Python: Bonus

Anthology of additional topics

Function +

Extra, built-in features for functions

Lambda Functions

Lambda functions are small, anonymous functions defined for simple return processes

lambda args : return_expression

```
1  def triple(x):
2      return x*3
3  print(triple(4))
```

```
1  triple = lambda x: x*3
2  print(triple(4))
```

Lambda Functions (Multiple Inputs)

Lambda functions are small, anonymous functions defined for simple return processes

lambda args : return_expression

```
1  def product (x, y):
2      return x*y
3  print(product(2, 4))
```

```
1  product = lambda x, y: x*y
2  print(product(2, 4))
```

Quick Exercise: Lambda Conversion

Convert the following regular function into a lambda function

```
1  def distance(x, y):
2      return (x**2 + y**2)**(1/2)
```

Test the function by calculating the following values

```
4  first_distance = distance(3, 4)
5  second_distance = distance(6, 8)
```

Is using a lambda preferable for this case?

Map Function

The map function applies a given function to every item in a collection of item (like a list)

```
1  def squared(x):
2      return x**2
```

```
4  numbers = [1, 2, 3, 4, 5]
5  numbers_squared = map(squared, numbers)
6  print(list(numbers_squared))
```

Map Function (with Lambdas)

The map function applies a given function to every item in a collection of item (like a list)

```
1 numbers = [1, 2, 3, 4, 5]
2 numbers_squared = map(lambda x: x**2, numbers)
3 print(list(numbers_squared))
```

Quick Exercise: Cost Cutting

Divide every item in the given *cost* by two

```
1 cost = [10_000, 200, 31, 45, 1]
2 cost_half = ...
3 print(cost_half)
```

Filter Function

The filter function keeps the items in a collection of item (like a list) if satisfies a function

```
1  def is_even(x):
2      return x % 2 == 0
```

```
4  numbers = [1, 2, 3, 4, 5]
5  numbers_even = filter(is_even, numbers)
6  print(list(numbers_even))
```

Filter Function (with Lambdas)

The filter function keeps the items in a collection of item (like a list) if satisfies a function

```
1 numbers = [1, 2, 3, 4, 5]
2 numbers_even = filter(lambda x: x % 2 == 0, numbers)
3 print(list(numbers_even))
```

Quick Exercise: Top Performers

Given the following ***scores***, keep the ***scores*** that are greater than 6

```
1 scores = [10, 7, 5, 3, 5, 8]
2 scores_top = ...
3 print(scores_top)
```

Decorator

A decorator is a function that modifies another function without changing its code

```
1  def decorator(function):
2      def wrapper():
3          print("Before the function runs...")
4          function()
5          print("After the function runs...")
6      return wrapper
```

```
7  @decorator
8  def say_hello():
9      print("Hello World")
10
11 say_hello()
```

Decorators for functions with return

Function returns need to be returned by the wrapper as well

```
1  def spaced(function):
2      def wrapper():
3          original = function()
4          return "\n" + original + "\n"
5      return wrapper
6
7  @spaced
8  def message():
9      return "Good morning"
10
11 print(message())
```

Decorator for Function with Inputs

Note: To accept a dynamic amount of inputs (zero to infinite), use `*args, **kwargs`

```
1  def check_authentication(function):
2      def wrapper(user):
3          if user != "admin":
4              print("Access denied!")
5          else:
6              function(user)
7      return wrapper
```

```
8  @check_authentication
9  def access_database(user):
10     print("Accessing database...")
11 access_database("user")
```

Decorators with input (Decorator Factory)

```
def repeat(n):
    def decorator(function):
        def wrapper(message):
            result = function(message)
            return result * n
        return wrapper
    return decorator
```

```
@repeat(3)
def greeting(message):
    return f"Hello, {message}! "
```

Quick Exercise: Console Input

Given functions that return a string

```
1  def get_command():
2      return input("Command: ")
3
4  def get_user():
5      return input("User: ")
```

```
>>> Command: user input 1
You inputted user input 1
```

```
>>> User: user input 2
You inputted user input 2
```

Modify the return of each function with a ***decorator*** to add >>> before the text and prints the input the user provided immediately

Challenge: Function input

Partial Function

The partial function creates a new function by partially filling some of the inputs

```
1  from functools import partial
2
3  def multiply(x, y):
4      return x * y
5
6  double = partial(multiply, y=2)
7  triple = partial(multiply, y=3)
8
9  print(double(100) + triple(20))
```

```
260
```

Quick Exercise: Fee Application

```
1  def apply_fee(amount, fee, label):
2      new_total = amount + fee(amount)
3      print(f"{label}: {new_total:.2f}")
4      return new_total
```

Create new functions from `apply_fee` with the following conditions

1. VAT: +12%
2. Service Fee: +100
3. Discount: -10%

Single Dispatch

The single dispatch decorator allows overriding for the same function name based on type

```python
from functools import singledispatch

@singledispatch
def combine(a, b):
    raise TypeError()

@combine.register(list)
def _(a, b):
    return a + b
@combine.register(dict)
def _(a, b):
    return {**a, **b}
@combine.register(set)
def _(a, b):
    return a | b
```

```python
x = [1, 2]
y = [3, 4]
print(combine(x, y))

x = {1, 2}
y = {3, 4}
print(combine(x, y))

x = {"a": 1}
y = {"b": 2}
print(combine(x, y))
```

Quick Exercise: Short Description

```python
from functools import singledispatch

@singledispatch
def summarize(data):
    """Default summarizer"""
    print(f"Default: just printing {x}")
```

Create new functions for the following data types:

1. str: Print character and word count
2. dict: Print key counts
3. list: Print item count, first and last
4. set: Print item count, min and max

