

# FULL STACK



## Python Training Certification Course

## Error Handling and File Operations





# Learning Objectives

By the end of this lesson, you will be able to:

- 👁 Demonstrate file handling
- 👁 Explain errors and exceptions
- 👁 Explain debugging
- 👁 Explain object-oriented programming



# FULL STACK

## File Handling

# File Handling

File handling handles file operations like reading and writing to files. Functions that allow file handling in Python:

`open()`: Opens a file in read or write mode

`w`: Creates and writes to a file

`a`: Appends to a file

`r`: Reads a file

`r+`: Reads as well as writes to a file



# Create a File and Write to It

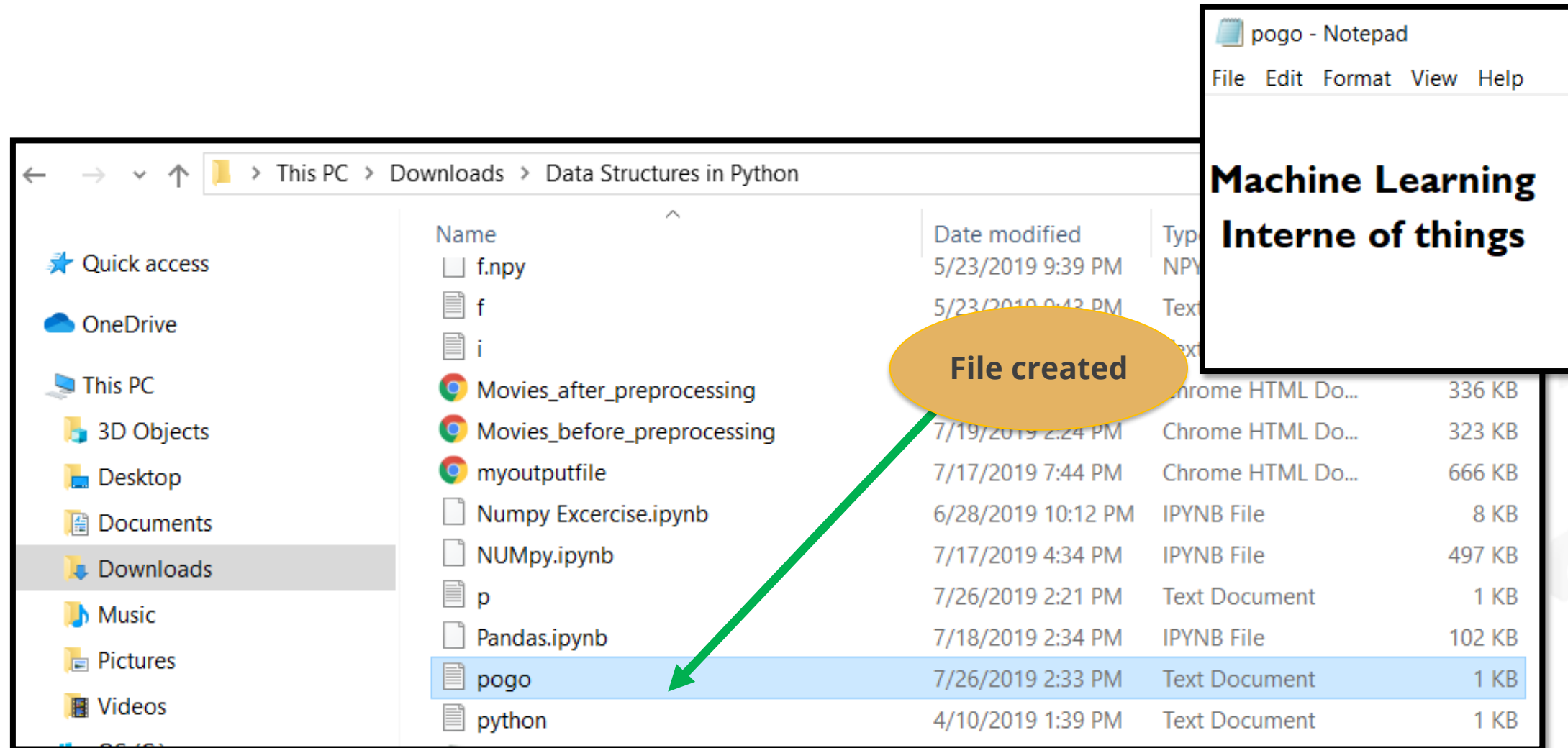
Variable  
declaration

File name  
and  
location

Write to a  
file

```
In [13]: f=open("pogo.txt","w")#creating a text file
          f.write("\nMachine Learning ")# writing to file
          f.write("\n Interne of things\n")
          f.close()
```

# Check Created File



The screenshot shows a Windows File Explorer window with the address bar set to 'This PC > Downloads > Data Structures in Python'. The left sidebar shows the 'Downloads' folder selected. The main pane displays a list of files and folders. The file 'pogo' is highlighted in blue. A green arrow points from an orange oval labeled 'File created' to the 'pogo' file. An inset window titled 'pogo - Notepad' shows the content of the file: 'Machine Learning' and 'Interne of things'.

