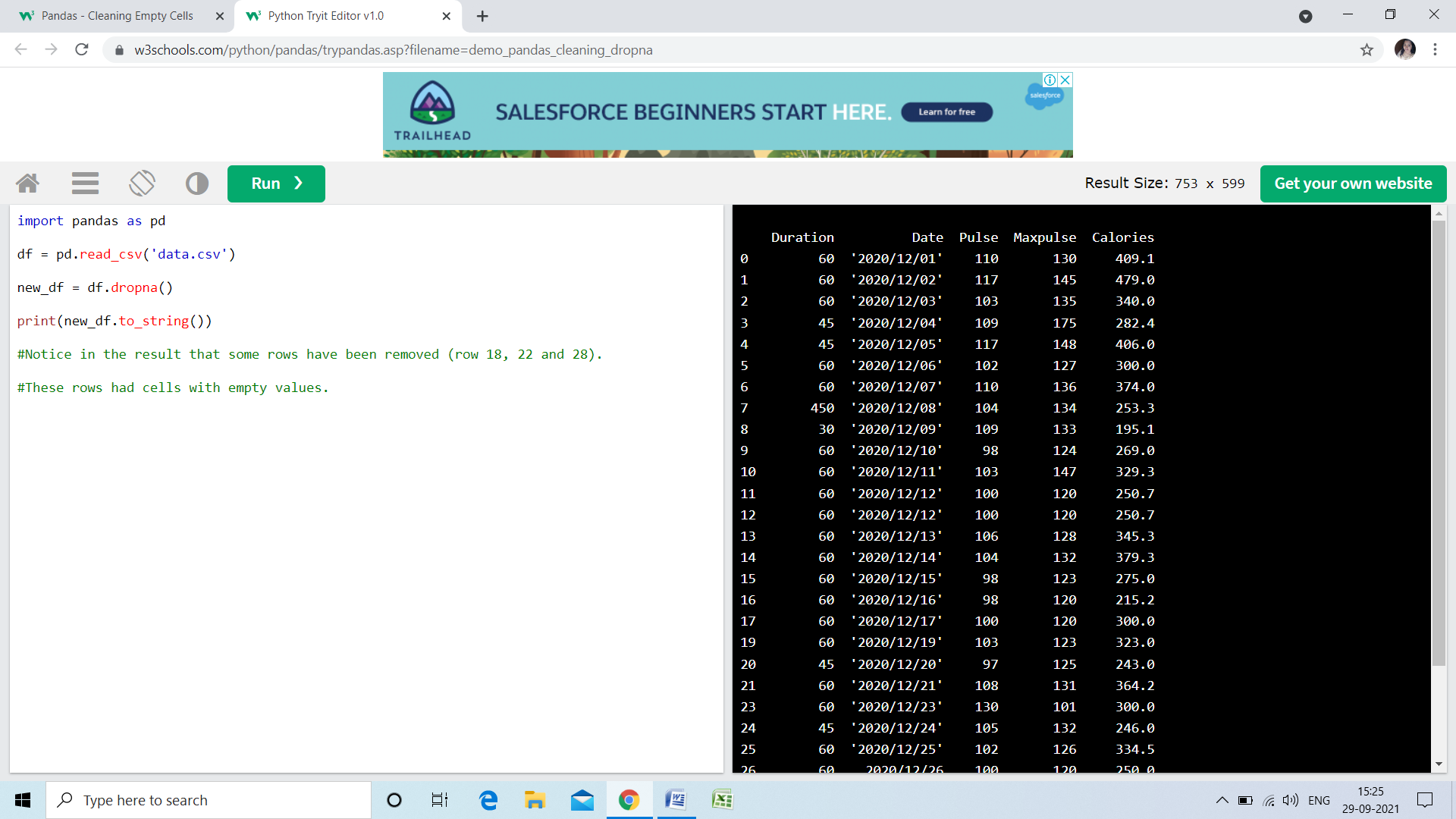
**One way to deal with empty cells is to remove rows that contain empty cells.**

**This is usually OK, since data sets can be very big, and removing a few rows will not have a big impact on the result.**

**Example**

**Return a new Data Frame with no empty cells:**

**import pandas as pd  
df = pd.read\_csv('data.csv')  
new\_df = df.dropna()  
print(new\_df.to\_string())**

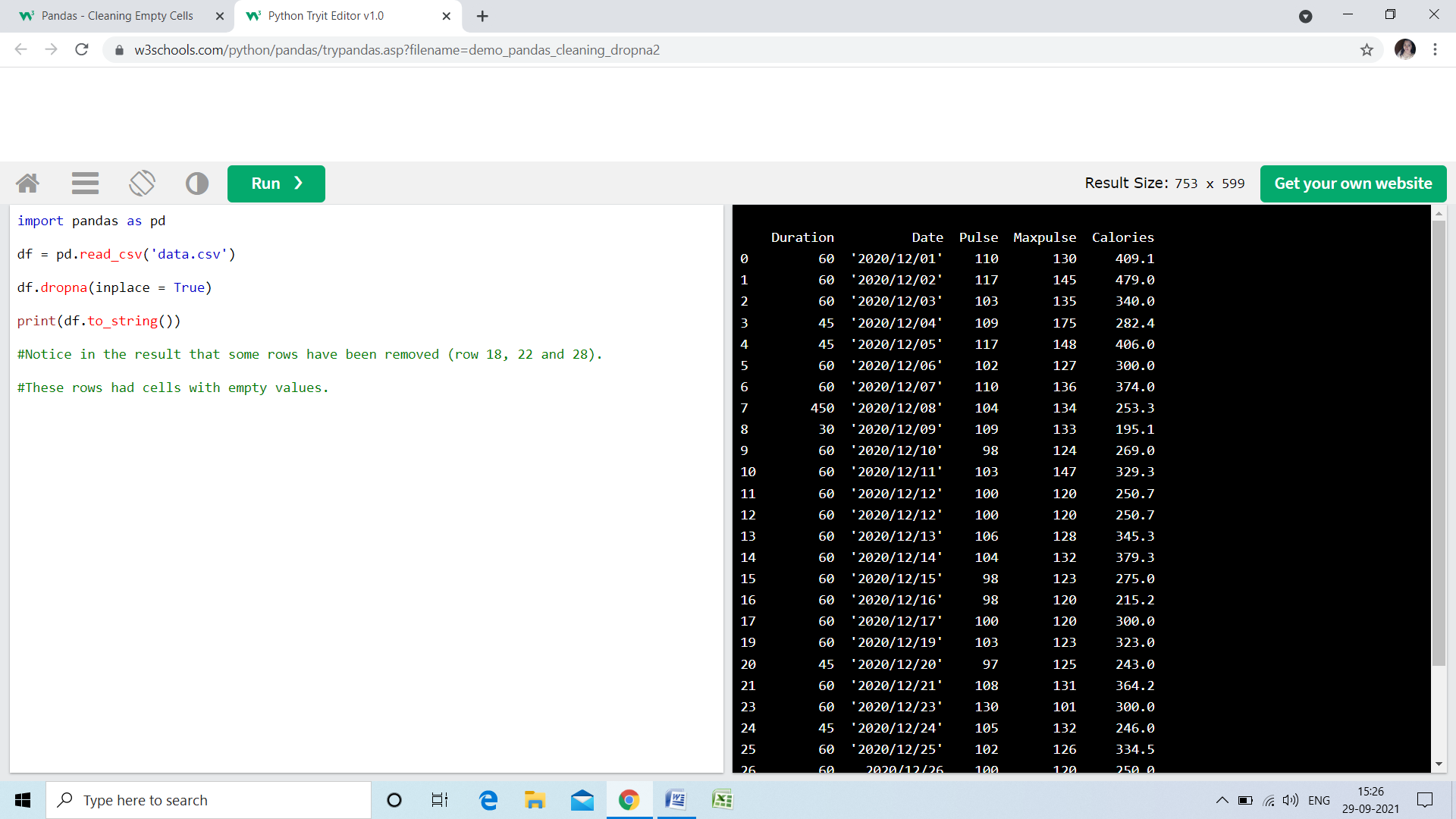
****

**If you want to change the original DataFrame, use the inplace = True argument:**

**Example**

**Remove all rows with NULL values:**

**import pandas as pd  
df = pd.read\_csv('data.csv')  
df.dropna(inplace = True)  
print(df.to\_string())**

****

**Replace Empty Values**

**Another way of dealing with empty cells is to insert a *new* value instead.**

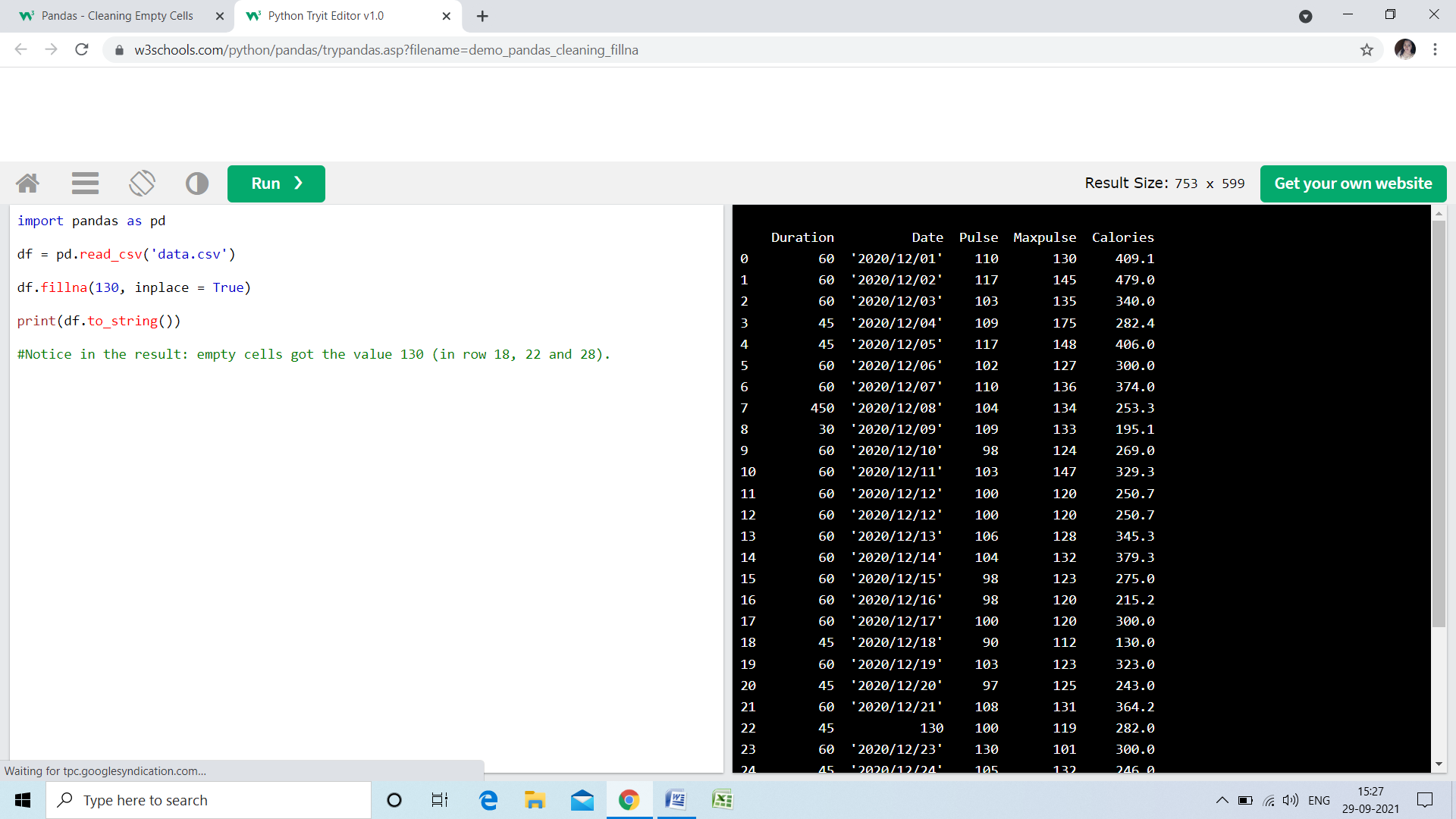
**This way you do not have to delete entire rows just because of some empty cells.**