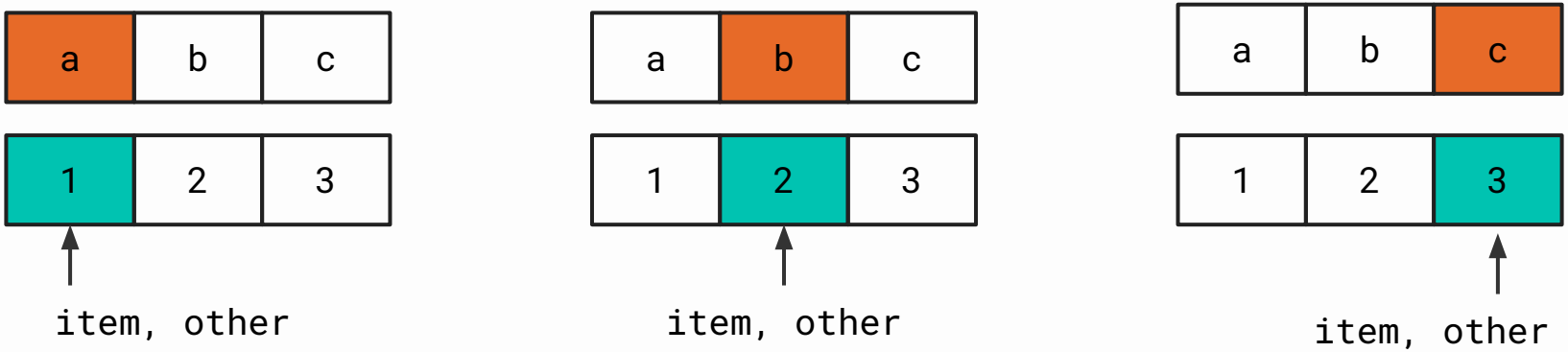
Unpacking

Expand the mechanics of loop unpacking

Multiple Looping

You can access two items at once from two different sequences using the zip function

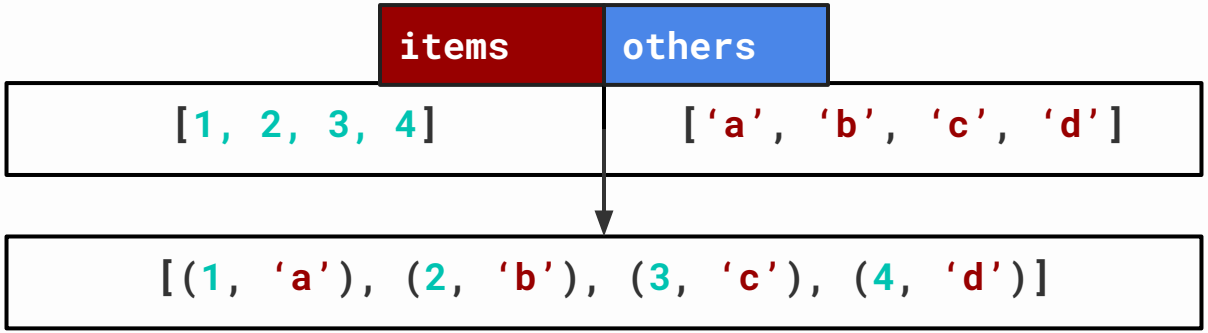
```
1  items = ("a", "b", "c")
2  others = (1, 2, 3)
3  for item, other in zip(items, others):
4      print(item, other)
```



Zip Function Contents

The **zip** function creates a list of tuples from all of its parameters

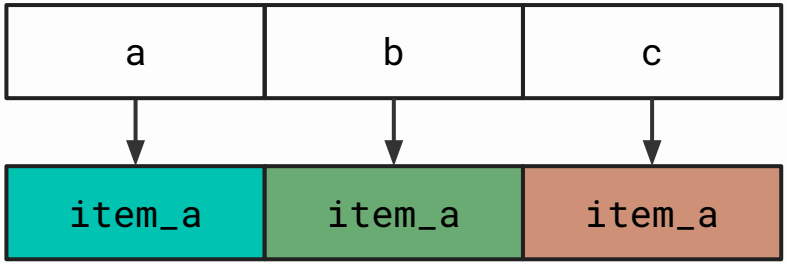
```
1  items = ("a", "b", "c")
2  others = (1, 2, 3)
3  zipped = zip(items, others)
4  print(list(zipped))
```



Tuple Unpacking

Because tuples have a fixed size, Python added an unpacking feature for convenience

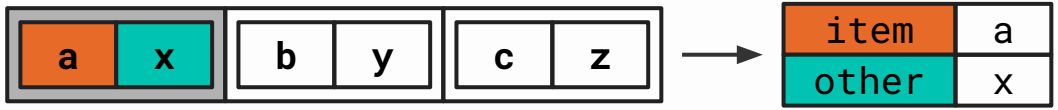
```
1  items = ('a', 'b', 'c')
2  item_a, item_b, item_c = items
```



Unpacking in Loops

You can access two items at once from two different sequences using the zip function

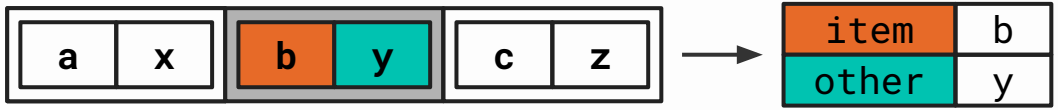
```
1  items = ("a", "b", "c")
2  others = (1, 2, 3)
3  for item, other in zip(items, others):
4      print(item, other)
```



Unpacking in Loops

You can access two items at once from two different sequences using the zip function

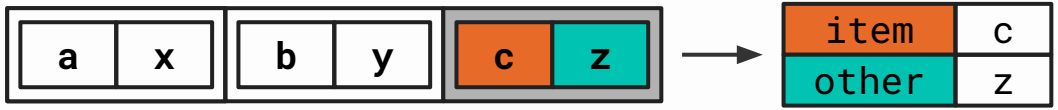
```
1  items = ("a", "b", "c")
2  others = (1, 2, 3)
3  for item, other in zip(items, others):
4      print(item, other)
```



Unpacking in Loops

You can access two items at once from two different sequences using the zip function

```
1  items = ("a", "b", "c")
2  others = (1, 2, 3)
3  for item, other in zip(items, others):
4      print(item, other)
```



Enumerate Looping

You can loop through a sequence of items and get their position using the enumerate function.

