
# Python
## Cheat Sheet

Python 3 is a truly versatile programming language, loved both by web developers, data scientists and software engineers. And there are several good reasons for that!

- Python is open-source and has a great support community,
- Plus, extensive support libraries.
- Its data structures are user-friendly.

**Once you get a hang of it, your development speed and productivity will soar!**

# Table of Contents

# Python Basics: Getting Started

Most Windows and Mac computers come with Python pre-installed. You can check that via a Command Line search. The particular appeal of Python is that you can write a program in any text editor, save it in .py format and then run via a Command Line. But as you learn to write more complex code or venture into data science, you might want to switch to an IDE or IDLE.

## What is IDLE (Integrated Development and Learning)

IDLE (Integrated Development and Learning Environment) comes with every Python installation. Its advantage over other text editors is that it highlights important keywords (e.g. string functions), making it easier for you to interpret code.

Shell is the default mode of operation for Python IDLE. In essence, it's a simple loop that performs that following four steps:

- Reads the Python statement
- Evaluates the results of it
- Prints the result on the screen
- And then loops back to read the next statement.

Python shell is a great place to test various small code snippets.

# Main Python Data Types

Every value in Python is called an "**object**". And every object has a specific data type. The three most-used data types are as follows:

**Integers (int)** — an integer number to represent an object such as "number 3".

```
Integers                    -2, -1, 0, 1, 2, 3, 4, 5
```

**Floating-point numbers (float)** — use them to represent floating-point numbers.

```
Floating-point numbers      -1.25, -1.0, --0.5, 0.0, 0.5, 1.0, 1.25
```

**Strings** — codify a sequence of characters using a string. For example, the word "hello". In Python 3, strings are immutable. If you already defined one, you cannot change it later on.

While you can modify a string with commands such as **replace()** or **join()**, they will create a copy of a string and apply modification to it, rather than rewrite the original one.

```
Strings                     'yo', 'hey', 'Hello!', 'what's up!'
```

Plus, another three types worth mentioning are **list**s, **dict**ionaries, and **tuple**s. All of them are discussed in the next sections.

For now, let's focus on the **strings**.

# How to Create a String in Python

You can create a string in three ways using **single**, **double** or **triple** quotes. Here's an example of every option:

## Basic Python String

```
my_string = "Let's Learn Python!"
another_string = 'It may seem difficult first, but you can do it!'
a_long_string = '''Yes, you can even master multi-line strings
 that cover more than one line
 with some practice'''
```

**IMP!** Whichever option you choose, you should stick to it and use it consistently within your program.

As the next step, you can use the **print()** function to output your string in the console window. This lets you review your code and ensure that all functions well.

Here's a snippet for that:

```
print("Let's print out a string!")
```

## String Concatenation

The next thing you can master is **concatenation** — a way to add two strings together using the "+" operator. Here's how it's done:

```
string_one = "I'm reading "
string_two = "a new great book!"
string_three = string_one + string_two
```

**Note:** You can't apply + operator to two different data types e.g. string + integer. If you try to do that, you'll get the following Python error:

```
TypeError: Can't convert 'int' object to str implicitly
```

## String Replication

As the name implies, this command lets you repeat the same string several times. This is done using * operator. Mind that this operator acts as a replicator only with string data types. When applied to numbers, it acts as a multiplier.

String replication example:

```
'Alice' * 5 'AliceAliceAliceAliceAlice'
```

And with print ()

```
print("Alice" * 5)
```

And your output will be Alice written five times in a row.

# Math Operators

For reference, here's a list of other math operations you can apply towards numbers:

| Operators | Operation | Example |
|---|---|---|
| ** | Exponent | 2 ** 3 = 8 |
| % | Modulus/Remainder | 22 % 8 = 6 |
| // | Integer division | 22 // 8 = 2 |
| / | Division | 22 / 8 = 2.75 |
| * | Multiplication | 3 * 3 = 9 |
| - | Subtraction | 5 – 2 = 3 |
| + | Addition | 2 + 2 = 4 |

# How to Store Strings in Variables

**Variables** in Python 3 are special symbols that assign a specific storage location to a value that's tied to it. In essence, variables are like special labels that you place on some value to know where it's stored.