**The fillna() method allows us to replace empty cells with a value:**

**Example**

**Replace NULL values with the number 130:**

**import pandas as pd  
df = pd.read\_csv('data.csv')  
df.fillna(130, inplace = True)**

****

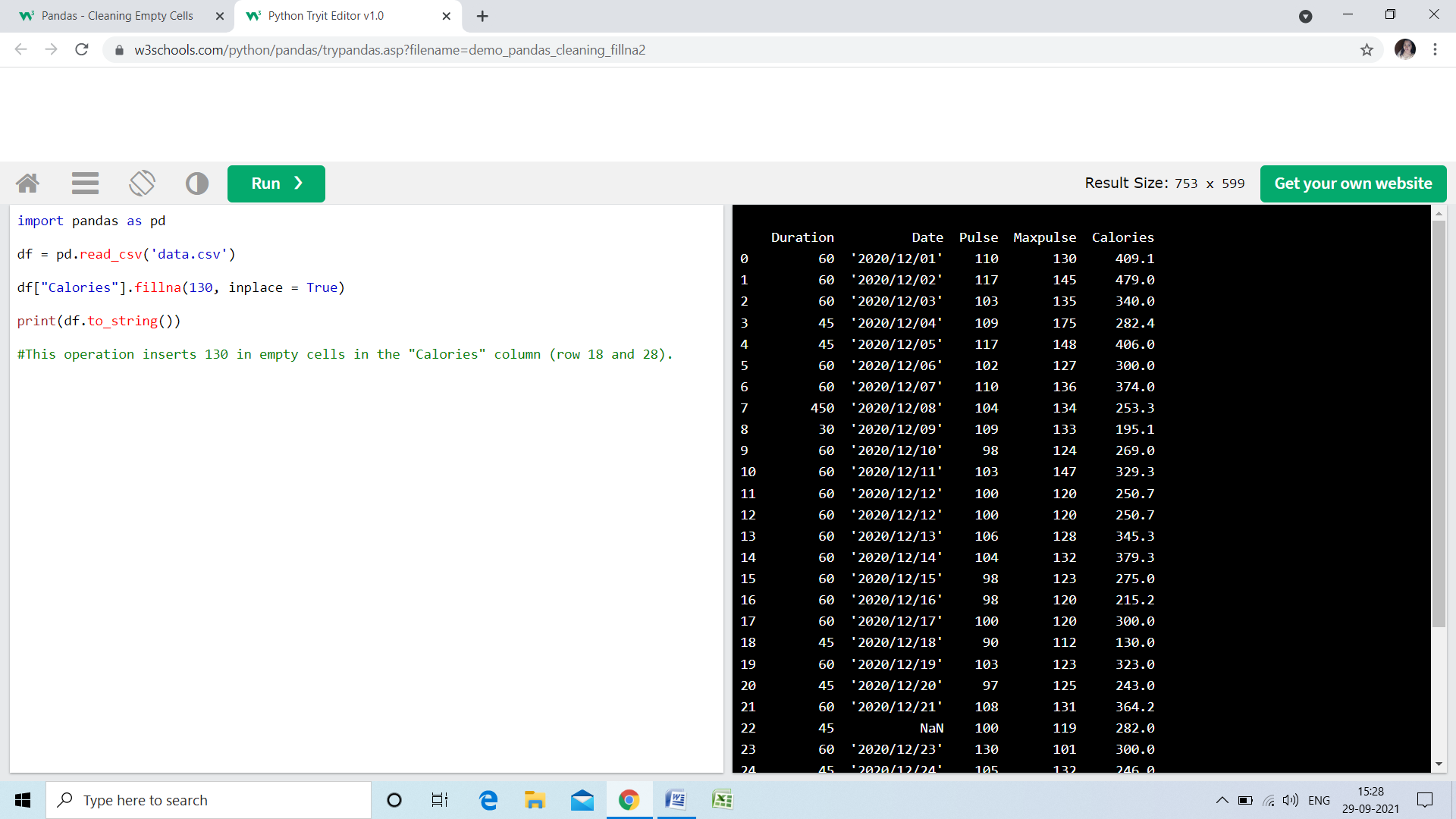
**Replace Only For a Specified Columns**

* **The example above replaces all empty cells in the whole Data Frame.**
* **To only replace empty values for one column, specify the *column name* for the DataFrame:**

**Example**

**Replace NULL values in the "Calories" columns with the number 130:**

**import pandas as pd  
df = pd.read\_csv('data.csv')  
df["Calories"].fillna(130, inplace = True)**

****

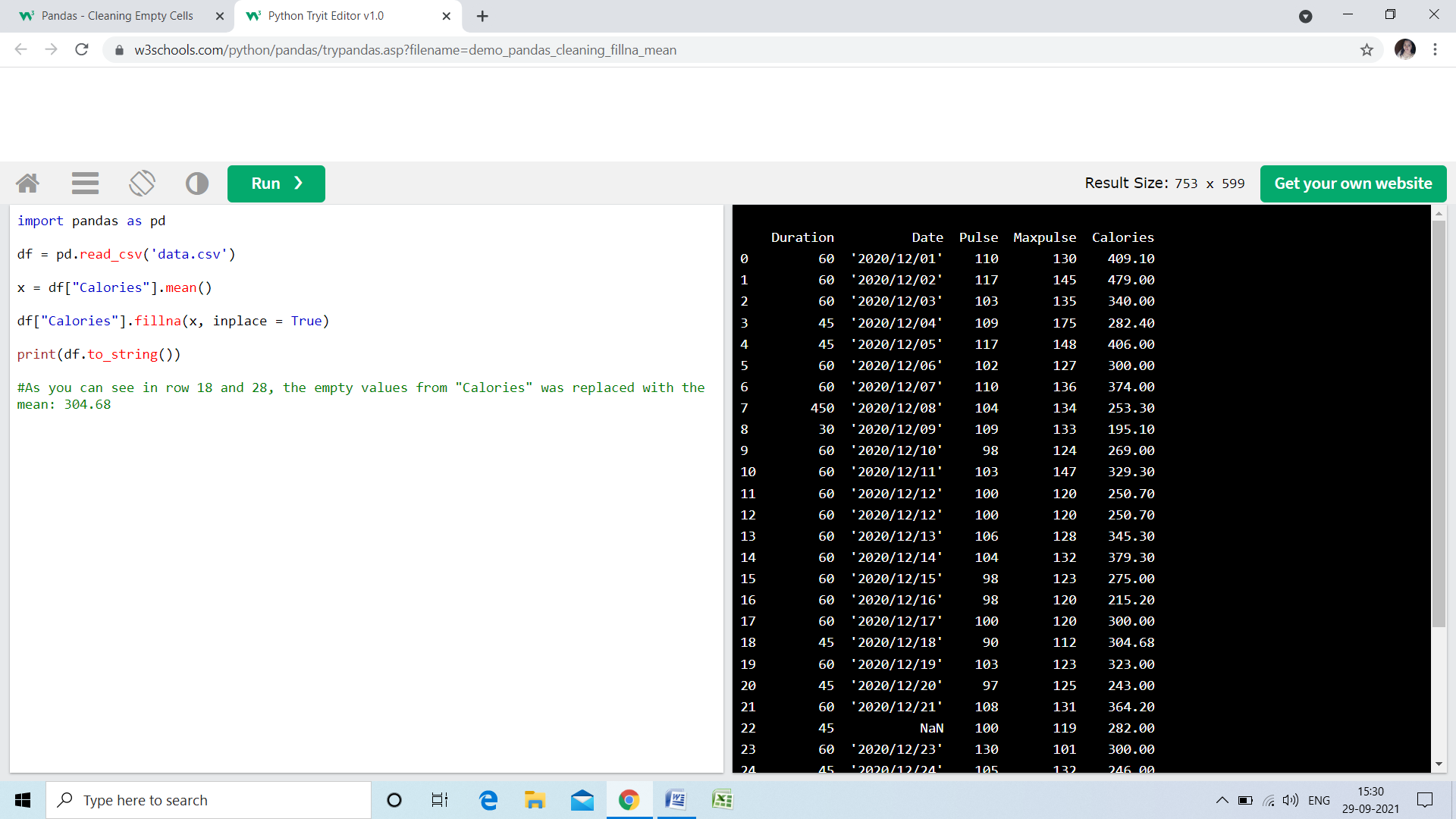
**Replace Using Mean, Median, or Mode**

* **A common way to replace empty cells, is to calculate the mean, median or mode value of the column.**
* **Pandas uses the mean() median() and mode() methods to calculate the respective values for a specified column:**