Name	Date modified	Type	Size
f.npy	5/23/2019 9:39 PM	NPY	
f	5/23/2019 9:43 PM	Text	
i		Text	
Movies_after_preprocessing		Chrome HTML Do...	336 KB
Movies_before_preprocessing	7/19/2019 2:24 PM	Chrome HTML Do...	323 KB
myoutputfile	7/17/2019 7:44 PM	Chrome HTML Do...	666 KB
Numpy Excercise.ipynb	6/28/2019 10:12 PM	IPYNB File	8 KB
NUMpy.ipynb	7/17/2019 4:34 PM	IPYNB File	497 KB
p	7/26/2019 2:21 PM	Text Document	1 KB
Pandas.ipynb	7/18/2019 2:34 PM	IPYNB File	102 KB
pogo	7/26/2019 2:33 PM	Text Document	1 KB
python	4/10/2019 1:39 PM	Text Document	1 KB

**File created**

**Machine Learning**  
**Interne of things**

**Note: File location is based on the path we give while running code for file creation.**

# Read Files in Python

Reads a complete file

```
In [15]: f=open("pogo.txt","r")#creating a text file  
a=f.read()# writing to file  
print(a)  
f.close()
```

Machine Learning  
Interne of things

Reads 6 characters

```
In [16]: f=open("pogo.txt","r")#creating a text file  
a=f.read(6)  
print(a)  
f.close()
```

Machi

Reads according to number of lines

```
In [17]: f=open("pogo.txt","r")#creating a text file  
a=f.readlines()  
print(a)  
f.close()
```

```
['\n', 'Machine Learning \n', ' Interne of things\n']
```



# File Functions

## To get the file size of a plain file

```
In [23]: def file_size(file):  
         import os  
         statinfo = os.stat(file)  
         return statinfo.st_size  
  
print("File size in bytes of a plain file = ",file_size("pogo.txt"),"bytes")
```

File size in bytes of a plain file = 45 bytes

## To copy one file to another

```
In [24]: from shutil import copyfile  
         copyfile('pogo.txt', 'ab1.txt')  
         print("ab1.txt is copy created")
```

ab1.txt is copy created

# File Handling



**Duration: 20 min.**

**Objective:** Write a program using Python to demonstrate file handling.

Steps to demonstrate file handling:

1. Open Jupyter Notebook
2. Click on File ▾ New ▾ Notebook
3. Select Python (version 3)
4. Write your program
5. Save your program
6. Click on Run to execute program

ASSISTED PRACTICE

## Errors and Exceptions

# Errors

One has to follow language constraints while writing code. If something doesn't work correctly, errors will be shown in the program. Common errors that may occur while writing a Python script:

Syntax Error occurs when a certain statement is not in accordance with the prescribed use.

```
In [1]: ##Synatx error  
print("hi"
```

```
File "<ipython-input-1-8e7882f98363>", line 2  
    print("hi"  
          ^
```

```
SyntaxError: unexpected EOF while parsing
```



# Errors

Indentation Error occurs when an indentation is not in accordance with the prescribed format.

```
In [3]: ##Indentation error
print("Errors")
x=8
if x>9:
    print("Hey")
    print("Hello")
```

File "<ipython-input-3-89ca33a20ade>", line 6

```
    print("Hello")
        ^
```

**IndentationError:** unindent does not match any outer indentation level

# Errors

NameError occurs when we call an undefined variable.

```
In [4]: ##Name error
        if x>6:
            print("YES")
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-4-34f98b2fd3af> in <module>()
      1 ##Name error
----> 2 if x>6:
      3     print("YES")

NameError: name 'x' is not defined
```

# Errors

ValueError occurs when we use the wrong data type.

```
In [5]: ##ValueError
x=int(input("Enter your age "))

Enter your age 23.4

-----
ValueError                                Traceback (most recent call last)
<ipython-input-5-e81cfb30c237> in <module>()
      1 ##ValueError
----> 2 x=int(input("Enter your age "))

ValueError: invalid literal for int() with base 10: '23.4'
```

# Errors

TypeError shows up for invalid data operations.

```
In [6]: ##TypeError
        "name"*"age"

-----
TypeError                                Traceback (most recent call last)
<ipython-input-6-e5120324eb45> in <module>()
      1 ##TypeError
----> 2 "name"*"age"

TypeError: can't multiply sequence by non-int of type 'str'
```



# Errors

IndexError occurs when we try to access an invalid index.

```
In [7]: ##IndexError
List1=[1,2,3,4]
List1[5]
```

```
-----
IndexError                                Traceback (most recent call last)
<ipython-input-7-84a2aa1eb90b> in <module>()
      1 ##IndexError
      2 List1=[1,2,3,4]
----> 3 List1[5]

IndexError: list index out of range
```

# Errors

KeyError occurs when we call a wrong key.

```
In [8]: ##KeyError
d={"one":1,"two":2}
d["three"]

-----
KeyError                                Traceback (most recent call last)
<ipython-input-8-fa73398e8746> in <module>()
      1 ##KeyError
      2 d={"one":1,"two":2}
----> 3 d["three"]

KeyError: 'three'
```

# Errors

ModuleNotFoundError occurs when we use an invalid module.

```
In [9]: ##ModuleError  
import times
```

```
-----  
ModuleNotFoundError                                Traceback (most recent call last)  
<ipython-input-9-5cf5683187b9> in <module>()  
      1 ##KeyError  
----> 2 import times
```

```
ModuleNotFoundError: No module named 'times'
```

# Errors

ImportError occurs when we use a wrong function inside a module.

```
In [12]: ##ImportError
          from time import call
          call.time()

-----
ImportError                                Traceback (most recent call last)
<ipython-input-12-0935a03c31f7> in <module>()
      1 ##ImportError
----> 2 from time import call
      3 call.time()

ImportError: cannot import name 'call' from 'time' (unknown location)
```



# Errors

FileNotFoundError occurs when we open a nonexistent file.

```
In [14]: ##FileNotFoundError
f=open("sample.txt","r")
f.write()

-----
FileNotFoundError                                Traceback (most recent call last)
<ipython-input-14-757d86d3e22a> in <module>()
      1 ##FileNotFoundError
----> 2 f=open("sample.txt","r")
      3 f.write()

FileNotFoundError: [Errno 2] No such file or directory: 'sample.txt'
```

# Errors

ZeroDivisionError occurs while dividing a number by zero.

```
In [15]: ##ZeroDivisionError  
4/0
```

```
-----  
ZeroDivisionError                                Traceback (most recent call last)  
<ipython-input-15-d0cfd0260bb2> in <module>()  
      1 ##FileNotFoundError  
----> 2 4/0
```

```
ZeroDivisionError: division by zero
```