Strings incorporate data. So you can "pack" them inside a variable. Doing so makes it easier to work with complex Python programs.

Here's how you can store a string inside a variable.

```
my_str = "Hello World"
```

Let's break it down a bit further:

- my_str is the variable name.
- = is the assignment operator.
- "Just a random string" is a value you tie to the variable name.

Now when you print this out, you receive the string output.

```
print(my_str)
```

```
= Hello World
```

See? By using variables, you save yourself heaps of effort as you don't need to retype the complete string every time you want to use it.

# Built-in Functions in Python

You already know the most popular function in Python — **print()**. Now let's take a look at its equally popular cousins that are in-built in the platform.

## Input() Function

**input()** function is a simple way to prompt the user for some input (e.g. provide their name). All user input is stored as a string.

Here's a quick snippet to illustrate this:

```
name = input("Hi! What's your name? ")
print("Nice to meet you " + name + "!")

age = input("How old are you ")
print("So, you are already " + str(age) + " years old, "
+ name + "!")
```

When you run this short program, the results will look like this:

```
Hi! What's your name? "Jim"

Nice to meet you, Jim!

How old are you? 25

So, you are already 25 years old, Jim!
```

## len() Function

**len()** function helps you find the length of any string, list, tuple, dictionary, or another data type. It's a handy command to determine excessive values and trim them to optimize the performance of your program.

Here's an input function example for a string:

```
# testing len()
str1 = "Hope you are enjoying our tutorial!"
print("The length of the string is :", len(str1))
```

Output:

```
The length of the string is: 35
```

## filter()

Use the **Filter()** function to exclude items in an iterable object (lists, tuples, dictionaries, etc)

```
ages = [5, 12, 17, 18, 24, 32]

def myFunc(x):
  if x < 18:
    return False
  else:
    return True

adults = filter(myFunc, ages)

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

(Optional: The PDF version of the checklist can also include a full table of all the in-built functions).

# How to Define a Function

Apart from using in-built functions, Python 3 also allows you to define your own functions for your program.

To recap, a **function** is a block of coded instructions that perform a certain action. Once properly defined, a function can be reused throughout your program i.e. re-use the same code.

Here's a quick walkthrough explaining how to define a function in Python:

First, use **def** keyword followed by the function **name()**:. The parentheses can contain any parameters that your function should take (or stay empty).

```
def name():
```

Next, you'll need to add a second code line with a 4-space indent to specify what this function should do.

```
def name():
    print("What's your name?")
```

Now, you have to call this function to run the code.

```
name.py
def name():
    print("What's your name?")

name()
```

Now, let's take a look at a defined function with a parameter — an entity, specifying an argument that a function can accept.

```
def add_numbers(x, y, z):
    a = x + y
    b = x + z
    c = y + z
    print(a, b, c)

add_numbers(1, 2, 3)
```

In this case, you pass the number 1 in for the x parameter, 2 in for the y parameter, and 3 in for the z parameter. The program will that do the simple math of adding up the numbers:

Output:

```
a = 3
b = 4
c = 5
```

## How to Pass Keyword Arguments to a Function

A function can also accept keyword arguments. In this case, you can use parameters in random order as the Python interpreter will use the provided keywords to match the values to the parameters.

Here's a simple example of how you pass a keyword argument to a function.