**Example**

**Calculate the MEAN, and replace any empty values with it:**

**import pandas as pd  
df = pd.read\_csv('data.csv')  
x = df["Calories"].mean()  
df["Calories"].fillna(x, inplace = True)**

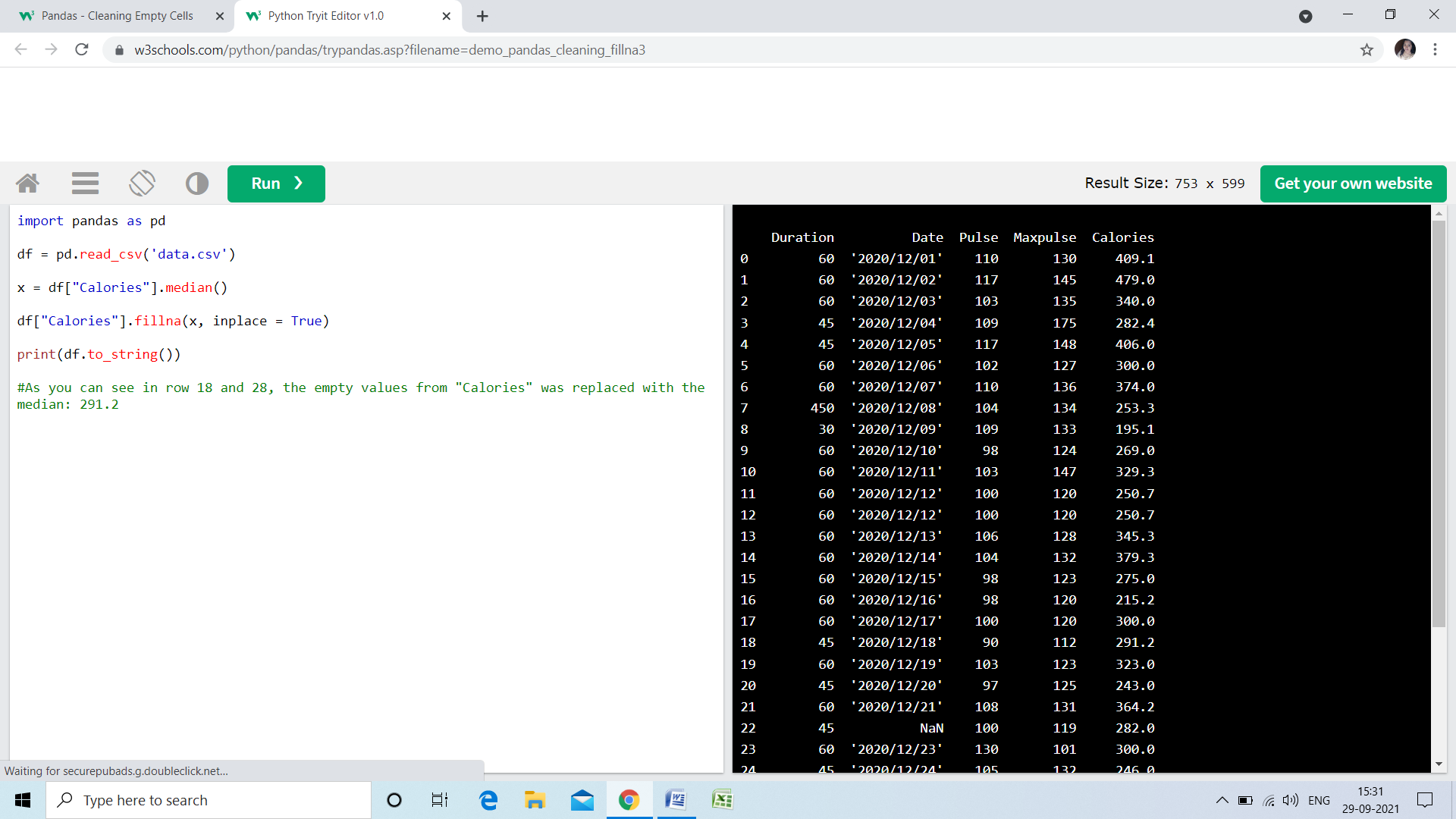
****

**Mean = the average value (the sum of all values divided by number of values).**

**Example**

**Calculate the MEDIAN, and replace any empty values with it:**

**import pandas as pd  
  
df = pd.read\_csv('data.csv')  
  
x = df["Calories"].median()  
  
df["Calories"].fillna(x, inplace = True)**

****

**Median = the value in the middle, after you have sorted all values ascending.**

**Example**

**Calculate the MODE, and replace any empty values with it:**

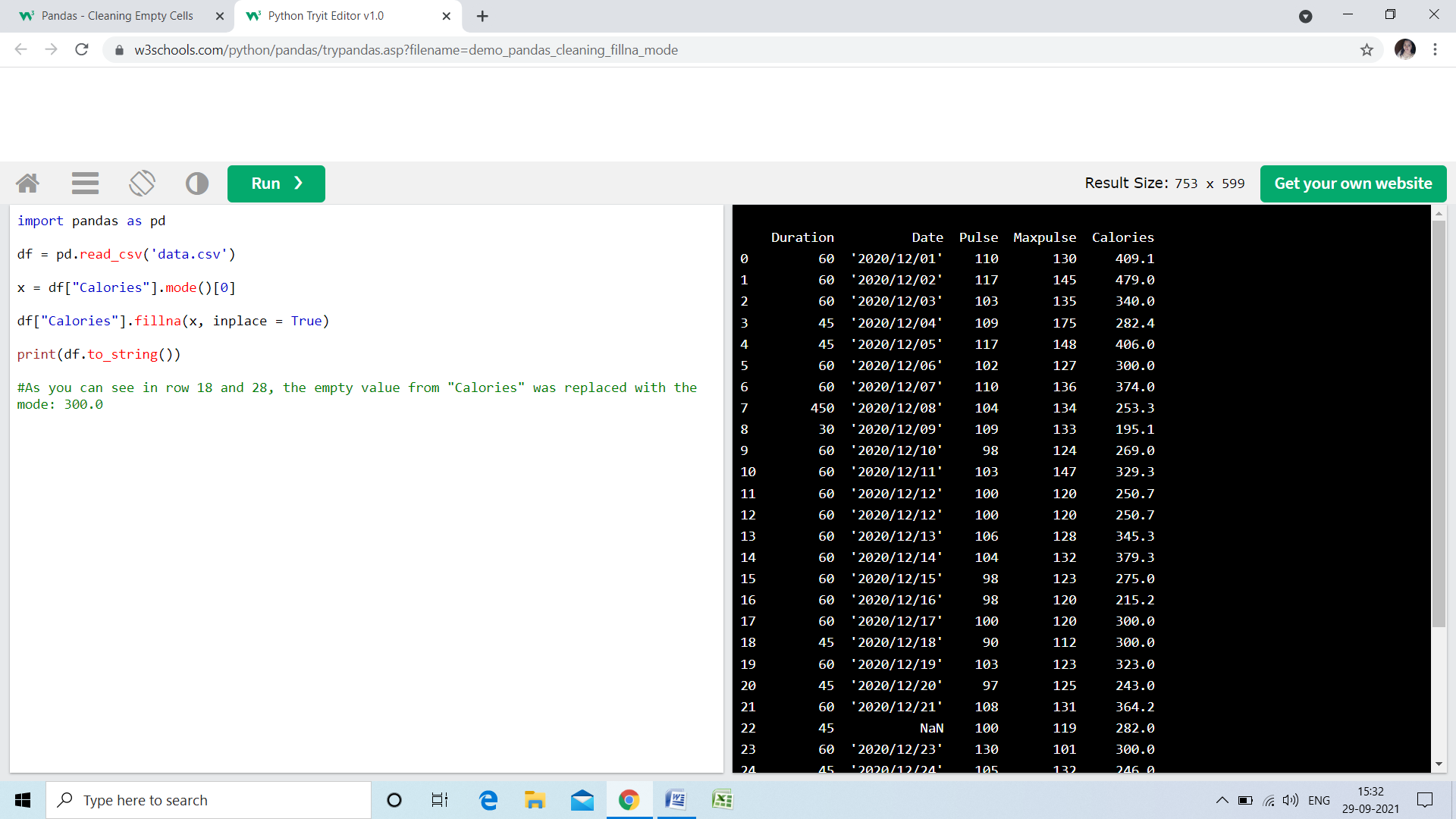
**import pandas as pd  
df = pd.read\_csv('data.csv')  
x = df["Calories"].mode()[0]  
df["Calories"].fillna(x, inplace = True)**

**Median = the value in the middle, after you have sorted all values ascending.**

**Example**

**Calculate the MODE, and replace any empty values with it:**

**import pandas as pd  
df = pd.read\_csv('data.csv')  
x = df["Calories"].mode()[0]  
df["Calories"].fillna(x, inplace = True)**

****

**Mode = the value that appears most frequently.**

**Data of Wrong Format**

* **Cells with data of wrong format can make it difficult, or even impossible, to analyze data.**
* **To fix it, you have two options: remove the rows, or convert all cells in the columns into the same format.**

**Convert Into a Correct Format**

**In our Data Frame, we have two cells with the wrong format. Check out row 22 and 26, the 'Date' column should be a string that represents a date:**

**Duration Date Pulse Maxpulse Calories**

**0 60 '2020/12/01' 110 130 409.1**

**1 60 '2020/12/02' 117 145 479.0**

**2 60 '2020/12/03' 103 135 340.0**

**3 45 '2020/12/04' 109 175 282.4**

**4 45 '2020/12/05' 117 148 406.0**

**5 60 '2020/12/06' 102 127 300.0**

**6 60 '2020/12/07' 110 136 374.0**

**7 450 '2020/12/08' 104 134 253.3**

**8 30 '2020/12/09' 109 133 195.1**

**9 60 '2020/12/10' 98 124 269.0**

**10 60 '2020/12/11' 103 147 329.3**

**11 60 '2020/12/12' 100 120 250.7**

**12 60 '2020/12/12' 100 120 250.7**

**13 60 '2020/12/13' 106 128 345.3**

**14 60 '2020/12/14' 104 132 379.3**

**15 60 '2020/12/15' 98 123 275.0**

**16 60 '2020/12/16' 98 120 215.2**

**17 60 '2020/12/17' 100 120 300.0**

**18 45 '2020/12/18' 90 112 NaN**

**19 60 '2020/12/19' 103 123 323.0**

**20 45 '2020/12/20' 97 125 243.0**

**21 60 '2020/12/21' 108 131 364.2**

**22 45 NaN 100 119 282.0**

**23 60 '2020/12/23' 130 101 300.0**

**24 45 '2020/12/24' 105 132 246.0**

**25 60 '2020/12/25' 102 126 334.5**

**26 60 20201226 100 120 250.0**

**27 60 '2020/12/27' 92 118 241.0**

**28 60 '2020/12/28' 103 132 NaN**

**29 60 '2020/12/29' 100 132 280.0**

**30 60 '2020/12/30' 102 129 380.3**

**31 60 '2020/12/31' 92 115 243.0**

**Let's try to convert all cells in the 'Date' column into dates.**

**Pandas has a to\_datetime() method for this:**

**Example**

**Convert to date:**

**import pandas as pd  
df = pd.read\_csv('data.csv')  
df['Date'] = pd.to\_datetime(df['Date'])  
print(df.to\_string())**