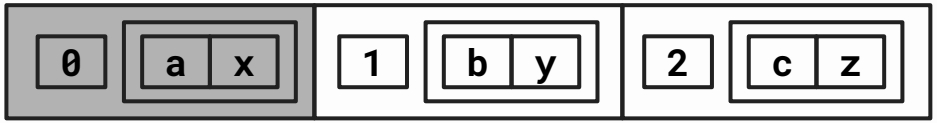
```
1 items = ("a", "b", "c")
2 for index, item in enumerate(items):
3     print(index, item)
```

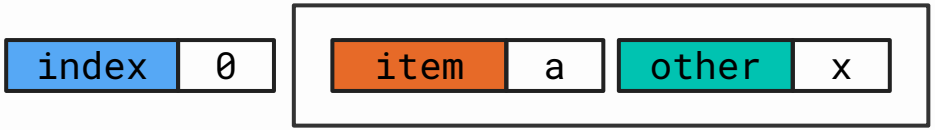
```
0 a
1 b
2 c
```

Nested Unpacking

For inner tuples inside another tuple, denote using parentheses

```
1  items = ("a", "b", "c")
2  others = (1, 2, 3)
3  for index, (items, other) in enumerate(zip(items, others)):
4      print(index, item, other)
```

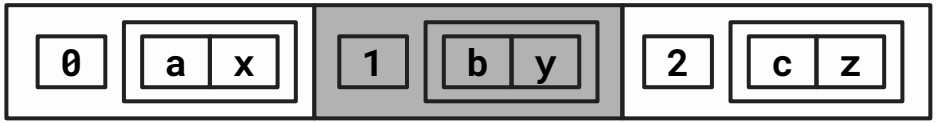


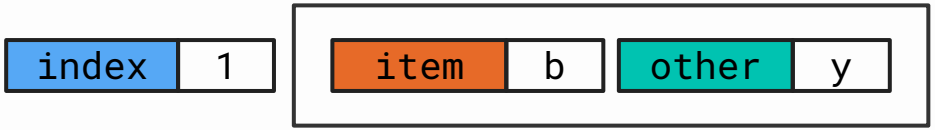


Nested Unpacking

For inner tuples inside another tuple, denote using parentheses

```
1  items = ("a", "b", "c")
2  others = (1, 2, 3)
3  for index, (items, other) in enumerate(zip(items, others)):
4      print(index, item, other)
```

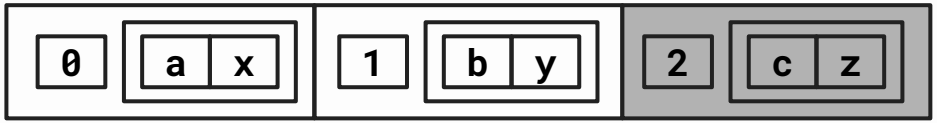


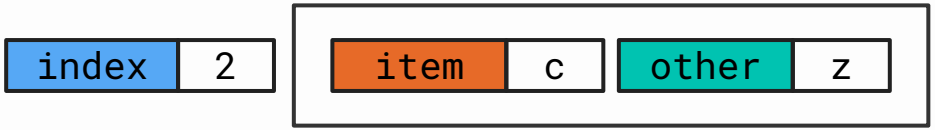


Nested Unpacking

For inner tuples inside another tuple, denote using parentheses

```
1  items = ("a", "b", "c")
2  others = (1, 2, 3)
3  for index, (items, other) in enumerate(zip(items, others)):
4      print(index, item, other)
```





Pair Unpacking

For inner tuples inside another tuple, denote using parentheses

```
1  dict1 = {'a': 1, 'b': 2}
2  dict2 = {'c': 10, 'd': 20}
3
4  for (k1, v1), (k2, v2) in zip(dict1.items(), dict2.items()):
       print(k1, v1, k2, v2)
```

Iteration

More complex iteration

Repeat Function

The repeat function generates an item multiple times. If second input is given, it's infinite.

```
1 from itertools import repeat
2
3 repeated = repeat("Hello", 3)
4 print(list(repeated))
```

```
["Hello", "Hello", "Hello"]
```

Quick Exercise: Functional Square

```
1  from itertools import repeat
2
3  # Use map, pow, and repeat to create the squares of nums
4  nums = [2, 3, 4, 5]
5  squares = None
6
7  print(squares)
```

Accumulate Function (Default)

The accumulate function applies a function (sum by default) to an increasing list

```
1  from itertools import accumulate
2
3  accumulation = accumulate([1, 2, 3, 4])
4  # sum([1]) -> 1
5  # sum([1, 2]) -> 3
6  # sum([1, 2, 3]) -> 6
7  # sum([1, 2, 3, 4]) -> 10
8  print(list(accumulation))
```

```
[1, 3, 6, 10]
```

Accumulate Function (Custom)

The accumulate function applies a function to an increasing list

```
1  from itertools import accumulate
2
3  running_max = accumulate([3, 1, 4, 2, 5], max)
4  # max([3]) -> 3
5  # max([3, 1]) -> 3
6  # max([3, 1, 4]) -> 4
7  # max([3, 1, 4, 2]) -> 4
8  # max([3, 1, 4, 5]) -> 5
   print(list(running_max))
```

```
[3, 3, 4, 4, 5]
```

Quick Exercise: Running Balance

```
1  from itertools import accumulate
2
3  # Generate the running balance
4  transactions = [10_000, -500, 3_000, -2_500, -1_000]
5  print(transactions)
```

Batched Function

The batched function groups an iterable into sub-iterables by a fixed max amount

```
1  from itertools import batched
2
3  numbers = range(10)
4  batched_numbers = batched(numbers, 3)
5
6  print(list(batched_numbers))
```

```
[(0, 1, 2), (3, 4, 5), (6, 7, 8), (9,)]
```

