Basic operations in Python:

Arithmetic Operations

Addition

```
a=4
b=7
a+b
```
11

Subtraction

```
a-b
```
-3

Multiplication

```
a*b
```
28

Division

```
a/b
```
0.5714285714285714

Square

```
a**2
```
16

Square Root

```
a*0.5
```
2.0

Loops:

For Loop

When the number of iterations required is known i.e. n, the 'for' is used.

```
n=5 #n is the variable with assigned value 10
for i in range(0,n): #for every value in range 0 to n
print('Hello World') #print
```

Hello World
Hello World
Hello World
Hello World
Hello World

While Loop

When the number of iterations required is unknown i.e. n, the 'while' is used.

```
i=0 # value of variable i is 0
while i<5: #till i is smaller than 5
print('Hello World') # print
i=i+1 # increase value by 1 after every print
```

Hello World
Hello World
Hello World
Hello World
Hello World

Conditional Statements:
IF Statement

This statement check the condition provided and if the condition is True, then the program moves ahead with defined steps

```
x=300
if x>200:
print('Hey, x is greater than 200!')
```

Hey, x is greater than 200!

IF-Else Statement

This statement is an extension to IF statement. The program moves to Else statement and performs the alternative steps defined in case 'IF' condition is not met.

```
x=100
if x>200:
print('Hey, x is greater than 200!')
else: print('Hey, x is smaller than 200!')
```

Hey, x is smaller than 200!

IF-ELIF Statement

This statement is an extension to IF-Else statement. The program moves to ELIF statement and performs the alternative steps defined in case 'IF' condition is not met and if the else if 'ELIF' in python is not met, then the program moves to another ELIF or Else statement.
Syntax along with example In this example else would be executed only when x=y

```
x=100
y=150
if x>y:
print('Hey, x is greater than y!')
elif x<y:
print('Hey, y is greater than x!')
else: print('Hey, x is equal to y!')
```

Hey, y is greater than x!

User Defined Functions:
User-defined functions are very helpful in automating repetitive tasks like selecting odd numbers out of a series or converting a series of dates to timestamps

```
#A function is defined using 'def' followed by function name and arguments the function takes in

def addition(x,y):

return (x+y) #in a function, return is used most preferably

#this is a function which returns addition of two variables passed into it.
#calling a function

a=1
b=2
addition(a,b)
```

3

Data Types:

Following are most used built-in data types in Python:

Text Type: str
Numeric Types: int, float, complex
Sequence Types: list, tuple, range

Mapping Type: dict
Set Types: set
Boolean Type: bool

Numeric data type consists of numbers. These numbers can be whole numbers, decimal numbers or complex numbers.
Whole numbers are stored in Integer (Int) type.
Decimal numbers are stored in Float (Float64) type.
Complex numbers are stored in Complex type.
Let's look at some examples:

```
Integer = 7
type(Integer)
int

Float = 2.356
type(Float)
float

Complex = 3.908J
type(Complex)
complex
```

String Data type usually is used to store text. The data to be stored in this data type is enclosed between single ('') or double ("") quotes. Recall that you printed your name in the previous Notebook. That was string data type. Let's look at an example.

```
My_Name = "My name is Jupyter!"
My_Name
'My name is Jupyter!'
type(My_Name)
str
```

The boolean data type has just two values, i.e., True or False.

```
# Let's look at examples.

x = True
y = False
type(x)
bool
type(y)
bool
```

List is a collection which is ordered and changeable. Allows duplicate members.

```
my_list = [1,2,3,4,5]
my_list
[1, 2, 3, 4, 5]
type(my_list)
list
my_list[2]
3
```

Tuple is a collection which is ordered and unchangeable. It allows duplicate members.

```
my_tuple = ("apple", "banana", "cherry", 'cherry')
my_tuple
('apple', 'banana', 'cherry', 'cherry')
type(my_tuple)
tuple
my_tuple[2]
'cherry'
```

Note: Tuple items can be of any data type

```
my_tuple1 = ("apple", "banana", "cherry", 'cherry', 1,2,True)
my_tuple1
('apple', 'banana', 'cherry', 'cherry', 1, 2, True)
type(my_tuple1)
tuple
```