**Result:**

**Duration Date Pulse Maxpulse Calories**

**0 60 '2020/12/01' 110 130 409.1**

**1 60 '2020/12/02' 117 145 479.0**

**2 60 '2020/12/03' 103 135 340.0**

**3 45 '2020/12/04' 109 175 282.4**

**4 45 '2020/12/05' 117 148 406.0**

**5 60 '2020/12/06' 102 127 300.0**

**6 60 '2020/12/07' 110 136 374.0**

**7 450 '2020/12/08' 104 134 253.3**

**8 30 '2020/12/09' 109 133 195.1**

**9 60 '2020/12/10' 98 124 269.0**

**10 60 '2020/12/11' 103 147 329.3**

**11 60 '2020/12/12' 100 120 250.7**

**12 60 '2020/12/12' 100 120 250.7**

**13 60 '2020/12/13' 106 128 345.3**

**14 60 '2020/12/14' 104 132 379.3**

**15 60 '2020/12/15' 98 123 275.0**

**16 60 '2020/12/16' 98 120 215.2**

**17 60 '2020/12/17' 100 120 300.0**

**18 45 '2020/12/18' 90 112 NaN**

**19 60 '2020/12/19' 103 123 323.0**

**20 45 '2020/12/20' 97 125 243.0**

**21 60 '2020/12/21' 108 131 364.2**

**22 45 NaT 100 119 282.0**

**23 60 '2020/12/23' 130 101 300.0**

**24 45 '2020/12/24' 105 132 246.0**

**25 60 '2020/12/25' 102 126 334.5**

**26 60 '2020/12/26' 100 120 250.0**

**27 60 '2020/12/27' 92 118 241.0**

**28 60 '2020/12/28' 103 132 NaN**

**29 60 '2020/12/29' 100 132 280.0**

**30 60 '2020/12/30' 102 129 380.3**

**31 60 '2020/12/31' 92 115 243.0**

**As you can see from the result, the date in row 26 was fixed, but the empty date in row 22 got a NaT (Not a Time) value, in other words an empty value. One way to deal with empty values is simply removing the entire row.**

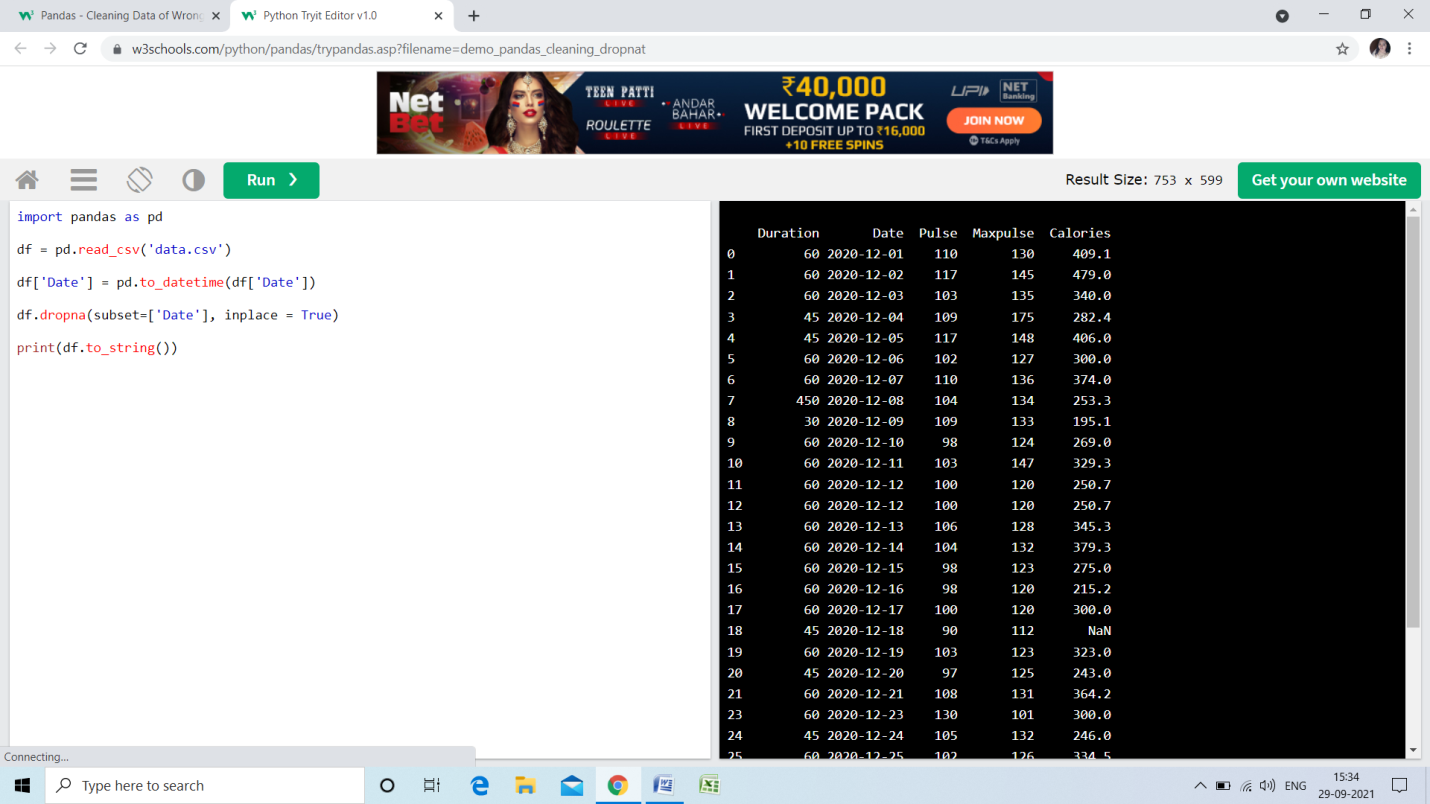
**Removing Rows**

**The result from the converting in the example above gave us a NaT value, which can be handled as a NULL value, and we can remove the row by using the dropna() method.**

**Example**

**Remove rows with a NULL value in the "Date" column:**

**df.dropna(subset=['Date'], inplace = True)**

****

**Wrong Data**

* **"Wrong data" does not have to be "empty cells" or "wrong format", it can just be wrong, like if someone registered "199" instead of "1.99".**
* **Sometimes you can spot wrong data by looking at the data set, because you have an expectation of what it should be.**
* **If you take a look at our data set, you can see that in row 7, the duration is 450, but for all the other rows the duration is between 30 and 60.**
* **It doesn't have to be wrong, but taking in consideration that this is the data set of someone's workout sessions, we conclude with the fact that this person did not work out in 450 minutes.**