Quick Exercise: Random Groupings

```
from itertools import batched
from random import choice
import string

def random_name(length=6):
    letters = string.ascii_lowercase
    return ''.join(choice(letters) for _ in range(length))

names = [random_name() for _ in range(40)]

# Create group of fives
groups = []
print(groups)
```

Chain Function

The chain function appends another iterable at the end of another

```
1  from itertools import chain
2
3  a = [1, 2, 3]
4  b = ['x', 'y', 'z']
5
6  print(list(chain(a, b)))
```

```
[1, 2, 3, 'x', 'y', 'z']
```

Iterable Chain Function

The best use case of chain is flattening nested iterables into a single dimension

```
1  from itertools import chain
2
3  nested = [[1, 2], [3, 4], [5]]
4
5  print(list(chain.from_iterable(nested)))
```

```
[1, 2, 3, 4, 5]
```

Product Function (Nested Loop)

```
1 colors = ["red", "blue"]
2 sizes = ["S", "M", "L"]
```

```
3 from itertools import product
4
5 items = product(colors, sizes)
6 print(list(items))
7
8
```

```
products = []
for color in colors:
    for size in sizes:
        item = (color, size)
        products.append(item)
print(products)
```

```
[
    ('red', 'S'), ('red', 'M'),
    ('red', 'L'), ('blue', 'S'),
    ('blue', 'M'), ('blue', 'L')
]
```

Quick Exercise: Complete Cards

```
1  # Create a standard 52 deck of cards
2  cards = []
```

Match

Structural pattern matching in Python

Match Case - Literal Matching

Match case statements is a more concise yet powerful alternative to if-elif-else statements

```
1  match variable:
2      case value_1:
3          # Process
4      case value_2:
5          # Process
6      case _:
7          # Process
```



```
1
2  if variable==value_1:
3      # Process
4  elif variable==value_2:
5      # Process
6  else:
7      # Process
```

Match Case - Literal Matching

```
1  you_said = input("You said: ")
2
3  match you_said:
4      case "Wish":
5          print("107.5")
6      case "Hello":
7          print("...it's me")
8      case "Jopay":
9          print("...kamusta ka na")
10     case "Black Pink":
11         print("...in your area")
12     case _:
13         print("I don't know that song!")
```

Quick Exercise: Complete the Count

Create a simple console counter based on the user input

```
count = 0
count_complete = False

# Based on the command, add by one, subtract by one, or end
while not count_complete:
    command = input("Command (up, down): ")
    # Add code here

print("Final count:", count)
```

Multiple Literal Matching

Match case statements support multiple literal matching using the | operator

```
1  def menu(command):
2      match command:
3          case "add" | "create" | "new":
4              print("Adding new song...")
5          case "remove" | "delete":
6              print("Removing given song...")
7          case "play" | "start":
8              print("Playing the first song...")
9          case "show":
10             print("Showing all songs...")
11         case _:
12             print("Unknown command...")
```

Quick Exercise: Multi- Count

Update the previous exercise to support multiple formats for the same command

```
count = 0
count_complete = False

# Based on the command, add by one, subtract by one, or end
while not count_complete:
    command = input("Command (+, -, up, down, add, sub): ")
    # Add code here

print("Final count:", count)
```

Match Case - Variable Capture

Match cases can also reuse the variable instead of dereferencing it in the default case

```
1  match variable:
2      case value_1:
3          # Process
4      case value_2:
5          # Process
6      case new_variable:
7          # Process
```

```
1
2  if variable==value_1:
3      # Process
4  elif variable==value_2:
5      # Process
6  else:
7      new_variable = variable
8      # Process
```

Match Case - Variable Capture Example

```
1  you_said = input("You said: ")
2
3  match you_said:
4      case "Wish":
5          print("107.5")
6      case "Hello":
7          print("...it's me")
8      case "Jopay":
9          print("...kamusta ka na")
10     case "Black Pink":
11         print("...in your area")
12     case song:
13         print("I don't know", song)
```

Match Case - Conditional Matching

Given the following ***scores***, keep the ***scores*** that are greater than 6

```
match variable:
    case value_1:
        # Process
    case value_2 if condition:
        # Process
    ...
    case _:
        # Process
```

```

if variable==value_1:
    # Process
elif condition:
    # Process
...
else:
    # Process
```

Match Case - Conditional Example

```
1  def http_status(status_code):
2      match status_code:
3          case 200:
4              return "OK"
5          case 404:
6              return "Not Found"
7          case 500:
8              return "Internal Server Error"
9          case code if 400 <= code < 500:
10             return "Client Error"
11         case code if 500 <= code < 600:
12             return "Server Error"
13         case code:
14             return f"Unknown Status - {code}"
```

Quick Exercise: Dispute Count

Update the previous exercise to prevent subtraction if count is already zero

```
count = 0
count_complete = False

# Based on the command, add by one, subtract by one, or end
while not count_complete:
    command = input("Command (+, -, up, down, add, sub): ")
    # Add code here

print("Final count:", count)
```

Match with Classes

Match also works with classes by distinguishing using the class and attributes

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
def location(point):
    match point:
        case Point(0, 0):
            return "Origin"
        case Point(0, y):
            return f"On Y axis at {y}"
        case Point(x, 0):
            return f"On X axis at {x}"
        case Point(x, y):
            return f"Point at ({x}, {y})"
```

Match with Classes

Match also works with classes by distinguishing using the class and attributes

```
...
match point:
    case Square():
        return "Perfect square"
    case Rectangle(width=w, height=h) if w == h:
        return f"Rectangle (Perfect Square)"
    case Rectangle(width=w, height=h):
        return f"Normal rectangle {w}x{h}"
    case Circle():
        return "Circle"
    case _:
            return "Unknown shape"
```

OpenPyXL

Lightweight library for reading xlsx and xlsm files

Creating a Workbook

In OpenPyXL, an entire Excel file is represented using the **Workbook** class. All of the data processes (loading, saving, editing), sheet handling, and cell management is done here.