```
# Define function with parameters
def product_info (product name, price):
    print("Product Name: " + product_name)
    print("Price: " + str(price))

# Call function with parameters assigned as above
product_info("White T-Shirt: ", 15)

# Call function with keyword arguments
product_info(productname="Jeans", price=45)
```

Output:

```
Product Name: White T-Shirt
Price: 15
Product Name: Jeans
Price: 45
```

# Lists

**Lists** are another cornerstone data type in Python used to specify an ordered sequence of elements. In short, they help you keep related data together and perform the same operations on several values at once. Unlike strings, lists are mutable (=changeable).

Each value inside a list is called an **item** and these are placed between square brackets.

## Example lists

```
my_list = [1, 2, 3]
my_list2 = [“a”, “b”, “c”]
my_list3 = [“4”, d, “book”, 5]
```

Alternatively, you can use **list()** function to do the same:

```
alpha_list = list((“1”, “2”, “3”))
print(alpha_list)
```

## How to Add Items to a List

You have two ways to add new items to existing lists.

The first one is using **append()** function:

```
beta_list = [“apple”, “banana”, “orange”]
beta_list.append(“grape”)
print(beta_list)
```

The second option is to **insert()** function to add an item at the specified index:

```
beta_list = [“apple”, “banana”, “orange”]
beta_list.insert(2,“grape”)
print(beta_list)
```

## How to Remove an Item from a List

Again, you have several ways to do so. First, you can use **remove()** function:

```
beta_list = [“apple”, “banana”, “orange”]
beta_list.remove(“apple”)
print(beta_list)
```

Secondly, you can use the **pop() function**. If no index is specified, it will remove the last item.

```
beta_list = [“apple”, “banana”, “orange”]
beta_list.pop()
print(beta_list)
```

The last option is to use **del keyword** to remove a specific item:

```
beta_list = [“apple”, “banana”, “orange”]
del beta_list [1]
print(beta_list)
```

P.S. You can also apply del towards the entire list to scrap it.

## Combine Two Lists

To mash up two lists use the + operator.

```
my_list = [1, 2, 3]
my_list2 = [“a”, “b”, “c”]
combo_list = my_list + my_list2
combo_list
[1, 2, 3, ‘a’, ‘b’, ‘c’]
```

## Create a Nested List

You can also create a list of your lists when you have plenty of them :)

```
my_nested_list = [my_list, my_list2]
my_nested_list
[[1, 2, 3], [‘a’, ‘b’, ‘c’]]
```

## Sort a List

Use the **sort()** function to organize all items in your list.

```
alpha_list = [34, 23, 67, 100, 88, 2]
alpha_list.sort()
alpha_list
[2, 23, 34, 67, 88, 100]
```

## Slice a List

Now, if you want to call just a few elements from your list (e.g. the first 4 items), you need to specify a range of index numbers separated by a colon [x:y]. Here's an example:

```
alpha_list[0:4]
[2, 23, 34, 67]
```

## Change Item Value on Your List

You can easily overwrite a value of one list items:

```
beta_list = [“apple”, “banana”, “orange”]
beta_list[1] = “pear”
print(beta_list)
```

Output:

```
[‘apple’, ‘pear’, ‘cherry’]
```

## Loop Through the List

Using **for loop** you can multiply the usage of certain items, similarly to what * operator does. Here's an example:

```
for x in range(1,4):
    beta_list += [‘fruit’]
    print(beta_list)
```

## Copy a List

Use the built-in **copy()** function to replicate your data:

```
beta_list = [“apple”, “banana”, “orange”]
beta_list = beta_list.copy()
print(beta_list)
```

Alternatively, you can copy a list with the **list()** method:

```
beta_list = [“apple”, “banana”, “orange”]
beta_list = list (beta_list)
print(beta_list)
```

# List Comprehensions

**List comprehensions** are a handy option for creating lists based on existing lists. When using them you can build by using **strings** and **tuples** as well.

## List comprehensions examples

```
list_variable = [x for x in iterable]
```

Here's a more complex example that features math operators, integers, and the **range()** function:

```
number_list = [x ** 2 for x in range(10) if x % 2 == 0]
print(number_list)
```

# Tuples

Tuples are similar to lists — they allow you to display an ordered sequence of elements. However, they are immutable and you can't change the values stored in a tuple.

The advantage of using tuples over lists is that the former are slightly faster. So it's a nice way to optimize your code.

## How to Create a Tuple

```
my_tuple = (1, 2, 3, 4, 5)
my_tuple[0:3]
(1, 2, 3)
```

Note: Once you create a tuple, you can't add new items to it or change it in any other way!

## How to Slide a Tuple

The process is similar to slicing lists.