**Duration Date Pulse Maxpulse Calories**

**0 60 '2020/12/01' 110 130 409.1**

**1 60 '2020/12/02' 117 145 479.0**

**2 60 '2020/12/03' 103 135 340.0**

**3 45 '2020/12/04' 109 175 282.4**

**4 45 '2020/12/05' 117 148 406.0**

**5 60 '2020/12/06' 102 127 300.0**

**6 60 '2020/12/07' 110 136 374.0**

**7 450 '2020/12/08' 104 134 253.3**

**8 30 '2020/12/09' 109 133 195.1**

**9 60 '2020/12/10' 98 124 269.0**

**10 60 '2020/12/11' 103 147 329.3**

**11 60 '2020/12/12' 100 120 250.7**

**12 60 '2020/12/12' 100 120 250.7**

**13 60 '2020/12/13' 106 128 345.3**

**14 60 '2020/12/14' 104 132 379.3**

**15 60 '2020/12/15' 98 123 275.0**

**16 60 '2020/12/16' 98 120 215.2**

**17 60 '2020/12/17' 100 120 300.0**

**18 45 '2020/12/18' 90 112 NaN**

**19 60 '2020/12/19' 103 123 323.0**

**20 45 '2020/12/20' 97 125 243.0**

**21 60 '2020/12/21' 108 131 364.2**

**22 45 NaN 100 119 282.0**

**23 60 '2020/12/23' 130 101 300.0**

**24 45 '2020/12/24' 105 132 246.0**

**25 60 '2020/12/25' 102 126 334.5**

**26 60 20201226 100 120 250.0**

**27 60 '2020/12/27' 92 118 241.0**

**28 60 '2020/12/28' 103 132 NaN**

**29 60 '2020/12/29' 100 132 280.0**

**30 60 '2020/12/30' 102 129 380.3**

**31 60 '2020/12/31' 92 115 243.0**

**How can we fix wrong values, like the one for "Duration" in row 7?**

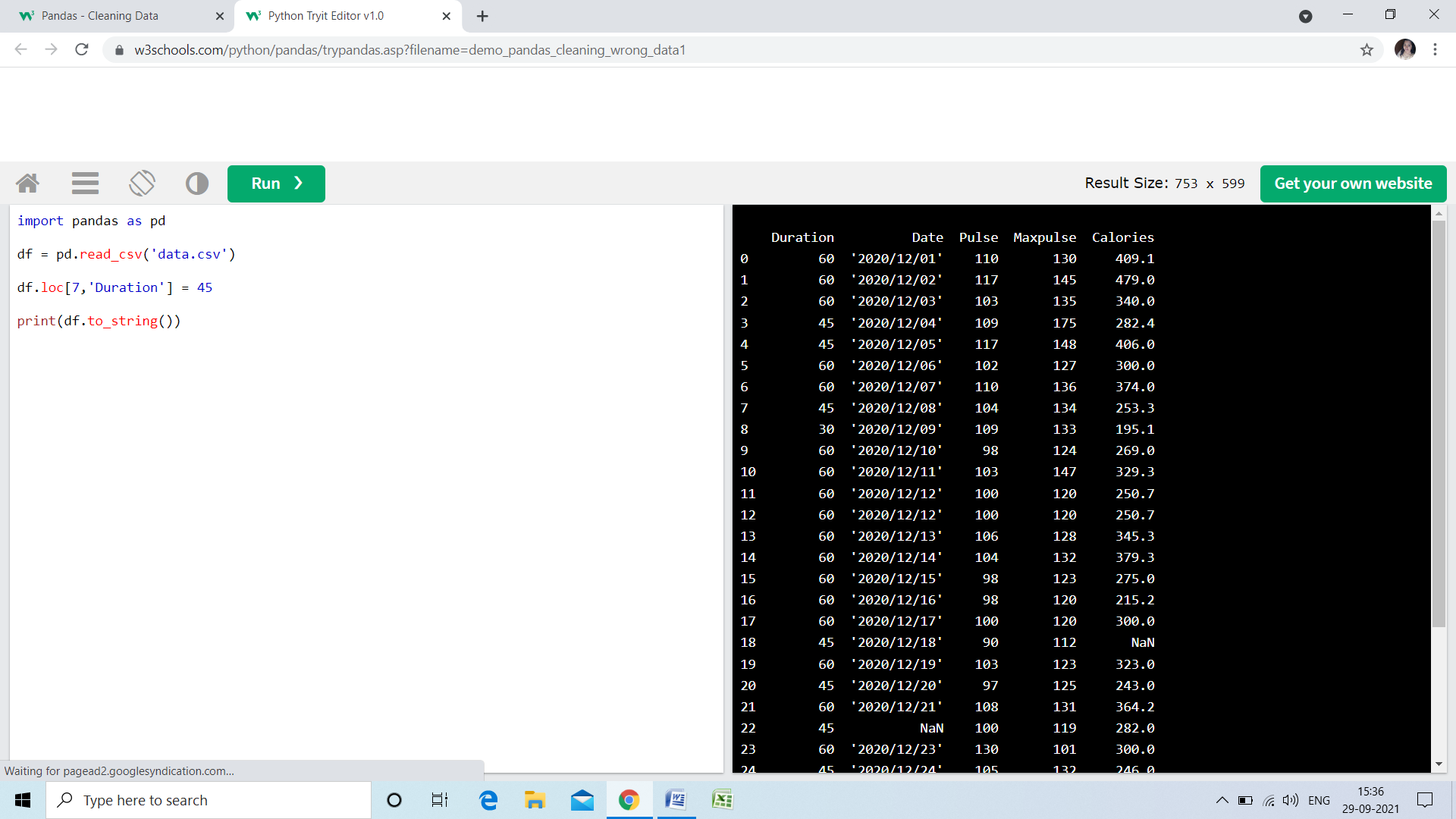
**Replacing Values**

* **One way to fix wrong values is to replace them with something else.**
* **In our example, it is most likely a typo, and the value should be "45" instead of "450", and we could just insert "45" in row 7:**

**Example**

**Set "Duration" = 45 in row 7:**

**df.loc[7, 'Duration'] = 45**

****

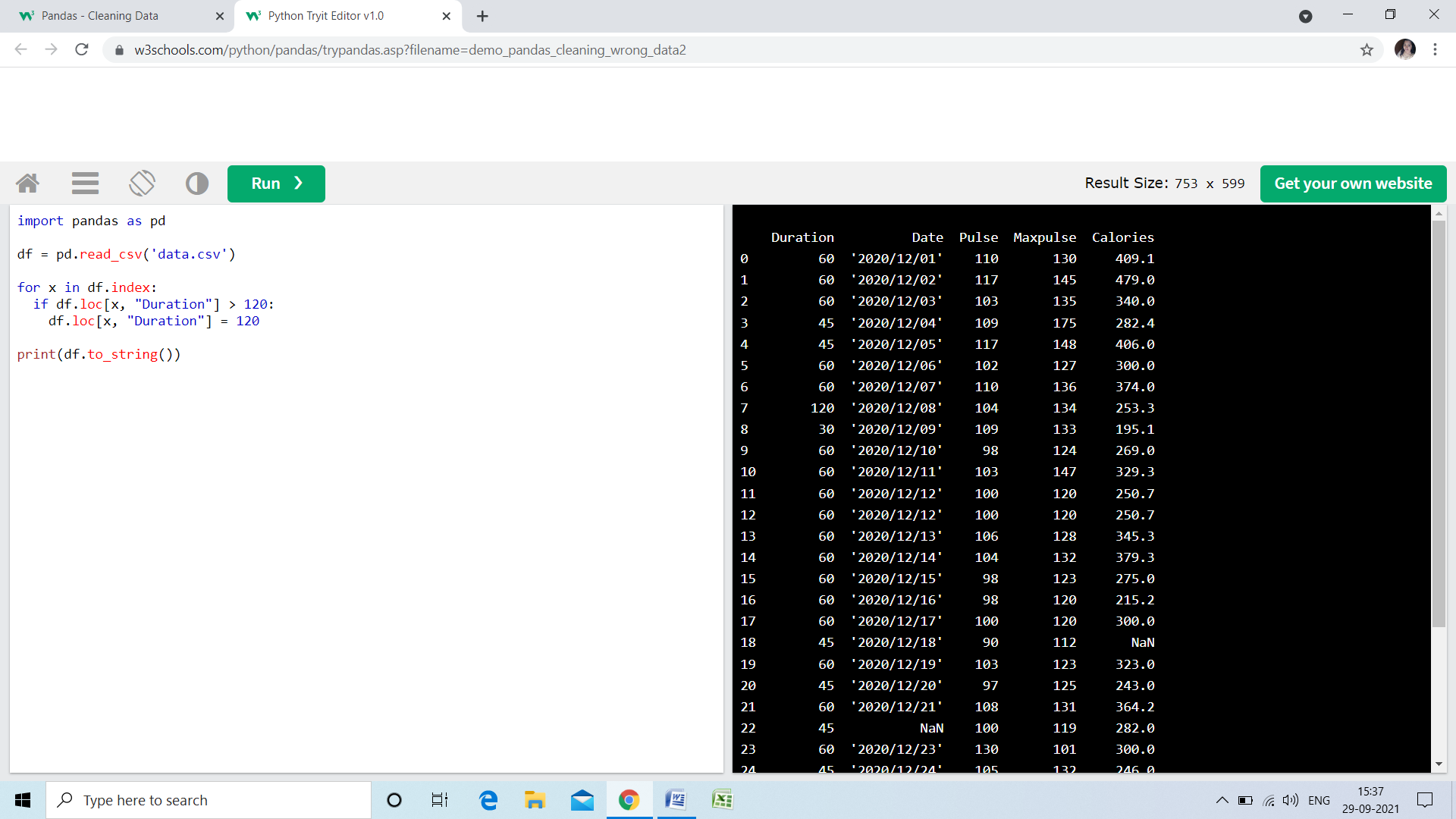
* **For small data sets you might be able to replace the wrong data one by one, but not for big data sets.**
* **To replace wrong data for larger data sets you can create some rules, e.g. set some boundaries for legal values, and replace any values that are outside of the boundaries.**

**Example**

**Loop through all values in the "Duration" column.**

**If the value is higher than 120, set it to 120:**

**for x in df.index:  
  if df.loc[x, "Duration"] > 120:  
    df.loc[x, "Duration"] = 120**

****

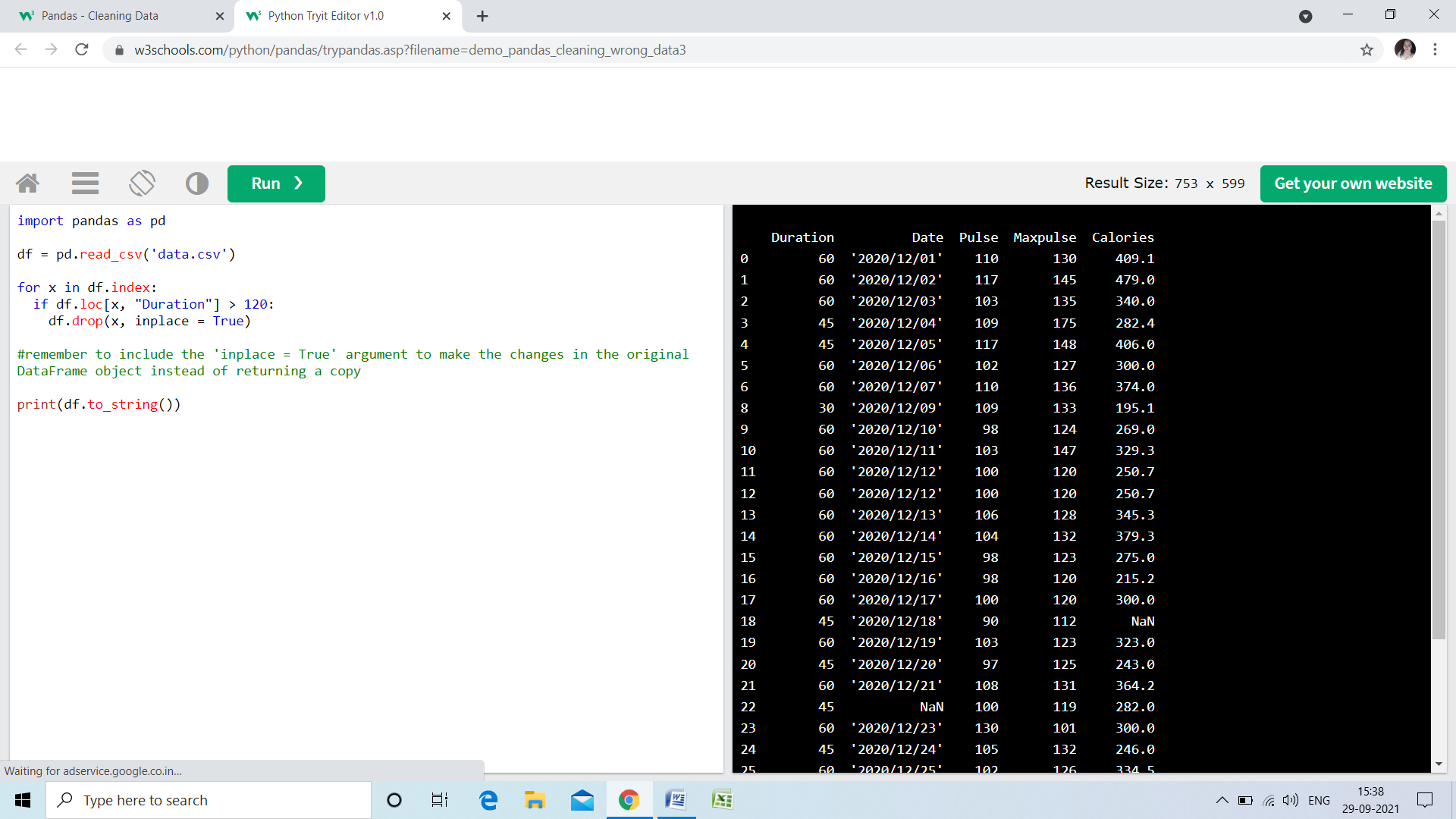
**Removing Rows**

* **Another way of handling wrong data is to remove the rows that contains wrong data.**
* **This way you do not have to find out what to replace them with, and there is a good chance you do not need them to do your analyses.**

