Snakes N' Shakes 3

Benjamin Short

Agenda

Input Widgets

Databases

Sockets / Threading

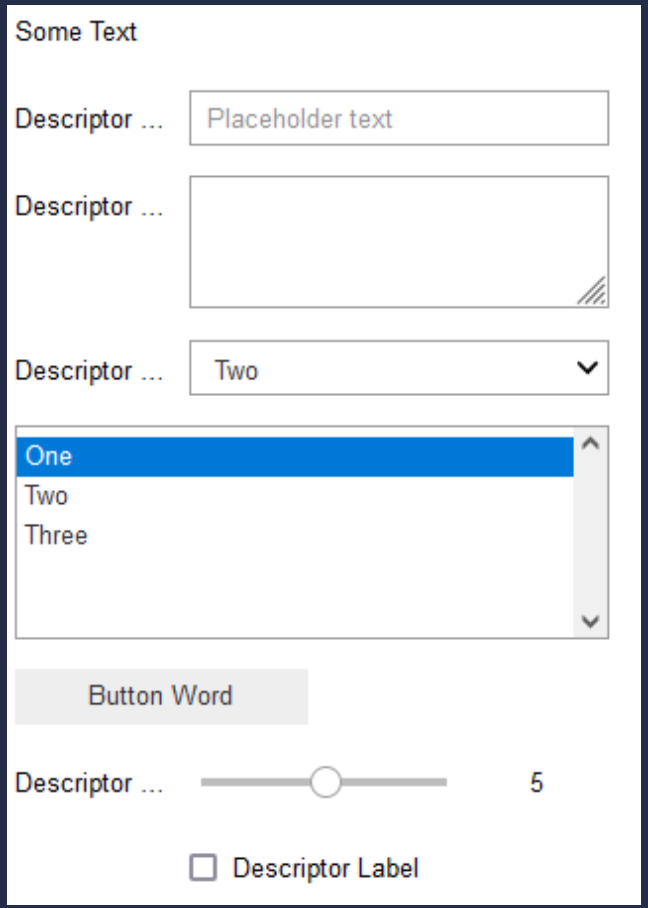
Classes

Dunder Methods

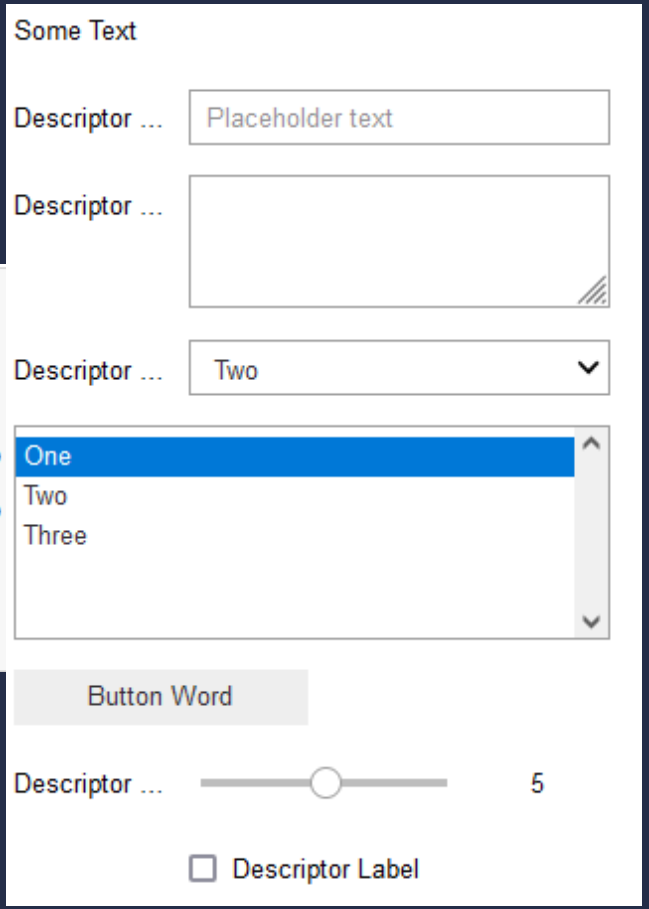
User Input - Widgets

Jupyter uses 'widgets' for visual interaction with users. Input can also be obtained through numerous different widget types:

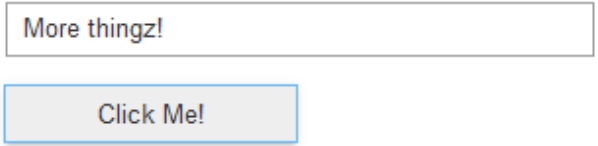
- Text
- Dropdown
- Selection
- Buttons
- Sliders
- Checkboxes
- Etc...



```
from ipywidgets import widgets

opts = ['One', 'Two', 'Three']
lay = widgets.Layout()

display(widgets.Label(value="Some Text"))
display(widgets.Text(placeholder="Placeholder text", description="Descriptor Label", layout=lay))
display(widgets.Textarea(description="Descriptor Label", layout=lay))
display(widgets.Dropdown(options=opts, value="Two", description="Descriptor Label", layout=lay))

display(widgets.Select(options=opts, layout=lay))
display(widgets.Button(description="Button Word", layout=lay))
display(widgets.IntSlider(value=5, min=0, max=10, step=1, description="Descriptor Label"))
display(widgets.Checkbox(value=False, description="Descriptor Label"))
```



```
from ipywidgets import widgets
my_btn = widgets.Button(description="Click Me!")
text_area = widgets.Text(placeholder="Enter some text...")

def print_me(unused):
    print(text_area.value)

my_btn.on_click(print_me)
display(text_area)
display(my_btn)
```




```
Here's a thing
And a second thing
More thingz!
```

```
from ipywidgets import widgets
my_btn = widgets.Button(description="Click Me!")
text_area = widgets.Text(placeholder="Enter some text...")

def print_me(unused):
    print(text_area.value)

my_btn.on_click(print_me)
display_box = widgets.HBox([text_area, my_btn])
display(display_box)
```


```
Nicely aligned!
```

Widgets Lab - Maybe

- Text input
- Dropdown input
- Button submission

Sockets / Threading

Network communication is achieved through the use of "sockets", which are IP-Port pairs. Using these, servers can specify their listening status, and clients can reach out to try and connect. Communication can also be done through either TCP or UDP connections, depending on the service you're attempting to communicate with.

Threading is the practice of splitting your program into threads. Each thread will run on its own, allowing private sessions of similar code to run at the same time.

```python
import socket
import threading

_HOST = "127.0.0.1"
_PORT = 31337

def handle_connection(conn, addr):
    print("Connection accepted from: {}".format(addr))
    conn.send("Hello! Welcome to my chat server!".encode())
    resp = conn.recv(1024).decode()
    while resp != "exit" and resp != "quit":
        conn.send(f"You said: {resp}".encode())
        resp = conn.recv(1024).decode()
    conn.send("Bye now!".encode())
    conn.close()
    print("Connection terminated with: {}".format(addr))

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((_HOST, _PORT))
while True:
    print("Awaiting new connection...")
    s.listen()
    argz = s.accept()
    t = threading.Thread(target=handle_connection, args=argz)
    t.start()
```

Server code

```python
import socket

_HOST = "127.0.0.1"
_PORT = 31337

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((_HOST, _PORT))

while True:
    print(s.recv(1024).decode())
    msg = input("What would you like to say? ")
    s.send(msg.encode())
    if msg == "quit" or msg == "exit":
        print(s.recv(1024).decode())
        s.close()
        break

print("Done!")
```

Client code

```
C:\Users\Ben\Documents\Python Stuff>python sockets_server.py
Awaiting new connection...
Connection accepted from: ('127.0.0.1', 53419)
Awaiting new connection...
Connection terminated with: ('127.0.0.1', 53419)
```

```
C:\Users\Ben\Documents\Python Stuff>python sockets_client.py
Hello! Welcome to my chat server!
What would you like to say? Hello chat Server!
You said: Hello chat Server!
What would you like to say? quit
Bye now!
Done!
```

```
C:\Users\Ben\Documents\Python Stuff>python sockets_server.py
Awaiting new connection...
Connection accepted from: ('127.0.0.1', 53933)
Awaiting new connection...
Connection accepted from: ('127.0.0.1', 53934)
Awaiting new connection...
Connection accepted from: ('127.0.0.1', 53935)
Awaiting new connection...
```

Sockets / Threading lab

- Server startup
- Client connection
- Client interaction
- Client disconnect / server shutdown

Databases

Databases are the backbone of the information world; when you want easy access to data, as well as associations with other data, you want to look at using a database.

Databases provide organization and structure over your data, as well as allowing you to use special querying languages to access the specific data you want quickly and easily.

The two main types of databases are relational and non-relational, each with their own advantages and disadvantages, as well as slight changes in terminology and grammar. Python naturally supports SQLite3, a relational database.

```python
import sqlite3

conn = sqlite3.connect('example.db')
conn.execute('drop table if exists test_table')
conn.execute('create table test_table (data text, value int, byte_data blob)')
byts = b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09'
conn.execute('insert into test_table values ("My Test String", 1, ?)', [byts])
conn.execute('insert into test_table (data, value) values ("Another String", 1)')
conn.commit()

curr = conn.execute('select * from test_table where value = 1')
res = curr.fetchall()

curr.close()
conn.close()

for r in res:
    print(r)
```

```
C:\Users\Ben\Documents\Python Stuff>python database_example.py
('My Test String', 1, b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t')
('Another String', 1, None)

C:\Users\Ben\Documents\Python Stuff>
```

Databases Lab

- Setup simple database
- Enter data
- Load data

Classes

Classes are used to organize data alongside functionality. A Class also represents a certain 'type' of data which may not be easily expressed with primitive types.