```
numbers = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
print(numbers[1:11:2])
```

Output:

```
(1, 3, 5, 7, 9)
```

## Convert Tuple to a List

Since Tuples are immutable, you can't change them. What you can do though is convert a tuple into a list, make an edit and then convert it back to a tuple.

Here's how to accomplish this:

```
x = ("apple", "orange", "pear")
y = list(x)
y[1] = "grape"
x = tuple(y)
print(x)
```

# Dictionaries

A dictionary holds indexes with keys that are mapped to certain values. These key-value pairs offer a great way of organizing and storing data in Python. They are mutable, meaning you can change the stored information.

A key value can be either a **string**, **Boolean**, or **integer**. Here's an example dictionary illustrating this:

```
Customer 1= {'username': 'john-sea', 'online': false,
'friends':100}
```

## How to Create a Python Dictionary

Here's a quick example showcasing how to make an empty dictionary.

Option 1: **`new_dict = {}`**

Option 2: **`other_dict= dict()`**

And you can use the same two approaches to add values to your dictionary:

```
new_dict = {
  "brand": "Honda",
  "model": "Civic",
  "year": 1995
}
print(new_dict)
```

## How to Access a Value in a Dictionary

You can access any of the values in your dictionary the following way:

```
x = new_dict[“brand”]
```

You can also use the following methods to accomplish the same.

- **dict.keys()** isolates keys
- **dict.values()** isolates values
- **dict.items()** returns items in a list format of (key, value) tuple pairs

## Change Item Value

To change one of the items, you need to refer to it by its key name:

```
#Change the “year” to 2020:

new_dict= {
  “brand”: “Honda”,
  “model”: “Civic”,
  “year”: 1995
}
new_dict[“year”] = 2020
```

## Loop Through the Dictionary

Again to implement looping, use for loop command.

Note: In this case, the return values are the keys of the dictionary. But, you can also return values using another method.

```
#print all key names in the dictionary

for x in new_dict:
  print(x)

#print all values in the dictionary

for x in new_dict:
  print(new_dict[x])

#loop through both keys and values

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

# If Statements (Conditional Statements) in Python

Just like other programming languages, Python supports the basic logical conditions from math:

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

You can leverage these conditions in various ways. But most likely, you'll use them in "**if statements"** and **loops.**

## If Statement Example

The goal of a conditional statement is to check if it's True or False.

```
if 5 > 1:
        print("That's True!")
```

Output:

```
That's True!
```

## Nested If Statements

For more complex operations, you can create nested if statements. Here's how it looks:

```
x = 35

if x > 20:
  print("Above twenty,")
  if x > 30:
    print("and also above 30!")
```

## Elif Statements

**elif** keyword prompts your program to try another condition if the previous one(s) was not true. Here's an example:

```
a = 45
b = 45
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

## If Else Statements

**else** keyword helps you add some additional filters to your condition clause. Here's how an if-elif-else combo looks:

```
if age < 4:
ticket_price = 0
elif age < 18:
ticket_price = 10
else: ticket_price = 15
```

## If-Not-Statements

**Not** keyword let's you check for the opposite meaning to verify whether the value is NOT True:

```
new_list = [1, 2, 3, 4]
x = 10
if x not in new_list:
    print("'x' isn't on the list, so this is True!")
```

## Pass Statements

If statements can't be empty. But if that's your case, add the **pass** statement to avoid having an error:

```
a = 33
b = 200

if b > a:
  pass
```

# Python Loops

Python has two simple loop commands that are good to know:

- for loops
- while loops

Let's take a look at each of these.

## For Loop

As already illustrated in the other sections of this Python checklist, **for loop** is a handy way for iterating over a sequence such as a list, tuple, dictionary, string, etc.

Here's an example showing how to loop through a string:

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

Plus, you've already seen other examples for lists and dictionaries.

## While Loops

While loop enables you to execute a set of statements as long as the condition for them is true.