**Example**

**Delete rows where "Duration" is higher than 120:**

**for x in df.index:  
  if df.loc[x, "Duration"] > 120:  
    df.drop(x, inplace = True)**

****

**Discovering Duplicates**

**Duplicate rows are rows that have been registered more than one time.**

**Duration Date Pulse Maxpulse Calories**

**0 60 '2020/12/01' 110 130 409.1**

**1 60 '2020/12/02' 117 145 479.0**

**2 60 '2020/12/03' 103 135 340.0**

**3 45 '2020/12/04' 109 175 282.4**

**4 45 '2020/12/05' 117 148 406.0**

**5 60 '2020/12/06' 102 127 300.0**

**6 60 '2020/12/07' 110 136 374.0**

**7 450 '2020/12/08' 104 134 253.3**

**8 30 '2020/12/09' 109 133 195.1**

**9 60 '2020/12/10' 98 124 269.0**

**10 60 '2020/12/11' 103 147 329.3**

**11 60 '2020/12/12' 100 120 250.7**

**12 60 '2020/12/12' 100 120 250.7**

**13 60 '2020/12/13' 106 128 345.3**

**14 60 '2020/12/14' 104 132 379.3**

**15 60 '2020/12/15' 98 123 275.0**

**16 60 '2020/12/16' 98 120 215.2**

**17 60 '2020/12/17' 100 120 300.0**

**18 45 '2020/12/18' 90 112 NaN**

**19 60 '2020/12/19' 103 123 323.0**

**20 45 '2020/12/20' 97 125 243.0**

**21 60 '2020/12/21' 108 131 364.2**

**22 45 NaN 100 119 282.0**

**23 60 '2020/12/23' 130 101 300.0**

**24 45 '2020/12/24' 105 132 246.0**

**25 60 '2020/12/25' 102 126 334.5**

**26 60 20201226 100 120 250.0**

**27 60 '2020/12/27' 92 118 241.0**

**28 60 '2020/12/28' 103 132 NaN**

**29 60 '2020/12/29' 100 132 280.0**

**30 60 '2020/12/30' 102 129 380.3**

**31 60 '2020/12/31' 92 115 243.0**

* **By taking a look at our test data set, we can assume that row 11 and 12 are duplicates.**
* **To discover duplicates, we can use the duplicated() method.**
* **The duplicated() method returns a Boolean values for each row:**

**Example**

**Returns True for every row that is a duplicate, othwerwise False:**

**print(df.duplicated())**

**Discovering Duplicates**

**Duplicate rows are rows that have been registered more than one time.**

**Duration Date Pulse Maxpulse Calories**

**0 60 '2020/12/01' 110 130 409.1**

**1 60 '2020/12/02' 117 145 479.0**

**2 60 '2020/12/03' 103 135 340.0**

**3 45 '2020/12/04' 109 175 282.4**

**4 45 '2020/12/05' 117 148 406.0**

**5 60 '2020/12/06' 102 127 300.0**

**6 60 '2020/12/07' 110 136 374.0**

**7 450 '2020/12/08' 104 134 253.3**

**8 30 '2020/12/09' 109 133 195.1**

**9 60 '2020/12/10' 98 124 269.0**

**10 60 '2020/12/11' 103 147 329.3**

**11 60 '2020/12/12' 100 120 250.7**

**12 60 '2020/12/12' 100 120 250.7**

**13 60 '2020/12/13' 106 128 345.3**

**14 60 '2020/12/14' 104 132 379.3**

**15 60 '2020/12/15' 98 123 275.0**

**16 60 '2020/12/16' 98 120 215.2**

**17 60 '2020/12/17' 100 120 300.0**

**18 45 '2020/12/18' 90 112 NaN**

**19 60 '2020/12/19' 103 123 323.0**

**20 45 '2020/12/20' 97 125 243.0**

**21 60 '2020/12/21' 108 131 364.2**

**22 45 NaN 100 119 282.0**

**23 60 '2020/12/23' 130 101 300.0**

**24 45 '2020/12/24' 105 132 246.0**

**25 60 '2020/12/25' 102 126 334.5**

**26 60 20201226 100 120 250.0**

**27 60 '2020/12/27' 92 118 241.0**

**28 60 '2020/12/28' 103 132 NaN**

**29 60 '2020/12/29' 100 132 280.0**

**30 60 '2020/12/30' 102 129 380.3**

**31 60 '2020/12/31' 92 115 243.0**

* **By taking a look at our test data set, we can assume that row 11 and 12 are duplicates.**
* **To discover duplicates, we can use the duplicated() method.**
* **The duplicated() method returns a Boolean values for each row:**

**Example**

**Returns True for every row that is a duplicate, othwerwise False:**

**print(df.duplicated())**

**Discovering Duplicates**

**Duplicate rows are rows that have been registered more than one time.**

**Duration Date Pulse Maxpulse Calories**

**0 60 '2020/12/01' 110 130 409.1**

**1 60 '2020/12/02' 117 145 479.0**

**2 60 '2020/12/03' 103 135 340.0**

**3 45 '2020/12/04' 109 175 282.4**

**4 45 '2020/12/05' 117 148 406.0**

**5 60 '2020/12/06' 102 127 300.0**

**6 60 '2020/12/07' 110 136 374.0**

**7 450 '2020/12/08' 104 134 253.3**

**8 30 '2020/12/09' 109 133 195.1**

**9 60 '2020/12/10' 98 124 269.0**

**10 60 '2020/12/11' 103 147 329.3**

**11 60 '2020/12/12' 100 120 250.7**

**12 60 '2020/12/12' 100 120 250.7**

**13 60 '2020/12/13' 106 128 345.3**

**14 60 '2020/12/14' 104 132 379.3**

**15 60 '2020/12/15' 98 123 275.0**

**16 60 '2020/12/16' 98 120 215.2**

**17 60 '2020/12/17' 100 120 300.0**

**18 45 '2020/12/18' 90 112 NaN**

**19 60 '2020/12/19' 103 123 323.0**

**20 45 '2020/12/20' 97 125 243.0**

**21 60 '2020/12/21' 108 131 364.2**

**22 45 NaN 100 119 282.0**

**23 60 '2020/12/23' 130 101 300.0**

**24 45 '2020/12/24' 105 132 246.0**

**25 60 '2020/12/25' 102 126 334.5**

**26 60 20201226 100 120 250.0**

**27 60 '2020/12/27' 92 118 241.0**

**28 60 '2020/12/28' 103 132 NaN**

**29 60 '2020/12/29' 100 132 280.0**

**30 60 '2020/12/30' 102 129 380.3**

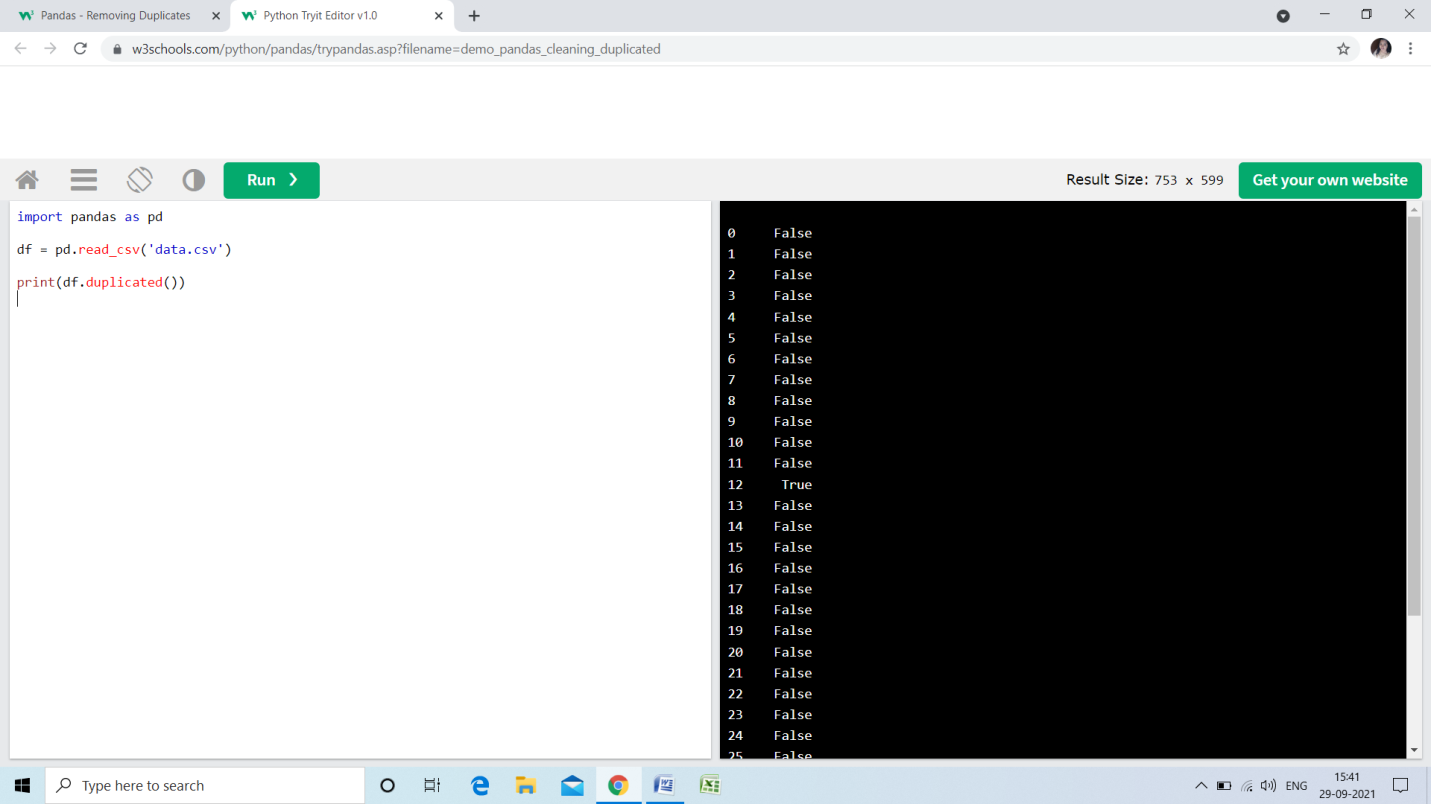
**31 60 '2020/12/31' 92 115 243.0**

* **By taking a look at our test data set, we can assume that row 11 and 12 are duplicates.**
* **To discover duplicates, we can use the duplicated() method.**
* **The duplicated() method returns a Boolean values for each row:**