```
1  from openpyxl import Workbook
2
3  workbook = Workbook()
4
5
6
7  workbook.save("sample.xlsx")
```

Default Worksheet

Accessing a worksheet is done using indexing. By default, a new workbook has a starting sheet with the title **"Sheet"**

```
1  from openpyxl import Workbook
2
3  workbook = Workbook()
4  sheet = workbook["Sheet"]
5
6
7  workbook.save("sample.xlsx")
```

Creating a Worksheet

A `Workbook` object can use the `create_sheet(str)` method to create a new sheet. It gets added at the end by default. If you want to set the index, use `create_sheet(str, int)`.

```
1  from openpyxl import Workbook
2
3  workbook = Workbook()
4  sheet = workbook["Sheet"]
5  workbook.create_sheet("Additional")
6
7  workbook.save("sample.xlsx")
```

Editing a Cell

Accessing a worksheet is done using indexing. The key depends on the coordinate used in Excel workbooks

```
1  from openpyxl import Workbook
2
3  workbook = Workbook()
4  sheet = workbook["Sheet"]
5  workbook.create_sheet("Additional")
6  sheet["A1"] = "Hello"
7  workbook.save("sample.xlsx")
```

Loading a Workbook

You can also load existing Excel files using the `load_workbook` helper function.

```
1  from openpyxl import load_workbook
2
3  workbook = load_workbook("sample.xlsx")
```

Cell Management

Example operations and methods for cell read and writes

Read-Write Cells

Cells inside worksheets can either be accessed using indexing or the **Cell** interface.

```
1  from openpyxl import load_workbook
2
3  workbook = load_workbook("sample.xlsx")
4  sheet = workbook["Additional"]
5
6  sheet["A1"] = "Tickets"
7  print(sheet["A1"].value)
8
9  cell = sheet.cell(row=1, column=2)
10 cell.value = 100
11 print(cell.value)
12
13 workbook.save("sample.xlsx")
```

Multiple Cell Write

There is no dedicated method for writing in multiple cells at once. Instead, the expected approach is to use a standard loop

```
1  from openpyxl import load_workbook
2
3  workbook = load_workbook("sample.xlsx")
4  sheet = workbook["Additional"]
5
6  tickets = {"HR": 30, "Legal": 23, "Sales": 34, "Admin": 13}
7
8  for i, (group, count) in enumerate(tickets.items(), start=3):
9      sheet.cell(row=i, column=1).value = group
10     sheet.cell(row=i, column=2).value = count
11
12 workbook.save("sample.xlsx")
```

Multiple Cell Write (Ranges)

Worksheets support Excel-based formulas for getting items. This allows cell-based coding.

```
1  from openpyxl import load_workbook
2
3  workbook = load_workbook("sample.xlsx")
4  sheet = workbook["Additional"]
5
6  tickets = {"HR": 30, "Legal": 23, "Sales": 34, "Admin": 13}
7
8  ticket_and_cells = zip(tickets.items(), sheet["A3:B6"])
9
10 for (group, count), (group_cell, count_cell) in ticket_and_cells:
11     group_cell.value = group
12     count_cell.value = count
13
14 workbook.save("sample.xlsx")
```

Multiple Cell Append

While OpenPyXL doesn't support writing on ranges directly, it allows appends.

```
1  from openpyxl import load_workbook
2  workbook = load_workbook("sample.xlsx")
3  sheet = workbook["Additional"]
4
5  new_data = ["Tech", 300]
6  sheet.append(new_data)
7
8  workbook.save("sample.xlsx")
```

Multiple Cell Read

Each **Worksheet** object has an `iter_rows` method to loop or iterate through all of the cells. Each row is a tuple of **Cell** objects.

```
1  from openpyxl import load_workbook
2
3  workbook = load_workbook("sample.xlsx")
4  sheet = workbook["Additional"]
5
6  for row in sheet.iter_rows():
7      print(row)
```

Multiple Cell Read (Unpacked)

If there are only a few number of columns, you can directly assign the values to variables similar to how **`enumerate`** and **`zip`** operates.

```
1  from openpyxl import load_workbook
2
3  workbook = load_workbook("sample.xlsx")
4  sheet = workbook["Additional"]
5
6  for header, item in sheet.iter_rows():
7      print(header.value, item.value)
```

Multiple Cell Read (Bounded)

The **iter_rows** method can change where it starts and ends using the min_row, and max_col optional parameters. The default is the first row and the last row with a value.

```
1  from openpyxl import load_workbook
2
3  workbook = load_workbook("sample.xlsx")
4  sheet = workbook["Additional"]
5
6  for header, item in sheet.iter_rows(min_row=3, max_row=6):
7      print(header.value, item.value)
```

```
tip: you can use sheet.max_row and max.column
```

Quick Exercise: Product Orders

Create a new sheet called **`Order`** in **`samples.xlsx`** and generate the following data

Category	Brand	Unit
Laptop	HP	1
Laptop	HP	2
Laptop	Acer	3
Laptop	Acer	4
Monitor	HP	1
Monitor	HP	2
Monitor	Acer	3
Monitor	Acer	4

Cell+

Adding styling and rules for the cell layouts

Cell Font

`Cell` objects have the `font` property that can be changed to add font-specific styling

```
1  from openpyxl import load_workbook
2  from openpyxl.styles import Font
3
4  workbook = load_workbook("sample.xlsx")
5  sheet = workbook["Additional"]
6
7  sheet["A1"].font = Font(name="Arial", size=20)
8  workbook.save("sample.xlsx")
```

Cell Font (Options)

`Cell` objects have the `font` property that can be changed to add styling

Property	Description
name	`'Calibri', 'Arial', 'Times New Roman', etc. (system-based)`
size	`float/int`
bold	`bool`
italic	`bool`
underline	`'single', 'double', 'singleAccounting', 'doubleAccounting', None/False`
strike	`bool`
color	**`Hex Codes:`** `'FF0000' (Red), '00FF00' (Green), '000000' (Black), etc.`

Cell Pattern Fill

Cell objects have the `fill` property that can be changed to add background styling