```
#print as long as x is less than 8

i = 1
while i< 8:
  print(x)
  i += 1
```

## How to Break a Loop

You can also stop the loop from running even if the condition is met. For that, use the break statement both in while and for loops:

```
i = 1
while i < 8:
  print(i)
  if i == 4:
    break
  i += 1
```

# Class

Since Python is an object-oriented programming language almost every element of it is an **object** — with its methods and properties.

**Class** acts as a blueprint for creating different objects. **Objects** are an instance of a class, where the class is manifested in some program.

## How to Create a Class

Let's create a class named TestClass, with one property named z:

```
class TestClass:
  z = 5
```

## How To Create an Object

As a next step, you can create an object using your class. Here's how it's done:

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

Further, you can assign different attributes and methods to your object. The example is below:

```
class car(object):
    """docstring"""

    def __init__(self, color, doors, tires):
        """Constructor"""
        self.color = color
        self.doors = doors
        self.tires = tires

    def brake(self):
        """
        Stop the car
        """
        return "Braking"

    def drive(self):
        """
        Drive the car
        """
        return "I'm driving!"
```

## How to Create a Subclass

Every object can be further sub-classified. Here's an example

```
class Car(Vehicle):
    """
    The Car class
    """

    def brake(self):
        """
        Override brake method
        """
        return "The car class is breaking slowly!"

if __name__ == "__main__":
    car = Car("yellow", 2, 4, "car")
    car.brake()
    'The car class is breaking slowly!'
    car.drive()
    "I'm driving a yellow car!"
```

# Dealing with Python Exceptions (Errors)

Python has a list of in-built exceptions (errors) that will pop up whenever you make a mistake in your code. As a newbie, it's good to know how to fix these.

## The Most Common Python Exceptions

- `AttributeError` — pops up when an attribute reference or assignment fails.
- `IOError` — emerges when some I/O operation (e.g. an `open()` function) fails for an I/O-related reason, e.g., "file not found" or "disk full".
- `ImportError` — comes up when an import statement cannot locate the module definition. Also, when a from... import can't find a name that must be imported.
- `IndexError` — emerges when a sequence subscript is out of range.
- `KeyError` — raised when a dictionary key isn't found in the set of existing keys.
- `KeyboardInterrupt` — lights up when the user hits the interrupt key (such as Control-C or Delete).
- `NameError` — shows up when a local or global name can't be found.

- `OSError` — indicated a system-related error.
- `SyntaxError` — pops up when a parser encounters a syntax error.
- `TypeError` — comes up when an operation or function is applied to an object of inappropriate type.
- `ValueError` — raised when a built-in operation/function gets an argument that has the right type but not an appropriate value, and the situation is not described by a more precise exception such as IndexError.
- `ZeroDivisionError` — emerges when the second argument of a division or modulo operation is zero.

# How to Troubleshoot the Errors

Python has a useful statement, design just for the purpose of handling exceptions – **try/except** statement. Here's a code snippet showing how you can catch KeyErrors in a dictionary using this statement:

```
my_dict = {“a”:1, “b”:2, “c”:3}
try:
        value = my_dict[“d”]
    except KeyError:
        print(“That key does not exist!”)
```

You can also detect several exceptions at once with a single statement. Here's an example for that:

```
my_dict = {“a”:1, “b”:2, “c”:3}
try:
    value = my_dict[“d”]
except IndexError:
    print(“This index does not exist!”)
except KeyError:
    print(“This key is not in the dictionary!”)
except:
    print(“Some other problem happened!”)
```

## try/except with else clause

Adding an else clause will help you confirm that no errors were found:

```
my_dict = {“a”:1, “b”:2, “c”:3}

try:
    value = my_dict[“a”]
except KeyError:
    print(“A KeyError occurred!”)
else:
    print(“No error occurred!”)
```

## Variables

We use variables to temporarily store data in computer's memory.