# Errors

AttributeError occurs when we use a wrong method.

```
In [20]: a=(1,2,3)
         a.append(3)

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-20-a68dc5693b1a> in <module>()
      1 a=(1,2,3)
----> 2 a.append(3)

AttributeError: 'tuple' object has no attribute 'append'
```

# Errors

KeyboardInterrupt is thrown when the user hits the interrupt key(normally, Control-C) during the execution of the program.

```
In [19]: Entry1=input('enter your name')
```

```
enter your name
```





# Exceptions

Error handling can be done in Python using Exceptions. Exception handling includes try(), except(), catch(), and finally() statements. It's not necessary to use them in a single program. In the code below, ZeroDivisionError is handled through try, except, and finally functions.

```
In [22]: try:
          x=int(input("NUMBER1 = "))
          y=int(input("NUMBER2 = "))
          print(x+y)
          print(x/y)
          print(x*y)
        except ZeroDivisionError:
          print("Wrong operation :")

        finally:
          print("The 'try except' is finished")

NUMBER1 = 2
NUMBER2 = 0
2
Wrong operation :
The 'try except' is finished
```

If y==0, it jumps to except block

Exception handled without error message

# Exceptions

In the example below, try and except functions are used to perform exception handling for ValueError.

```
In [23]: try:
          x=int(input("NUMBER="))
          y=int(input("Number"))
          print(x/y)
        except Exception as e:
          print('Caught this error: ' + repr(e))
```

NUMBER=2  
Number3.4  
Caught this error: ValueError("invalid literal for int() with base 10: '3.4'")

Error thrown through try method

# Exceptions

In the example below, multiple exceptions are handled.

```
In [26]: try:
          x=int(input("Enter First Number="))
          y=int(input("Enter Second Number="))
          print("sum is:", x+y)
          print("div is:",x/y)
          print("product is:",x*y)
        except Exception as e:
            if type(e) == ZeroDivisionError:
                print("oops!!Error wrong divisor")
            elif type(e)== ValueError:
                print("Wrong datatype")
            else:
                pass
```

For  
ValueError

For  
ZeroDivisionError

```
Enter First Number=2
Enter Second Number=3.4
Wrong datatype
```

# Errors and exceptions



**Duration: 20 min.**

**Objective:** Write a program using Python to demonstrate errors and exceptions.

Steps to demonstrate errors and exceptions:

1. Open Jupyter Notebook
2. Click on File ▾ New ▾ Notebook
3. Select Python (version 3)
4. Write your program
5. Save your program
6. Click on Run to execute program

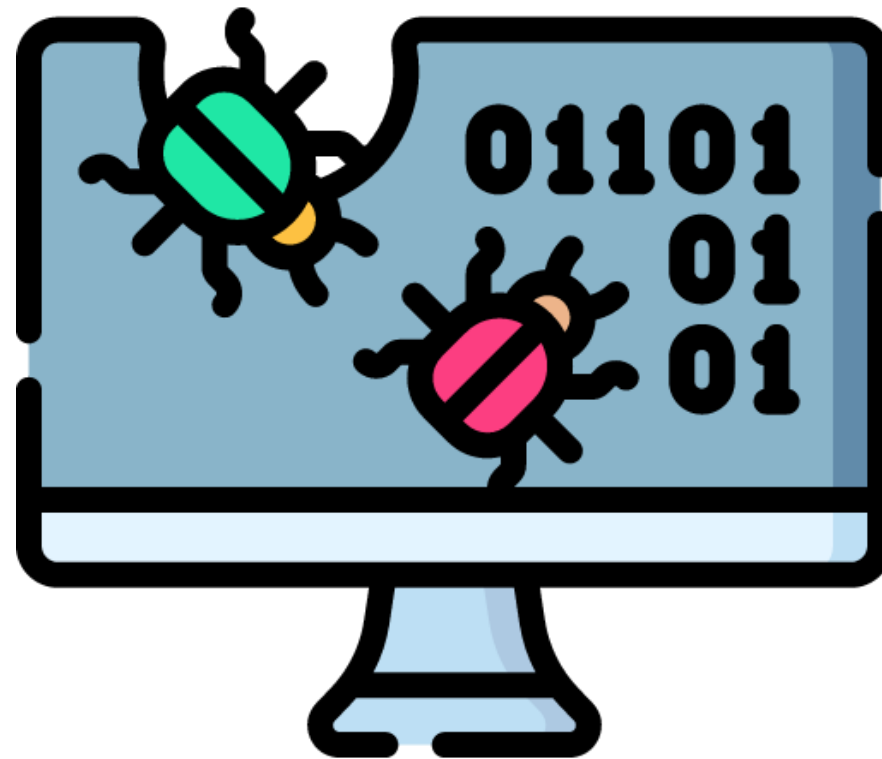
ASSISTED PRACTICE

# FULL STACK

## Debugging in Python

# Debugging

- Programming errors often cause programs to crash during execution or give the wrong output.
- Errors in programs are called bugs.
- The process of removing bugs is called debugging.