```
1  from openpyxl import load_workbook
2  from openpyxl.styles import PatternFill
3
4  workbook = load_workbook("sample.xlsx")
5  sheet = workbook["Additional"]
6
7  for (cell,) in sheet["A3:A7"]:
8      cell.fill = PatternFill(fill_type='solid', fgColor='4F81BD')
9
10 workbook.save("sample.xlsx")
```

Cell Pattern Border and Side

Cell objects have the **border** property that can be changed to add border styling

```
1  from openpyxl import load_workbook
2  from openpyxl.styles import Side, Border
3
4  workbook = load_workbook("sample.xlsx")
5  sheet = workbook["Additional"]
6
7  ss = Side(style="thin", color='000000')
8
9  for (cell,) in sheet["A3:A7"]:
10     cell.border = Border(left=ss, right=ss, top=ss, bottom=ss)
11
12 workbook.save("sample.xlsx")
```

Cell Side (Options)

`Side` objects have the following styles to choose from

Property	Description
style	`'thin', 'medium', 'thick', 'dashed', 'dotted', 'double', 'hair', 'mediumDashed', 'slantDashDot'`
color	**`Hex Codes:`** `‘FF0000’ (Red), ‘00FF00’ (Green), ‘000000’ (Black), etc.`

Cell Alignment

Cell objects have the **alignment** property that can be changed for text formatting

```
1  from openpyxl import load_workbook
2  from openpyxl.styles import Alignment
3
4  workbook = load_workbook("sample.xlsx")
5  sheet = workbook["Additional"]
6
7  for (cell,) in sheet["A3:A7"]:
8      cell.alignment = Alignment(
9          horizontal='center', vertical='center',
10         wrap_text=True, shrink_to_fit=True,
11         indent=1
12     )
13
14 workbook.save("sample.xlsx")
```

Cell Alignment (Options)

The properties in the `Alignment` class have the following options

Property	Description
horizontal	`'left', 'right', 'center', 'justify'`
vertical	`'top', 'center', 'bottom'`

Cell Number Format

Cell objects have the **alignment** property that can be changed for text formatting

```
1  from openpyxl import load_workbook
2
3  workbook = load_workbook("sample.xlsx")
4  sheet = workbook["Additional"]
5
6  sheet["B1"].number_format = '#,##0'
7  workbook.save("sample.xlsx")
```

Date Format	`'mm/dd/yyyy'`
Time	`'hh:mm:ss'`
Percentage	`'0%'`
Decimal	`'0.00'`

Quick Exercise: Product Orders (Styled)

Follow the styling below for the **`Order`** sheet in **`samples.xlsx`**

Category	Brand	Unit
Laptop	HP	1
Laptop	HP	2
Laptop	Acer	3
Laptop	Acer	4
Monitor	HP	1
Monitor	HP	2
Monitor	Acer	3
Monitor	Acer	4

Protection

Adding write safety to the worksheet

Sheet Protection (Specific)

```
1  from openpyxl import load_workbook
2
3
4  workbook = load_workbook("sample.xlsx")
5  sheet = workbook["Additional"]
6  sheet.protection.sheet = True
7
8
9
10
11 workbook.save("secured.xlsx")
12
13
14
15
16
```

Sheet Protection (Specific)

```
1  from openpyxl import load_workbook
2  from openpyxl.styles import Protection
3
4  workbook = load_workbook("sample.xlsx")
5  sheet = workbook["Additional"]
6  sheet.protection.sheet = True
7
8  for (cell,) in sheet["B2:B7"]:
9      cell.protection = Protection(locked=False)
10
11 workbook.save("secured.xlsx")
12
13
14
15
16
```

Data Validation (Contains)

Category-based (finite type of strings) can be limited using the **DataValidation** class

```
1  from openpyxl import load_workbook
2  from openpyxl.worksheet.datavalidation import DataValidation
3
4  workbook = load_workbook("sample.xlsx")
5  sheet = workbook["Order"]
6
7  options_str = '"Laptop,Monitor,Peripheral"'
8  dv = DataValidation(type="list", formula1=options_str)
9
10 sheet.add_data_validation(dv)
11 dv.add("A2:A100")
12 workbook.save("sample.xlsx")
```

Deletion

How to remove or clear out values

Sheet Deletion

Remove a sheet can be done directly using the **`del`** operator

```
1  from openpyxl import load_workbook
2
3  workbook = load_workbook("sample.xlsx")
4  del workbook["Sheet"]
5
6  workbook.save("sample.xlsx")
```

Cell Deletion

There is no direct way to delete cells since it works on a reference basis but you can clear it

```
1  from openpyxl import load_workbook
2
3  workbook = load_workbook("sample.xlsx")
4  sheet = workbook["Additional"]
5  sheet["A1"] = None
6  sheet["B1"] = None
7
8  workbook.save("sample.xlsx")
```

Row Deletion

There is no direct way to delete cells since it works on a reference basis but you can clear it

```
1  from openpyxl import load_workbook
2
3  workbook = load_workbook("sample.xlsx")
4  sheet = workbook["Additional"]
5  sheet.delete_rows(1)
6  sheet.delete_rows(1)
7
8  workbook.save("sample.xlsx")
```

Quick Exercise: Dummy Logs

Create a new workbook **tickets.xlsx**. In sheet **Tickets**, create **10_000** random entries

```
from random import randint, choice, seed
from datetime import datetime, timedelta

seed(123)

# Example of how to generate random values for a row
status = choice(["New", "Ongoing", "Done", "Close”, None])
priority = choice(["Low", "Medium", "High", None])
department = choice(["HR", "Legal", "sales ", "Adm", "Tech"])
points = randint(1, 100)
votes = randint(1, 10)
start = datetime(2023, 5, 1) + timedelta(hours=randint(0, 2000))
end = start + timedelta(hours=randint(0, 2000))
```

Quick Exercise: Dummy Accounts

Create a new workbook **accounts.xlsx**. In sheet **Logs** create 10_000 random entries