```
price = 10

rating = 4.9

course_name = 'Python for Beginners'

is_published = True
```

In the above example,

- **price** is an *integer* (a whole number without a decimal point)
- **rating** is a *float* (a number with a decimal point)
- **course_name** is a *string* (a sequence of characters)
- **is_published** is a *boolean*. Boolean values can be True or False.

## Comments

We use comments to add notes to our code. Good comments explain the hows and whys, not what the code does. That should be reflected in the code itself. Use comments to add reminders to yourself or other developers, or also explain your assumptions and the reasons you've written code in a certain way.

```
# This is a comment and it won't get executed.
# Our comments can be multiple lines.
```

## Receiving Input

We can receive input from the user by calling the **input()** function.

```
birth_year = int(input('Birth year: '))
```

The **input()** function always returns data as a string. So, we're converting the result into an integer by calling the built-in **int()** function.

## Strings

We can define strings using single (‘ ‘) or double (“ “) quotes.

To define a multi-line string, we surround our string with tripe quotes (“””).

We can get individual characters in a string using square brackets [].

```
course = ‘Python for Beginners’
course[0]   # returns the first character
course[1]   # returns the second character
course[-1]  # returns the first character from the end
course[-2]  # returns the second character from the end
```

We can slice a string using a similar notation:

```
course[1:5]
```

The above expression returns all the characters starting from the index position of 1 to 5 (but excluding 5). The result will be **ytho**

If we leave out the start index, 0 will be assumed.

If we leave out the end index, the length of the string will be assumed.

We can use formatted strings to dynamically insert values into our strings:

```
name = ‘Mosh’

message = f’Hi, my name is {name}’


message.upper()   # to convert to uppercase

message.lower()   # to convert to lowercase

message.title()   # to capitalize the first letter of every word

message.find(‘p’) # returns the index of the first occurrence of p
                   (or -1 if not found)

message.replace(‘p’, ‘q’)
```

To check if a string contains a character (or a sequence of characters), we use the **in** operator:

```
contains = 'Python' in course
```

## Arithmetic Operations

```
+

-

*

/     # returns a float

//    # returns an int

%     # returns the remainder of division

**    # exponentiation - x ** y = x to the power of y
```

Augmented assignment operator:

```
x = x + 10

x += 10
```

Operator precedence:

1. parenthesis
2. exponentiation
3. multiplication / division
4. addition / subtraction

## If Statements

```
if is_hot:
    print("hot day")
elif is_cold:
    print("cold day")
else:
    print("beautiful day")
```

Logical operators:

```
if has_high_income and has_good_credit:
    ...
if has_high_income or has_good_credit:
    ...
is_day = True
is_night = not is_day
```

## Comparison operators

```
a > b
a >= b (greater than or equal to)
a < b
a <= b
a == b (equals)
a != b (not equals)
```

## While loops

```
i = 1
while i < 5:
    print(i)
    i += 1
```

## For loops

```
for i in range(1, 5):
    print(i)
```

- **range(5):** generates 0, 1, 2, 3, 4
- **range(1, 5):** generates 1, 2, 3, 4
- **range(1, 5, 2):** generates 1, 3

## Lists

```
numbers = [1, 2, 3, 4, 5]
numbers[0]            # returns the first item
numbers[1]            # returns the second item
numbers[-1]           # returns the first item from the end
numbers[-2]           # returns the second item from the end


numbers.append(6)     # adds 6 to the end
numbers.insert(0, 6) # adds 6 at index position of 0
numbers.remove(6)     # removes 6
numbers.pop()         # removes the last item
numbers.clear()       # removes all the items
numbers.index(8)      # returns the index of first occurrence of 8
numbers.sort()        # sorts the list
numbers.reverse()     # reverses the list
numbers.copy()        # returns a copy of the list
```

## Tuples

They are like read-only lists. We use them to store a list of items. But once we define a tuple, we cannot add or remove items or change the existing items.