**Example**

**Returns True for every row that is a duplicate, othwerwise False:**

**print(df.duplicated())**

****

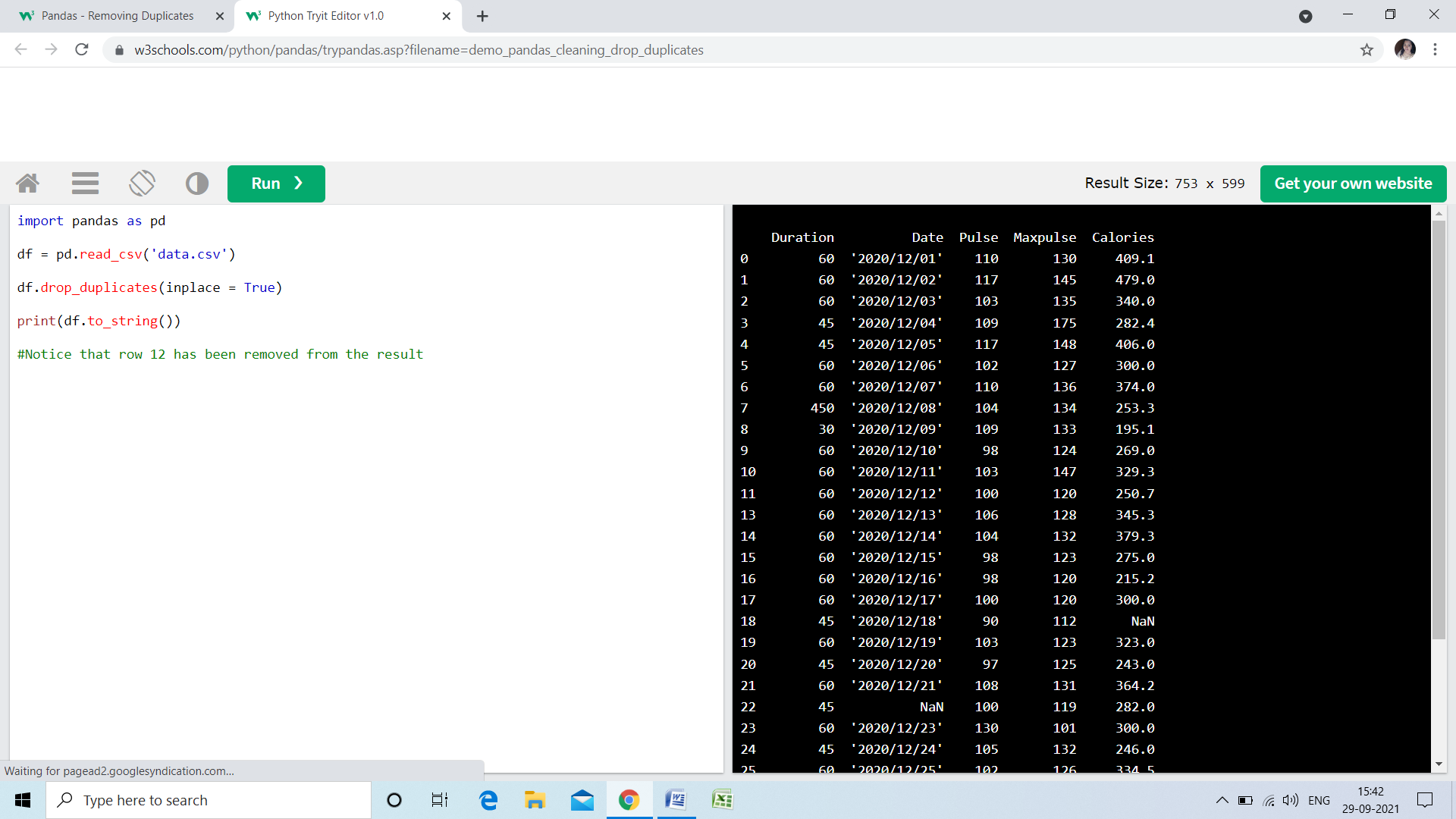
**Removing Duplicates**

**To remove duplicates, use the drop\_duplicates() method.**

**Example**

**Remove all duplicates:**

**df.drop\_duplicates(inplace = True)**

****

**Python Random module**

**The Python random module functions depend on a pseudo-random number generator function random(), which generates the float number between 0.0 and 1.0.**

**Different types of functions used in a random module**

**random.random()**

**This function generates a random float number between 0.0 and 1.0.**

**random.randint()**

**This function returns a random integer between the specified integers.**

**random.choice()**

**This function returns a randomly selected element from a non-empty sequence.**

**Example**

**import random**

**# We are using the choice() function to generate a random number from**

**# the given list of numbers.**

**print ("The random number from list is : ",end="")**

**print (random.choice([50, 41, 84, 40, 31]))**

**Output:**

**The random number from list is : 84**

**random.shuffle()**

**This function randomly reorders the elements in the list.**

**random.randrange(beg,end,step)**

**This function is used to generate a number within the range specified in its argument. It accepts three arguments, beginning number, last number, and step, which is used to skip a number in the range. Consider the following example.**

**# We are using randrange() function to generate in range from 100**

**# to 500. The last parameter 10 is step size to skip**

**# ten numbers when selecting.**

**import random**

**print ("A random number from range is : ",end="")**

**print (random.randrange(100, 500, 10))**

**Output:**

**A random number from range is : 290**

**random.seed()**

**This function is used to apply on the particular random number with the seed argument. It returns the mapper value. Consider the following example.**

**import random**

**# using random() to generate a random number**

**# between 0 and 1**

**print("The random number between 0 and 1 is : ", end="")**

**print(random.random())**

**# using seed() to seed a random number**

**random.seed(4)**

**Output:**

**The random number between 0 and 1 is : 0.4405576668981033**

**The random number between 0 and 1 is : 0.4405576668981033**

**Pickle Module of Python**

* A developer may sometimes want to send some complex object commands through the network and save the internal state of their objects to the disk or database for using it later.
* To achieve this, the developer can use the serialization process, which is supported by the standard library, cause of Python's Pickle Module.

**Serialization in Python**

The process of serializing is to convert the data structure into a linear form, which can be stored or transmitted through the network.

* In [Python](https://www.javatpoint.com/python-tutorial), the **serialization** allows the developer to convert the complex object structure into a stream of bytes that can be saved in the disk or can send through the network. The developer can refer to this process as **marshalling.**
* Whereas, **Deserialization** is the reverse process of serialization in which the user takes the stream of bytes and transforms it into the data structure. This process can be referred to as unmarshalling.

In Python there are three modules in the standard library that allows the developer to serialize and deserialize the objects:

1. The pickle module
2. The marshal module
3. The json module

The pickle module of Python is another method of serializing and deserializing the objects in Python. The object is serialized in the binary format, whose result is not readable by humans. Although, it is faster than the others, and it can work with many other python types, including the developer's custom -defined objects.

**Inside The pickle Module**

The pickle module of python contains the four methods:

1. **dump( obj, file, protocol = None, \* , fix\_imports = True, buffer\_callback = None)**
2. **dumps( obj, protocol = None, \* , fix\_imports = True, buffer\_callback = None)**
3. **load( file, \* , fix\_imports = True, encoding = " ASCII ", errors = "strict ", buffers = None)**
4. **loads( bytes\_object, \* , fix\_imports = True, encoding = " ASCII ", errors = " strict ", buffers = None)**

The first two methods are used for the pickling process, and the next two methods are used for the unpickling process.

The difference between dump() and dumps() is that dump() creates the file which contains the serialization results, and the dumps() returns the string.

For differentiation dumps() from the dump(), the developer can remember that in the dumps() function, ' s' stands for the string.

The same concept can be applied to the load() and loads() function. The load() function is used for reading the file for the unpickling process, and the loads() function operates on the string.

Suppose the user has a custom -defined class named **forexample\_class** with many different attributes, and each one of them is of different types:

* the\_number
* the\_string
* the\_list
* the\_dictionary
* the\_tuple

The example below explains how the user can instantiate the class and pickle the instance to get the plain string. After pickling the class, the user can modify the value of its attributes without affecting the pickled string. User can afterward unpickle the string which was pickled earlier in another variable, and restore the copy of the pickled class.

**For example:**

# pickle.py

**import pickle**

**class forexample\_class:**

**the\_number = 25**

**the\_string = " hello"**

**the\_list = [ 1, 2, 3 ]**

**the\_dict = { " first ": " a ", " second ": 2, " third ": [ 1, 2, 3 ] }**

**the\_tuple = ( 22, 23 )**

**user\_object = forexample\_class()**

user\_pickled\_object = **pickle.dumps**( user\_object )  # here, user is Pickling the object

print( f" This is user's pickled object: \n { user\_pickled\_object } \n " )

user\_object.the\_dict = None

user\_unpickled\_object = **pickle.loads(** user\_pickled\_object )  # here, user is Unpickling the object

print(

    f" This is the\_dict of the unpickled object: \n { user\_unpickled\_object.the\_dict } \n " )

**Output:**

This is user's pickled object:

b' \x80 \x04 \x95$ \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x8c \x08\_\_main\_\_ \x94 \x8c \x10forexample\_class \x94 \x93 \x94) \x81 \x94. '

This is the\_dict of the unpickled object:

{' first ': ' a ', ' second ': 2, ' third ': [ 1, 2, 3 ] }

**Example -**

**Explanation**

Here, the process of pickling has ended correctly, and it stores the user's whole instance in the string: b' \x80 \x04 \x95$ \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x8c \x08\_\_main\_\_ \x94 \x8c \x10forexample\_class \x94 \x93 \x94) \x81 \x94. 'After completing the process of pickling, the user can change their original objects making the\_dict attribute equals to None.