Set is a collection which is unordered and unindexed. No duplicate members.

```
my_set = {1,2,3,4,5,5}
my_set
{1, 2, 3, 4, 5}
type(my_set)
set
length = len(my_set)
length
5
my_set.remove(4)
my_set
{1, 2, 3, 5}
```

Dictionary is a collection which is unordered and changeable. No duplicate members.

```
my_dictry = {'A': 1, 'B': 2, 'C': 5}
my_dictry
{'A': 1, 'B': 2, 'C': 5}
type(my_dictry)
dict
my_dictry['B']
2
```

Pre-processing:

Note: "df" is a user defined name of the dataframe used below

df. **head()** returns the first 5 rows of the dataframe. To override the default, you may insert a value between the parentheses to change the number of rows returned. Example: **df**. **head**(10) will return 10 rows.

df. **tail()** returns the last 5 rows of the dataframe. To override the default, you may insert a value between the parentheses to change the number of rows returned. Example: **df**. **head**(10) will return 10 rows.

df.shape returns the dimensions of the dataframe i.e. number of rows and columns in a dataframe. It is an attribute.

df.info() method allows us to learn the shape of object types of our data in the dataframe.

df .describe() method gives us summary statistics for numerical columns in our dataframe.

Null Value Check

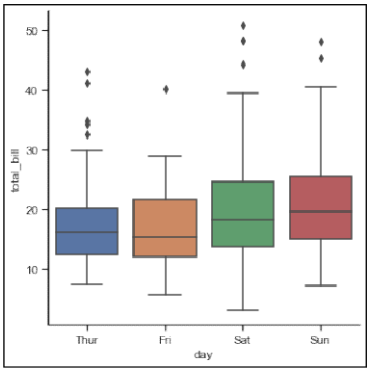
df.isnull() allows us to check Null or Missing values present in the dataframe.

dropna() function is used to remove rows and columns with Null/NaN values. By default, this function returns a new dataframe and the source dataframe remains unchanged.

Data Visualization:

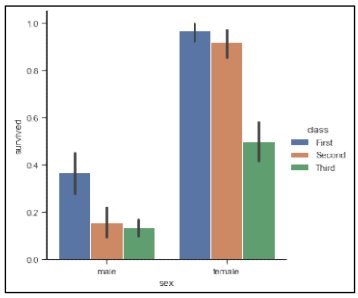
sns.boxplot()

The **seaborn boxplot** is a very basic plot used to visualize distributions. Thats very useful when you want to compare data between two groups. It is very useful to check outliers present in the data. An **outlier** is an observation that is numerically distant from the rest of the data. When reviewing a box plot, an outlier is defined as a data point that is located outside the whiskers of the box plot.



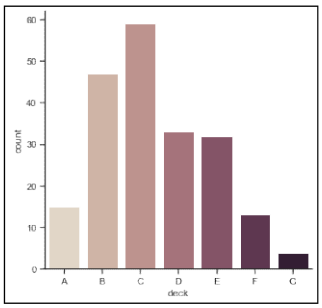
barplot()

A **bar plot** represents an estimate of central tendency for a numeric variable with the height of each rectangle



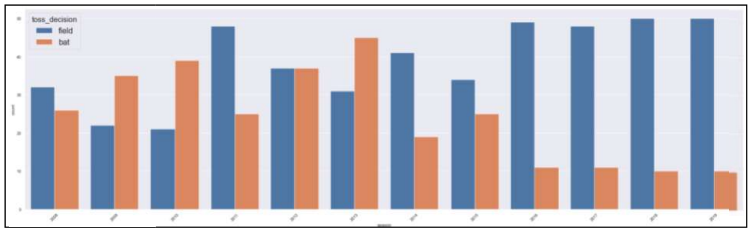
sns. **countplot**()

Count plot represents the count of each category using bars.



plt.subplots()

You can use subplots to represent multiple graphs in a single window.



pd.crosstab()

A **crosstab** is a table showing the relationship between two or more variables. Where the table only shows the relationship between two categorical variables, a crosstab is also known as a contingency table.

alive	no	yes
sex		
female	81	233
male	468	109