```
coordinates = (1, 2, 3)
```

We can unpack a list or a tuple into separate variables:

```
x, y, z = coordinates
```

## Dictionaries

We use dictionaries to store key/value pairs.

```
customer = {
    "name": "John Smith",
    "age": 30,
    "is_verified": True
}
```

We can use strings or numbers to define keys. They should be unique. We can use any types for the values.

```
customer["name"]                # returns "John Smith"
customer["type"]                # throws an error
customer.get("type", "silver") # returns "silver"
customer["name"] = "new name"
```

## Functions

We use functions to break up our code into small chunks. These chunks are easier to read, understand and maintain. If there are bugs, it's easier to find bugs in a small chunk than the entire program. We can also re-use these chunks.

```
def greet_user(name):
    print(f"Hi {name}")


greet_user("John")
```

**Parameters** are placeholders for the data we can pass to functions. **Arguments** are the actual values we pass.

We have two types of arguments:

- Positional arguments: their position (order) matters
- Keyword arguments: position doesn't matter - we prefix them with the parameter name.

```
# Two positional arguments
greet_user("John", "Smith")

# Keyword arguments
calculate_total(order=50, shipping=5, tax=0.1)
```

Our functions can return values. If we don't use the return statement, by default **None** is returned. None is an object that represents the absence of a value.

```
def square(number):
    return number * number

result = square(2)
print(result)  # prints 4
```

## Exceptions

Exceptions are errors that crash our programs. They often happen because of bad input or programming errors. It's our job to anticipate and handle these exceptions to prevent our programs from cashing.

```
try:
    age = int(input('Age: '))
    income = 20000
    risk = income / age
    print(age)
except ValueError:
    print('Not a valid number')
except ZeroDivisionError:
    print('Age cannot be 0')
```

## Classes

We use classes to define new types.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def move(self):
        print("move")
```

When a function is part of a class, we refer to it as a **method**.

Classes define templates or blueprints for creating objects. An object is an instance of a class. Every time we create a new instance, that instance follows the structure we define using the class.

```
point1 = Point(10, 5)
point2 = Point(2, 4)
```

__init__ is a special method called constructor. It gets called at the time of creating new objects. We use it to initialize our objects.

## Inheritance

Inheritance is a technique to remove code duplication. We can create a *base class* to define the common methods and then have other classes inherit these methods.

```
class Mammal:
  def walk(self):
     print("walk")

class Dog(Mammal):
  def bark(self):
     print("bark")


dog = Dog()
dog.walk()   # inherited from Mammal
dog.bark()   # defined in Dog
```

## Modules

A module is a file with some Python code. We use modules to break up our program into multiple files. This way, our code will be better organized. We won't have one gigantic file with a million lines of code in it!

There are 2 ways to import modules: we can import the entire module, or specific objects in a module.

```
# importing the entire converters module
import converters
converters.kg_to_lbs(5)

# importing one function in the converters module
from converters import kg_to_lbs
kg_to_lbs(5)
```

## Packages

A package is a directory with **__init__.py** in it. It can contain one or more modules.

```
# importing the entire sales module
from ecommerce import sales
sales.calc_shipping()

# importing one function in the sales module
from ecommerce.sales import calc_shipping
calc_shipping()
```

## Python Standard Library

Python comes with a huge library of modules for performing common tasks such as sending emails, working with date/time, generating random values, etc.

### Random Module

```
import random

random.random()       # returns a float between 0 to 1
random.randint(1, 6)  # returns an int between 1 to 6

members = ['John', 'Bob', 'Mary']
leader = random.choice(members) # randomly picks an item
```

# Pypi

Python Package Index (pypi.org) is a directory of Python packages published by Python developers around the world. We use **pip** to install or uninstall these packages.

```
pip install openpyxl
```

```
pip uninstall openpyxl
```