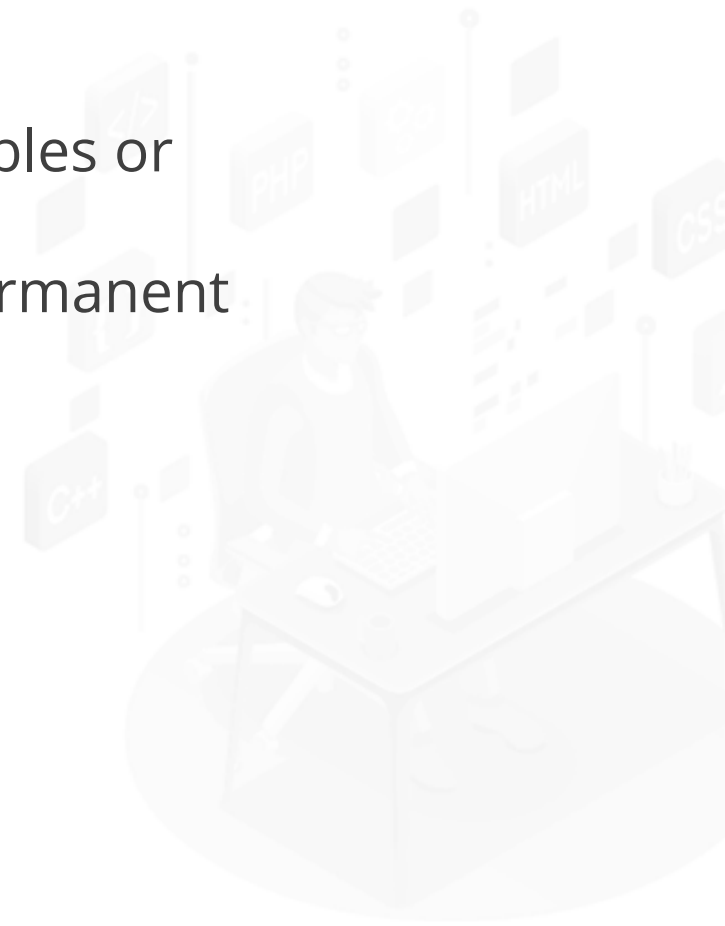




# Debugging through Logging

---

- The simplest debugging approach is to output variables and other information which are related to the code.
- This helps to follow the program flow.
- The easiest way to do this is print debugging.
- It involves inserting print statements at certain points to print the value of variables or points while debugging.
- Combining print debugging with logging techniques allow us to create a semipermanent trace of the execution of the program.



# Logging



**Objective:** You are given a project to debug a sorting program using log statements.

Steps to debug a sorting algorithm using logging:

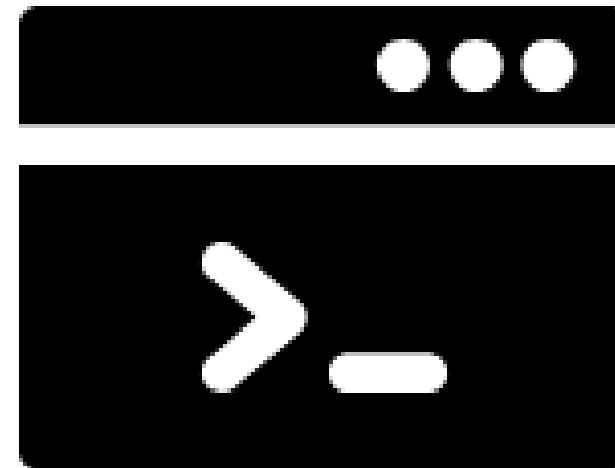
1. Open the code editor.
2. Code the merge sort algorithm in Python.
3. Add logging statements at strategic points.
4. Execute the code.

ASSISTED PRACTICE

# Debugging via Command Line

---

- Find errors suggesting wrong syntax or incorrect function calls using command line.
- Run what the program was doing when it crashed at the command line.
- Integrate it into your software once you get it to work on the command prompt.



# FULL STACK

## pdb Module

# Python Debugger

- You can easily debug in Python.
- You don't need a full-blown IDE to debug your Python application.
- You can debug your python code with the Python debugger, **pdb**.



# Python Debugger

---

The **pdb** module defines an interactive source code debugger for Python. It supports the following features:

- Breakpoints
- Single stepping at the source-line level
- Inspection of stack frames
- Source code listing
- Evaluation of arbitrary Python code in the context of any stack frame
- Post-mortem debugging





# Features of pdb

---

- pdb can be called program under control.
- The debugger is extensible.
- pdb is defined as the class pdb.
- The extension interface uses the modules, bdb and cmd.
- The debugger's prompt is **(Pdb)**.



# Debugger Prompt

---

Running a program under debugger control:

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
    <string>(0)?()
(Pdb) continue
    <string>(1)?()
(Pdb) continue
NameError: 'spam'
    <string>(1)?()
(Pdb)
```



# Debugger Prompt

---

Inspecting a crashed program with the debugger:

```
>>> import pdb
```

```
>>> import mymodule
```

```
>>> mymodule.test()
```

Traceback (innermost last):

File "<stdin>", line 1, in ?

File "./mymodule.py", line 4, in test  
test2()

File "./mymodule.py", line 3, in test2  
print spam

NameError: spam