```
1  from random import randint, choice, seed
2  from datetime import datetime, timedelta
3
3  seed(123)
4
5  # Example of how to generate random values for a row
6  accounts = choice([...])
7  sector = choice([...])
8  year_established = randint(1900, 2025)
9  revenue = randint(10_000, 100_000_000_000)
10 employees = randint(1, 1_000_000)
11 office_location = choice([...])
12 subsidiary_of = choice([...])
```

Pandas

The most common technique for tabular data manipulation

Reading Data

Pandas converts tabular data to data frames that are convenient to read and access

```
1  import pandas as pd
2
3  df = pd.read_csv("tickets.csv")
4  print(df)
5  print(df.info())
6  print(df.describe())
```

```
1  import pandas as pd
2
3  df = pd.read_excel("tickets.xlsx", sheet_name="Tickets")
4  print(df)
5  print(df.info())
6  print(df.describe())
```

Dataframe Columns

Pandas makes column access very convenient using the indexing operation

```
1  import pandas as pd
2
3  df = pd.read_excel("tickets.xlsx", sheet_name="Tickets")
4  print(df.columns)
5  print(df["Priority"])
6  print(df["Priority"].unique())
7  print(df["Priority"].value_counts())
```

Dataframe New Columns

Pandas specializes in creating new columns using data from other columns

```
1  import pandas as pd
2
3  df = pd.read_excel("tickets.xlsx", sheet_name="Tickets")
4
5  df["Duration"] = df["End"] - df["Start"]
6  df["Duration"] = df["Duration"].dt.total_seconds()
7  df["Duration"] = df["Duration"] / 3600
8
9  print(df)
```

Data Processes

Common operations and methods for data preparation

Common Data Cleaning Techniques

```python
import pandas as pd

df = pd.read_excel("tickets.xlsx", sheet_name="Tickets")
df.columns = df.columns.str.strip().str.title()

df["Department"] = df["Department"].str.strip().str.title()
df["Status"].fillna("Unknown", inplace=True)
df.dropna(subset=["Priority"], inplace=True)

print(df)
```

Sorting by Column

```
1  import pandas as pd
2
3  df = pd.read_excel("tickets.xlsx", sheet_name="Tickets")
4  df.columns = df.columns.str.strip().str.title()
5
6  df["Department"] = df["Department"].str.strip().str.title()
7  df["Status"].fillna("Unknown", inplace=True)
8  df.dropna(subset=["Priority"], inplace=True)
9
10 df.sort_values(
11     by='year_established', ascending=False)
12
13 print(df)
```

Saving in a New Excel File

```
1  import pandas as pd
2
3  df = pd.read_excel("tickets.xlsx", sheet_name="Tickets")
4  df.columns = df.columns.str.strip().str.title()
5
6  df["Department"] = df["Department"].str.strip().str.title()
7  df["Status"].fillna("Unknown", inplace=True)
8  df.dropna(subset=["Priority"], inplace=True)
9
10 df.sort_values(
11     by='year_established', ascending=False)
12
13 print(df)
14 df.to_excel("tick_new.xlsx", sheet_name="Tickets", index=False)
```

Appending to an Existing Excel File

```
1  import pandas as pd
2
3  df = pd.read_excel("tickets.xlsx", sheet_name="Tickets")
4  df.columns = df.columns.str.strip().str.title()
5
6  df["Department"] = df["Department"].str.strip().str.title()
7  df["Status"].fillna("Unknown", inplace=True)
8  df.dropna(subset=["Priority"], inplace=True)
9
10 df.sort_values(
11     by='year_established', ascending=False)
12
13 print(df)
14 with pd.ExcelWriter('tickets.xlsx', mode='a') as writer:
15     df.to_excel(writer, sheet_name="Clean Tickets", index=False)
```

Pandas Filtering

```python
import pandas as pd

df = pd.read_excel("accounts.xlsx", sheet_name="Logs")

high_revenue = df[df['Revenue'] > 100_000_000]
tech_sector = df[df['Sector'] == "Technology"]

print(df)
with pd.ExcelWriter('accounts.xlsx', mode='a') as writer:
    tech_sector.to_excel(writer, sheet_name="Tech", index=False)
    high_revenue.to_excel(writer, sheet_name="Top", index=False)
```

Grouping and Aggregation

```
1  import pandas as pd
2
3  df = pd.read_excel("accounts.xlsx", sheet_name="Logs")
4
5  avg_revenue = df.groupby('Sector')['Revenue'].mean()
6  total_employees = df.groupby('Sector')['Employees'].sum()
7  sector_count = df['Sector'].value_counts()
8
9  print('Average Revenue', avg_revenue)
10 print('Total Employees', total_employees)
11 print('Sector Count', sector_count)
```

Data Visualization

Examples of all visualizations

Histogram (Number Distribution)

```
1  import matplotlib.pyplot as plt
2  import pandas as pd
3
4  df = pd.read_excel("accounts.xlsx", sheet_name="Logs")
5  df["Revenue"].hist(bins=30, color="skyblue", edgecolor="black")
6  plt.title("Revenue Distribution")
7  plt.xlabel("Revenue")
8  plt.ylabel("Frequency")
9  plt.show()
```

Bar Chart (Change Over Unit)

```
1  import matplotlib.pyplot as plt
2  import pandas as pd
3
4  df = pd.read_excel("accounts.xlsx", sheet_name="Logs")
5  df["Sector"].value_counts().plot.bar(color="orange")
6  plt.title("Companies per Sector")
7  plt.xlabel("Sector")
8  plt.ylabel("Count")
9  plt.show()
```

Pie Chart (Percent Composition)