Since classes are so open-ended and have large creative bounds, examples will be best to illustrate what is meant:

```
class Car:
    numWheels = 4
    hasEngine = True

    def __init__(self, numDoors, gearStyle):
        if isinstance(numDoors, int):
            self.numDoors = numDoors

        if gearStyle == 'Automatic' or gearStyle == 'Manual':
            self.gearStyle = gearStyle
        else:
            self.gearStyle = 'Unknown'
        self.speed = 0
        self.distance = 0
        self.time = 0

    def setSpeed(self, speed):
        self.speed = speed

    def drive(self, time):
        self.time += time
        self.distance += time * self.speed
```

```
C:\Users\Ben\Documents\Python Stuff>python -i classes.py
>>> c = Car(2, 'Manual')
>>> c.drive(10)
>>> c.time
10
>>> c.distance
0
>>> c.speed
0
>>> c.speed = 25
>>> c.speed
25
>>> c.drive(5)
>>> c.distance
125
>>> c.time
15
>>>
```

Inheritance

Inheritance is the method of having "child" classes inherit from "parent" classes. This is how classes which are the same in many ways, but differ in a few, can maintain easy relationships and potential updates.

This also allows something called "abstraction"; when you're aware that whatever object you have will behave in a cewrtain way, but not specifically how it will be done, you're "abstracting". For instance, the difference between an Array and a LinkedList, when attempting to iterate.

```python
import math

class Shape:
    def area(self):
        pass

    def perimeter(self):
        pass
```

```python
class Square(Shape):
    numSides = 4

    def __init__(self, size):
        self.size = size

    def perimeter(self):
        return self.size * 4

    def area(self):
        return self.size * self.size
```

```python
class Triangle(Shape):
    numSides = 3

    def __init__(self, side1, side2, side3):
        self.side1 = side1
        self.side2 = side2
        self.side3 = side3

    def perimeter(self):
        return self.side1 + self.side2 + self.side3

    def area(self):
        s = (self.side1 + self.side2 + self.side3)/2
        inner = s * (s - self.side1) * (s - self.side2) * (s - self.side3)
        res = math.sqrt(inner)
        return res
```

Class Lab

- Car class
- Animal superclass
- Dog / Bird subclasses

Dunder Methods

'Dunder' means 'double underscore', referring to the characters that begin and end these types of functions

Dunder methods are usually meta in nature, that is, they refer to how the Class interacts with the program; for instance, determining how an object looks as a string, or how to compare your class for equality, etc.

The most common one is __init__, which is the constructor, but there are other prominent methods.

```python
class DunderClass:
    def __init__(self):
        print("Calling the 'init' method...")

    def __repr__(self):
        return "'Official' str representation"

    def __str__(self):
        return "Standard str representation"
```

```
C:\Users\Ben\Documents\Python Stuff>python -i dunder_classes.py
>>> d = DunderClass()
Calling the 'init' method...
>>> print(str(d))
Standard str representation
>>> print(d.__str__())
Standard str representation
>>> print(repr(d))
'Official' str representation
>>> print(d.__repr__())
'Official' str representation
```

```python
class CustomClass:
    def __init__(self, val):
        self.val = val

    def __add__(self, other):
        if isinstance(other, CustomClass):
            if type(self.val) == type(other.val):
                return self.val + other.val
            elif isinstance(self.val, str) and isinstance(other.val, int):
                return self.val + str(other.val)
            elif isinstance(self.val, int) and isinstance(other.val, str):
                return str(self.val) + other.val
            else:
                raise Exception("Unable to add value types")
        else:
            raise Exception("Invalid operands")

    def __eq__(self, other):
        return type(self.val) == type(other.val) and self.val == other.val
```

```
C:\Users\Ben\Documents\Python Stuff>python -i dunder_classes.py
>>> c1 = CustomClass("Hello")
>>> c2 = CustomClass("World")
>>> c3 = CustomClass(35)
>>> c1.val
'Hello'
>>> c1 + c2
<__main__.CustomClass object at 0x0000018AE8B4E488>
>>> c1 += c2
>>> c1.val
'HelloWorld'
>>> c1 += c3
>>> c1.val
'HelloWorld35'
>>> c4 = CustomClass(3.14)
>>> c1 += c4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "dunder_classes.py", line 24, in __add__
    raise Exception("Unable to add value types")
Exception: Unable to add value types
```

Dunder Methods Lab

- Modify previous class methods
- (le, ge, neg)

Questions?