```
>>> pdb.pm()
```

```
> ./mymodule.py(3)test2()
```

```
-> print spam
```

```
(Pdb)
```



# pdb Methods

The pdb module defines the following functions:

- **run (statement[, globals[, locals]])**: Executes the statement given as a string under debugger control
- **runeval (expression[, globals[, locals]])**: Evaluates the expression under debugger control and returns the value of the expression
- **runcall (function[, argument, ...])**: Calls the function with the given arguments and returns whatever the function call returned
- **set\_trace ()**: Enters the debugger at the calling stack frame and makes it easy to hardcode a breakpoint at a given point in a program
- **post\_mortem (traceback)**: Enters post-mortem debugging of the given traceback object
- **pm ()**: Enters post-mortem debugging of the traceback found in sys.last\_traceback

# pdb Commands

Popular pdb commands and their use:

Command	Description
p	Prints the value of an expression
pp	Pretty-prints the value of an expression
n	Continues execution until the next line in the current function is reached or it returns
s	Executes the current line and stops at the first possible occasion (either in a function that is called or in the current function)
c	Continues execution and only stops when a breakpoint is encountered
unt	Continues execution until the line with a number greater than the current one is reached
l	Lists source code for the current file and without arguments, lists 11 lines around the current line or continues the previous listing
ll	Lists the whole source code for the current function or frame

# pdb Commands

b	Lists all breaks with no arguments and with a line number argument, sets a breakpoint at this line in the current file
w	Prints a stack trace with the most recent frame at the bottom where an arrow indicates the current frame, which determines the context of most commands
d	Moves the current frame count (default one) levels down in the stack trace (to a newer frame)
h	Sees a list of available commands
h <topic>	Shows help for a command or topic
h pdb	Shows the full pdb documentation
q	Quits the debugger and exits



# Debugging Using pdb

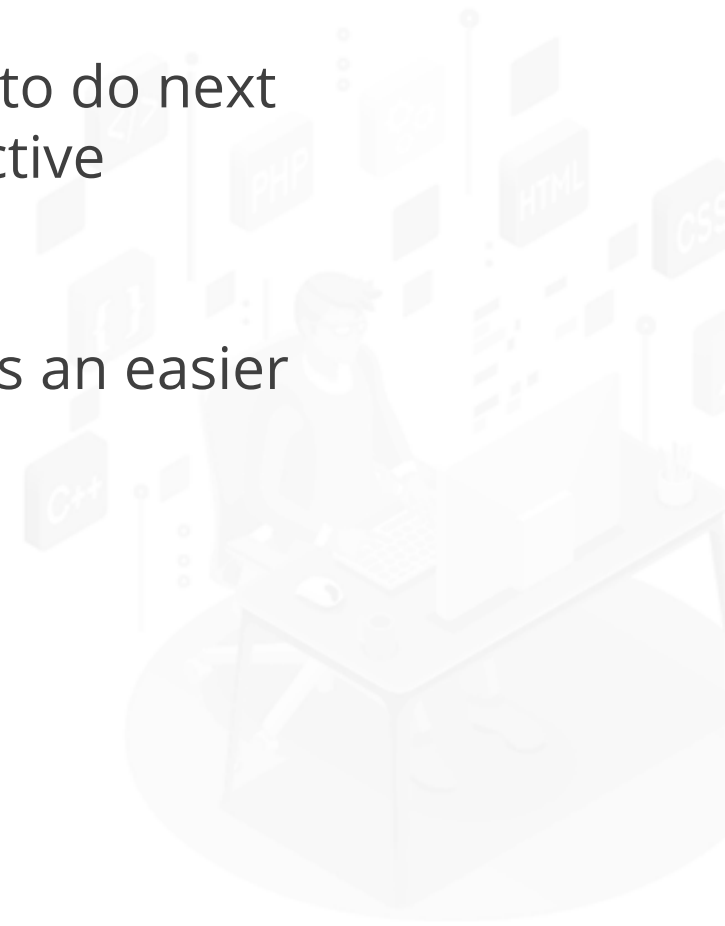
The pdb module defines the following functions:

- Printing a variable's value
- Printing expressions
- Stepping through code
- Using breakpoints
- Continuing execution
- Displaying expressions
- Identifying the calling function



# Printing a Variable's Value

- pdb checks the value of a variable
- Following code breaks into the debugger where you want:
  - **import pdb; pdb.set\_trace()**
- When this line is executed, Python stops and waits for your instruction on what to do next
- The (Pdb) prompt will be displayed, indicating that you are paused in the interactive debugger and can enter a command
- To print the value, enter p variable\_name at the (Pdb) prompt
- The built-in function breakpoint(), that is there from Python version 3.7, provides an easier alternative to break into code



# Printing Expressions

- When we use the print command, we're passing an expression to be evaluated by Python.
- If we pass a variable name, pdb prints its current value, but we can print much more to investigate the state of our running application.
- We could pause a function and use the command **ll** to list the function's source.
- We could run *p filename*, *p head*, *p get\_path*, or any valid Python expression.



# pdb Print Command



**Objective:** You are given a project to debug a program using the pdb print command.

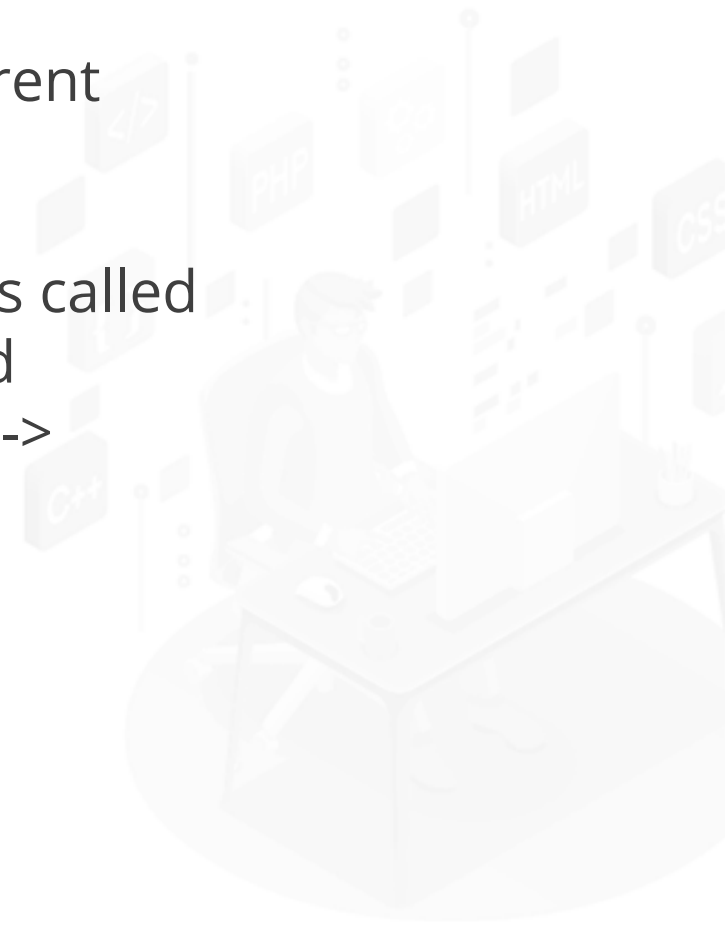
Steps to debug a program using the pdb print command:

1. Open the code editor.
2. Code a program in Python.
3. Add `pdb.set_trace` at strategic points to break into the debugger.
4. Execute the code and print variables and expressions when the program pauses.

ASSISTED PRACTICE

# Stepping through Code

- There are two commands that can step through code when debugging: n (next) and s (step)
- pdb stops between n and s
- We use n (next) to continue execution until the next line and stay within the current function
- n (next) does not stop in a foreign function if one is called
- We use s (step) to execute the current line and stop in a foreign function if one is called
- Both n and s will stop execution when the end of the current function is reached