Now, the user can process for unpickling the string into the utterly new instance. When the user gets a deep copy of their original object structure from the time when the process of pickling the object began.

**Python Random module**

**The Python random module functions depend on a pseudo-random number generator function random(), which generates the float number between 0.0 and 1.0.**

**Different types of functions used in a random module**

**random.random()**

**This function generates a random float number between 0.0 and 1.0.**

**random.randint()**

**This function returns a random integer between the specified integers.**

**random.choice()**

**This function returns a randomly selected element from a non-empty sequence.**

**Example**

**import random**

**# We are using the choice() function to generate a random number from**

**# the given list of numbers.**

**print ("The random number from list is : ",end="")**

**print (random.choice([50, 41, 84, 40, 31]))**

**Output:**

**The random number from list is : 84**

**random.shuffle()**

**This function randomly reorders the elements in the list.**

**random.randrange(beg,end,step)**

**This function is used to generate a number within the range specified in its argument. It accepts three arguments, beginning number, last number, and step, which is used to skip a number in the range. Consider the following example.**

**# We are using randrange() function to generate in range from 100**

**# to 500. The last parameter 10 is step size to skip**

**# ten numbers when selecting.**

**import random**

**print ("A random number from range is : ",end="")**

**print (random.randrange(100, 500, 10))**

**Output:**

**A random number from range is : 290**

**random.seed()**

**This function is used to apply on the particular random number with the seed argument. It returns the mapper value. Consider the following example.**

**import random**

**# using random() to generate a random number**

**# between 0 and 1**

**print("The random number between 0 and 1 is : ", end="")**

**print(random.random())**

**# using seed() to seed a random number**

**random.seed(4)**

**Output:**

**The random number between 0 and 1 is : 0.4405576668981033**

**The random number between 0 and 1 is : 0.4405576668981033**

**MySQL**

**pip install mysql-connector-python**

**Test MySQL Connector**

**import mysql.connector**

## Create Connection

**import mysql.connector**

**mydb = mysql.connector.connect(**

**host="localhost",**

**user="root",**

**password=""**

**)**

**print(mydb)**

**output:**

**<mysql.connector.connection.MySQLConnection object ar 0x016645F0>**

**Creating a Database**

**To create a database in MySQL, use the "CREATE DATABASE" statement:**

**import mysql.connector**

**mydb = mysql.connector.connect(**

**host="localhost",**

**user="root",**

**password=""**

**)**

**mycursor = mydb.cursor()**

**mycursor.execute("CREATE DATABASE mydatabase")**

**Check if Database Exists**

**import mysql.connector**

**mydb = mysql.connector.connect(**

**host="localhost",**

**user="root",**

**password=""**

**)**

**mycursor = mydb.cursor()**

**mycursor.execute("SHOW DATABASES")**

**for x in mycursor:**

**print(x)**

**Python MySQL Create Table**

**To create a table in MySQL, use the "CREATE TABLE" statement. Make sure you define the name of the database when you create the connection**

**import mysql.connector**

**mydb = mysql.connector.connect(**

**host="localhost",**

**user="root",**

**password="",**

**database="mydatabase"**

**)**

**mycursor = mydb.cursor()**

**mycursor.execute("CREATE TABLE customers (name VARCHAR(255), address VARCHAR(255))")**

**Check if Table Exists**

**import mysql.connector**

**mydb = mysql.connector.connect(**

**host="localhost",**

**user="root",**

**password="",**

**database="mydatabase"**

**)**

**mycursor = mydb.cursor()**

**mycursor.execute("SHOW TABLES")**

**for x in mycursor:**

**print(x)**

**Create primary key when creating the table:**

**import mysql.connector**

**mydb = mysql.connector.connect(**

**host="localhost",**

**user="root",**

**password="",**

**database="mydatabase"**

**)**

**mycursor = mydb.cursor()**

**mycursor.execute("CREATE TABLE customers (id INT AUTO\_INCREMENT PRIMARY KEY, name VARCHAR(255), address VARCHAR(255))")**

**Python MySQL Insert Into Table**

**import mysql.connector**

**mydb = mysql.connector.connect(**

**host="localhost",**

**user="root",**

**password="",**

**database="mydatabase"**

**)**

**mycursor = mydb.cursor()**

**sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"**

**val = ("John", "Highway 21")**

**mycursor.execute(sql, val)**

**mydb.commit()**

**print(mycursor.rowcount, "record inserted.")**

**Insert Multiple Rows**

**import mysql.connector**

**mydb = mysql.connector.connect(**

**host="localhost",**

**user="root",**

**password="",**

**database="mydatabase"**

**)**

**mycursor = mydb.cursor()**

**sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"**

**val = [**

**('Peter', 'Lowstreet 4'),**

**('Amy', 'Apple st 652'),**

**('Hannah', 'Mountain 21'),**

**('Michael', 'Valley 345'),**

**('Sandy', 'Ocean blvd 2'),**

**('Betty', 'Green Grass 1'),**

**('Richard', 'Sky st 331'),**

**('Susan', 'One way 98'),**

**('Vicky', 'Yellow Garden 2'),**

**('Ben', 'Park Lane 38'),**

**('William', 'Central st 954'),**

**('Chuck', 'Main Road 989'),**

**('Viola', 'Sideway 1633')**

**]**

**mycursor.executemany(sql, val)**

**mydb.commit()**

**print(mycursor.rowcount, "record was inserted.")**

**Python MySQL Select From**

**To select from a table in MySQL, use the "SELECT" statement:**

**import mysql.connector**

**mydb = mysql.connector.connect(**

**host="localhost",**

**user="myusername",**

**password="mypassword",**

**database="mydatabase"**

**)**

**mycursor = mydb.cursor()**

**mycursor.execute("SELECT \* FROM customers")**

**myresult = mycursor.fetchall()**

**for x in myresult:**

**print(x)**

**Selecting Columns**

**import mysql.connector**

**mydb = mysql.connector.connect(**

**host="localhost",**

**user="myusername",**

**password="mypassword",**

**database="mydatabase"**

**)**

**mycursor = mydb.cursor()**

**mycursor.execute("SELECT name, address FROM customers")**

**myresult = mycursor.fetchall()**

**for x in myresult:**

**print(x)**

**Using the fetchone() Method**

**If you are only interested in one row, you can use the fetchone() method. The fetchone() method will return the first row of the result:**

**import mysql.connector**

**mydb = mysql.connector.connect(**

**host="localhost",**

**user="myusername",**

**password="mypassword",**

**database="mydatabase"**

**)**

**mycursor = mydb.cursor()**

**mycursor.execute("SELECT \* FROM customers")**

**myresult = mycursor.fetchone()**

**print(myresult)**

**Python MySQL Where**

**import mysql.connector**

**mydb = mysql.connector.connect(**

**host="localhost",**

**user="myusername",**

**password="mypassword",**

**database="mydatabase"**

**)**

**mycursor = mydb.cursor()**

**sql = "SELECT \* FROM customers WHERE address = 'Park Lane 38'"**

**mycursor.execute(sql)**

**myresult = mycursor.fetchall()**

**for x in myresult:**

**print(x)**

**Wildcard Characters**

**You can also select the records that starts, includes, or ends with a given letter or phrase. Use the %  to represent wildcard characters:**

**import mysql.connector**

**mydb = mysql.connector.connect(**

**host="localhost",**

**user="myusername",**

**password="mypassword",**

**database="mydatabase"**

**)**

**mycursor = mydb.cursor()**

**sql = "SELECT \* FROM customers WHERE address Like '%way%'"**

**mycursor.execute(sql)**

**myresult = mycursor.fetchall()**

**for x in myresult:**

**print(x)**

**Python MySQL Order By**

* **Use the ORDER BY statement to sort the result in ascending or descending order.**
* **The ORDER BY keyword sorts the result ascending by default. To sort the result in descending order, use the DESC keyword.**

**import mysql.connector**

**mydb = mysql.connector.connect(**

**host="localhost",**

**user="myusername",**

**password="mypassword",**

**database="mydatabase"**

**)**

**mycursor = mydb.cursor()**

**sql = "SELECT \* FROM customers ORDER BY name"**

**mycursor.execute(sql)**

**myresult = mycursor.fetchall()**

**for x in myresult:**

**print(x)**

**ORDER BY DESC**

**Use the DESC keyword to sort the result in a descending order.**

**import mysql.connector**

**mydb = mysql.connector.connect(**

**host="localhost",**

**user="myusername",**

**password="mypassword",**

**database="mydatabase"**

**)**

**mycursor = mydb.cursor()**

**sql = "SELECT \* FROM customers ORDER BY name DESC"**

**mycursor.execute(sql)**

**myresult = mycursor.fetchall()**

**for x in myresult:**

**print(x)**

**Python MySQL Delete From By**

**You can delete records from an existing table by using the "DELETE FROM" statement:**

**import mysql.connector**

**mydb = mysql.connector.connect(**

**host="localhost",**

**user="myusername",**

**password="mypassword",**

**database="mydatabase"**

**)**

**mycursor = mydb.cursor()**

**sql = "DELETE FROM customers WHERE address = 'Mountain 21'"**

**mycursor.execute(sql)**

**mydb.commit()**

**print(mycursor.rowcount, "record(s) deleted")**

**Python MySQL Drop Table**

**import mysql.connector**

**mydb = mysql.connector.connect(**

**host="localhost",**

**user="myusername",**

**password="mypassword",**

**database="mydatabase"**

**)**

**mycursor = mydb.cursor()**

**sql = "DROP TABLE customers"**

**mycursor.execute(sql)**

**Python MySQL Update Table**

**import mysql.connector**

**mydb = mysql.connector.connect(**

**host="localhost",**

**user="myusername",**

**password="mypassword",**

**database="mydatabase"**

**)**

**mycursor = mydb.cursor()**

**sql = "UPDATE customers SET address = 'Canyon 123' WHERE address = 'Valley 345'"**

**mycursor.execute(sql)**

**mydb.commit()**

**print(mycursor.rowcount, "record(s) affected")**