- They print *--Return--* along with the return value at the end of the next line after ->



# next and step Commands



**Objective:** You are given a project to demonstrate the use of n and s commands.

Steps to demonstrate the use of n and s commands:

1. Open the code editor.
2. Code a program in Python.
3. Add `pdb.set_trace` at strategic points to break into the debugger.
4. Execute the code and use n and s commands to step through the code.

ASSISTED PRACTICE

# Using Breakpoints

- Breakpoints are easier and faster than stepping through code
- Instead of stepping through dozens of lines, simply create a breakpoint where you want to investigate
- The `b` (break) command sets a breakpoint
- A line number or a function name where execution has to be stopped can be specified as an argument
- The syntax for break is: `b(reak) [ ([filename:]lineno | function) [, condition] ]`
- If filename is not specified, the current source file is used
- The condition argument is optional
- The optional 2nd argument condition lets the code break only if a certain condition exists





# Breakpoints



**Objective:** You are given a project to demonstrate the use of breakpoints.

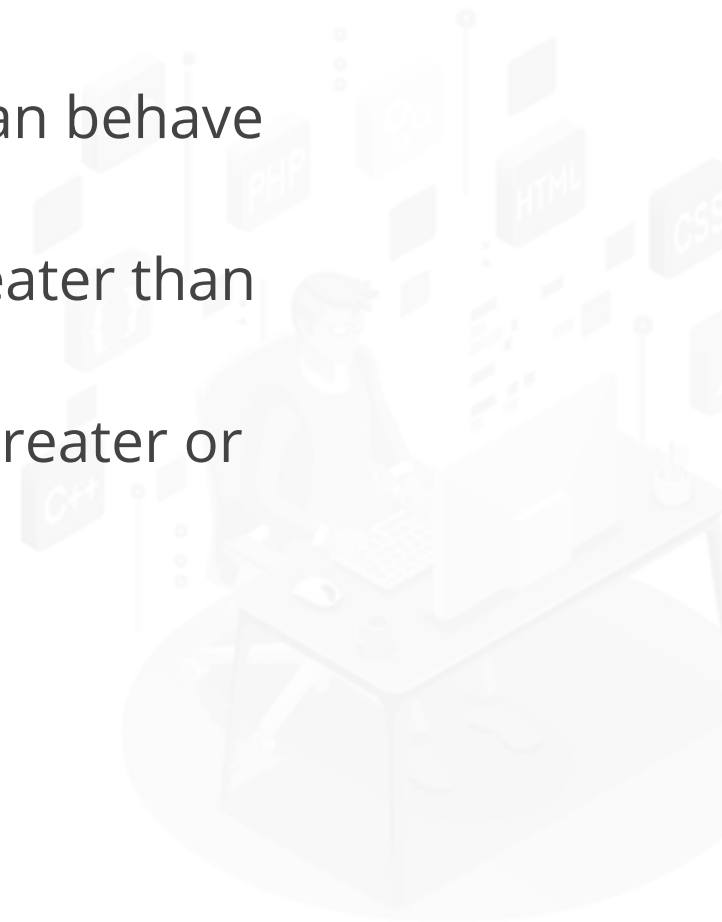
Steps to demonstrate the use of breakpoints:

1. Open the code editor.
2. Code a util function in Python.
3. Write a Python program with a breakpoint.
4. Execute the code.

ASSISTED PRACTICE

# Continuing Execution

- The **unt** command continues execution but stops at the next line greater than the current line
- Syntax for unt: `unt(il) [lineno]`
- Depending on whether or not you pass the line number argument `lineno`, `unt` can behave in two ways:
  - Without `lineno`, it will continue execution until the line with a number greater than the current one is reached, similar to `n` (next)
  - With `lineno`, the code will continue execution until a line with a number greater or equal to that is reached



# Continuing Execution

- `unt` stops when the current frame (function) returns, just like `n` (next) and `s` (step)
- `unt` continues execution and stops further down in the current source file
- It's a hybrid of `n` (next) and `b` (break), depending on whether or not a line number argument is passed



# unt Command



**Objective:** You are given a project to demonstrate the use of unt command.

Steps to demonstrate the use of unt command:

1. Open the code editor.
2. Code a program in Python.
3. Add `pdb.set_trace` at strategic points to break into the debugger.
4. Execute the code and run the unt command.

ASSISTED PRACTICE

# Displaying Expressions

- We can use the command **display** to tell pdb to automatically display the value of an expression if it is changed when execution stops
- It is similar to printing expressions with p and pp
- The command, **undisplay**, clears a display expression
- The syntax for display is: **display [expression]**
- The syntax for undisplay is: **undisplay [expression]**



# display Command



**Objective:** You are given a project to demonstrate the use of the display command.

Steps to demonstrate the use of display command:

1. Open the code editor.
2. Code a program in Python.
3. Add `pdb.set_trace` at strategic points to break into the debugger.
4. Execute the code and run the display command.

ASSISTED PRACTICE

# Identifying the Calling Function

- Suppose, there's a large code base with a function in a utility module that's being called with invalid input.
- If it's being called from many places in different packages, we'll need to find out where the call is from, to debug it.
- We can use the command `w` (where) to print a stack trace.
- `w` will print the stack trace with the most recent frame at the bottom.





# w Command



**Objective:** You are given a project to demonstrate the use of the w command.

Steps to demonstrate the use of w command:

1. Open the code editor.
2. Code a program with a util function in a different file.
3. Add `pdb.set_trace` at strategic points to break into the debugger.
4. Execute the code and run the w command to identify the function caller.

ASSISTED PRACTICE

# FULL STACK

## Object-Oriented Programming

# Python: Object-Oriented Language

Python is an object-oriented programming in which classes are models for objects. Objects get their variables and functions from classes.

A basic class in Python looks like:

```
In [6]: class A:
        x="Classes"
        #Tell about status of class.
        A.x
```

```
Out[6]: 'Classes'
```

```
In [5]: #Class variables
        class student:
            name="RAM"
            age = 29
        print("NAME is %s and age is %d"%(student.name,student.age))
```

```
NAME is RAM and age is 29
```

# Creating Objects and Accessing Variables

Variables for objects are generated through classes and functions.

```
In [9]: #Class & Objects
class Phone():
    Color=" "
    Price =00
    Type=" "
ob1=Phone()#Creation of objects and assinging it to class phone.
ob2=Phone()
ob1.Color="RED"
ob1.Price=10000
ob1.Type="Nokia"
ob2.Color="Black"
ob2.Price=12000
ob2.Type="Asus"
print("This is a %s phone of %s color and its price is %drs"%(ob1.Type,ob1.Color,ob1.Price))
print("This is a %s phone of %s color and its price is %drs"%(ob2.Type,ob2.Color,ob2.Price))
```

The diagram illustrates the relationship between class variables and object access. A green bracket on the left groups the class variable definitions (Color, Price, Type) in the code, with a callout bubble labeled "Class variables". Another green bracket on the left groups the object variable assignments (ob1, ob2) and their attribute accesses (ob1.Color, ob1.Price, ob1.Type, ob2.Color, ob2.Price, ob2.Type), with a callout bubble labeled "Objects having access to class variables".

```
This is a Nokia phone of RED color and its price is 10000rs
This is a Asus phone of Black color and its price is 12000rs
```

# Objects and Functions

In the following example, objects get the bill() function and return it.

```
In [12]: class Shop:
          "Things in MYshop"
          Quantity = 10
          Type = ""
          price=500.00
          def bill(self):
              t=self.price*self.Quantity
              return t

          coustmer1=Shop()
          coustmer1.Quantity=5
          coustmer1.Type="saree"
          coustmer1.price=20000
          coustmer1.bill()

Out[12]: 100000
```

Variables are  
defined inside  
functions through  
objects

## \_\_init\_\_() Method

- The **\_\_init\_\_()** method in Python is a constructor. This function gets called every time you instantiate the class.
- **self** represents the instance of the class. By using the **self** keyword, we can access attributes and methods of a class in Python. It binds attributes with the given arguments.

In [10]:

```
class ID:
    def __init__(self,name,age):
        self.name=name
        self.age=age
ob=ID("ran",18)
print(ob.name , ob.age)
```

Variables are  
defined using  
self keyword

ran 18




## \_\_init\_\_() Method

We can create a number of objects using the `__init__()` method.

```
In [16]: class Team():
          def __init__(self, id, type):
              self.id = id
              self.type = type

          player1 = Team(101, "batsman")
          player2 = Team(102, "fastbowler")
          print("Player1 with id %d is a %s" %(player1.id, player1.type))
          print("Player2 with id %d is a %s" %(player2.id, player2.type))
```




Player1 with id 101 is a batsman  
Player2 with id 102 is a fastbowler



# Inheritance

- Inheritance allows us to define a class that inherits methods and properties from another class.
- Parent class is the class being inherited from and is also called a base class.
- Child class is the class that inherits from another class and is also called a derived class.



```
In [18]: class House:
          def __init__(self, area, color):
              self.netarea = area
              self.color = color

          def printdetails(self):
              print(self.netarea, self.color)

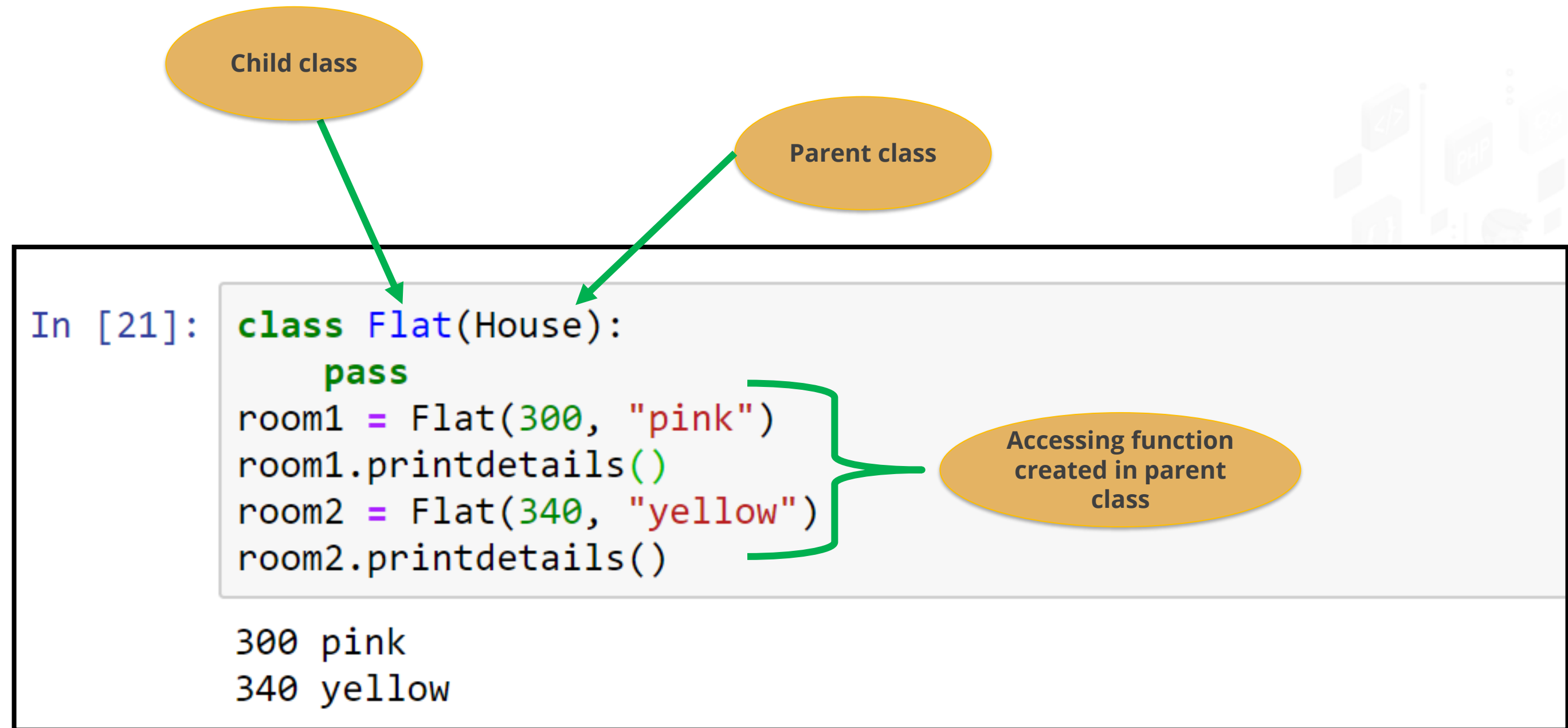
#Use the House class to create an object, and then execute the printdetails method:

room = House(250, "blue")
room.printdetails()

250 blue
```

# Inheritance

Example of inheritance:



# Classes and Objects



**Duration: 30 min.**

**Objective:** Write a program using Python to demonstrate classes and objects.

Steps to demonstrate classes and objects:

1. Open Jupyter Notebook
2. Click on File ▾ New ▾ Notebook
3. Select Python (version 3)
4. Write your program
5. Save your program
6. Click on Run to execute program

ASSISTED PRACTICE

# FULL STACK



## Knowledge Check

## Knowledge Check

1

Which of the following is the cause for ValueError in Python?

- a. Out-of-range value generation
- b. Wrong operations
- c. Incorrect data types
- d. Wrong methods



## Knowledge Check

1

Which of the following is the cause for ValueError in Python?

- a. Out-of-range value generation
- b. Wrong operations
- c. Incorrect data types
- d. Wrong methods



The correct answer is **c**

**ValueError occurs when we use wrong data types.**

## Knowledge Check

2

Which of the following is the difference between using “r” or “r+” while reading a file?

- a. “r” only reads a file while “r+” reads as well writes to a file
- b. “r” reads an entire file while “r+” reads lines of a file
- c. “r” reads an entire file while “r+” reads as well copies to a file
- d. “r” reads an entire file while “r+” reads as well writes to a file





## Knowledge Check

2

Which of the following is the difference between using “r” or “r+” while reading a file?

- a. “r” only reads a file while “r+” reads as well writes to a file
- b. “r” reads an entire file while “r+” reads lines of a file
- c. “r” reads an entire file while “r+” reads as well copies to a file
- d. “r” reads an entire file while “r+” reads as well writes to a file



The correct answer is **a**

**“r+” reads as well as writes to a file while “r” only reads.**

## Knowledge Check

3

**Which of the following statements is true for Python?**

- a. Objects get their variables and functions from classes
- b. Syntax errors can't be handled using exception
- c. File handling handles file operations like reading and writing to files
- d. All of the above



## Knowledge Check

3

Which of the following statements is true for Python?

- a. Objects get their variables and functions from classes
- b. Syntax errors can't be handled using exception
- c. File handling handles file operations like reading and writing to files
- d. All of the above



The correct answer is **d**

**In Python, objects get their variables and functions from classes, syntax errors can't be handled using exceptions, and file handling handles file operations like reading and writing to files.**

## Knowledge Check

4

Which of the following is required to create a new instance of a class?

- a. A constructor
- b. An object
- c. A class
- d. A value returning function



## Knowledge Check

4

Which of the following is required to create a new instance of a class?

- a. A constructor
- b. An object
- c. A class
- d. A value returning function



The correct answer is **a**

**The `__init()` method in Python is a constructor. This function gets called every time you instantiate a class.**

## Knowledge Check

5

What will be the output of following expression?

- a. Ashi
- b. Sid
- c. Error
- d. Sid Jon

```
class details:  
    def __init__(self, name):  
        self.name = name  
        name = "Ashi"  
entry1 = details("Sid")  
entry2 = details("Jon")  
print(entry1.name,entry2.name)
```



Knowledge  
Check

5

What will be the output of following expression?

- a. Ashi
- b. Sid
- c. Error
- d. Sid Jon

```
class details:  
    def __init__(self, name):  
        self.name = name  
        name = "Ashi"  
entry1 = details("Sid")  
entry2 = details("Jon")  
print(entry1.name,entry2.name)
```



The correct answer is **d**

Python is an object-oriented language where objects gets their variable & functions from classes.



## Key Takeaways

- ➊ Exception handling is done to maintain flow of a code.
- ➋ pdb module defines an interactive source code debugger for Python.
- ➌ Class is a model for objects in Python.
- ➍ Class methods must have an extra first parameter in function definition to point to class attributes.