```
1  import matplotlib.pyplot as plt
2  import pandas as pd
3
4  df = pd.read_excel("accounts.xlsx", sheet_name="Logs")
5  df["Office Location"].value_counts().head(5).plot.pie()
6  plt.title("Top 5 Office Locations (Share)")
7  plt.xlabel("Sector")
8  plt.ylabel("")
9  plt.show()
```

Box Plot (Statistics Summary)

```
1  import matplotlib.pyplot as plt
2  import pandas as pd
3
4  df = pd.read_excel("accounts.xlsx", sheet_name="Logs")
5  df.boxplot(column="Revenue", by="Sector")
6  plt.title("Revenue Distribution by Sector")
7  plt.xlabel("Sector")
8  plt.ylabel("Revenue")
9  plt.tight_layout()
10 plt.show()
```

Line Plot（Change Over Unit）

```
1  import matplotlib.pyplot as plt
2  import pandas as pd
3
4  df = pd.read_excel("accounts.xlsx", sheet_name="Logs")
5  df.groupby("Year Established")["Revenue"].mean().plot.line()
6  plt.title("Average Revenue by Year Established")
7  plt.xlabel("Year")
8  plt.ylabel("Average Revenue")
9  plt.show()
```

Stacked Bar Chart (Composition + Growth)

```
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_excel("accounts.xlsx", sheet_name="Logs")
stack_data = df.groupby(["Year Established", "Sector"])
stack_data = stack_data.size().unstack().fillna(0)

stack_data.plot.bar(stacked=True)
plt.title("Companies per Year by Sector")
plt.xlabel("Year Established")
plt.ylabel("Company Count")
plt.tight_layout()
plt.show()
```

Streamlit

Modern web app framework for simple, data-driven use cases

A faster way to build and share data apps

Turn your data scripts into shareable web apps in minutes.
All in pure Python. No front-end experience required.

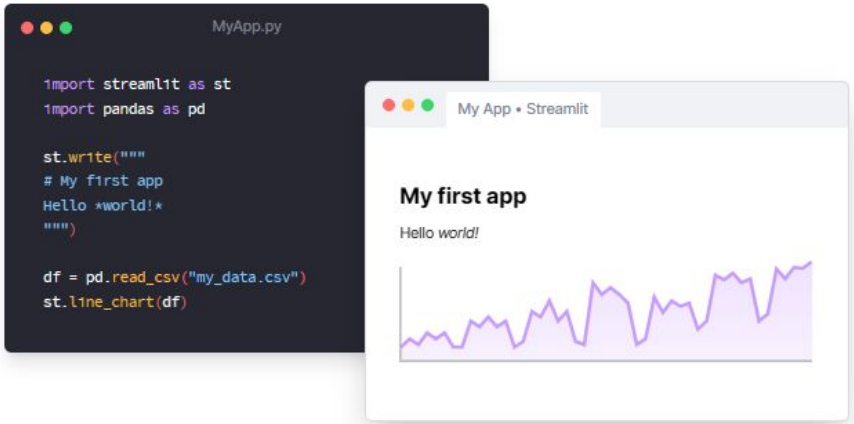
Get started Try the live playground!

On Streamlit.

Learn more with the Streamlit crash course on YouTube

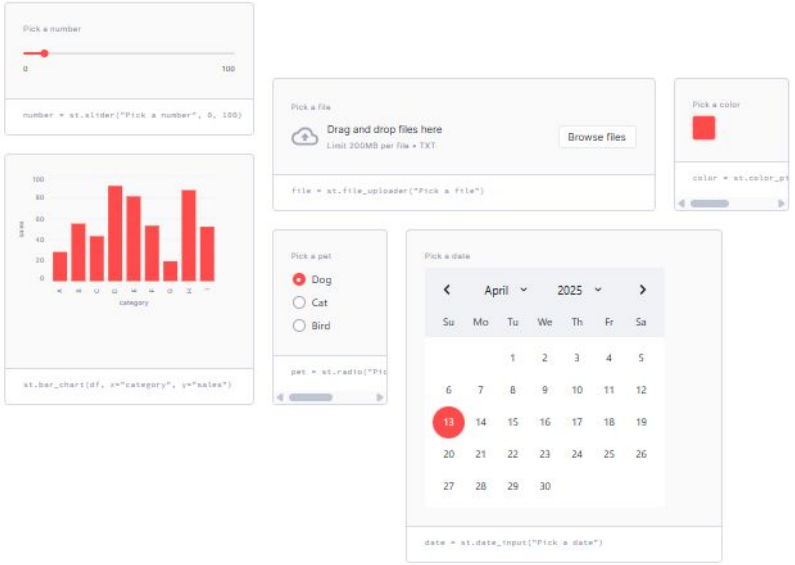
Embrace scripting

Build an app in a few lines of code with our **magically simple API**. Then see it automatically update as you iteratively save the source file.



Weave in interaction

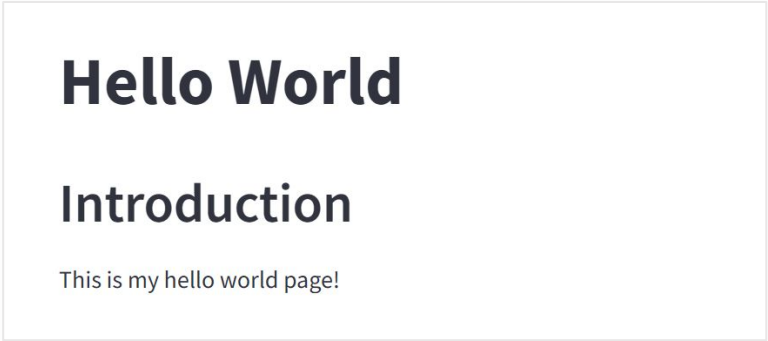
Adding a widget is the same as **declaring a variable**. No need to write a backend, define routes, handle HTTP requests, connect a frontend, write HTML, CSS, JavaScript, ...



Streamlit: Hello World

Make a new file with the following Python code.

```
import streamlit as st

st.title("Hello World")
st.header("Introduction")
st.text("This is my hello world page!")
```



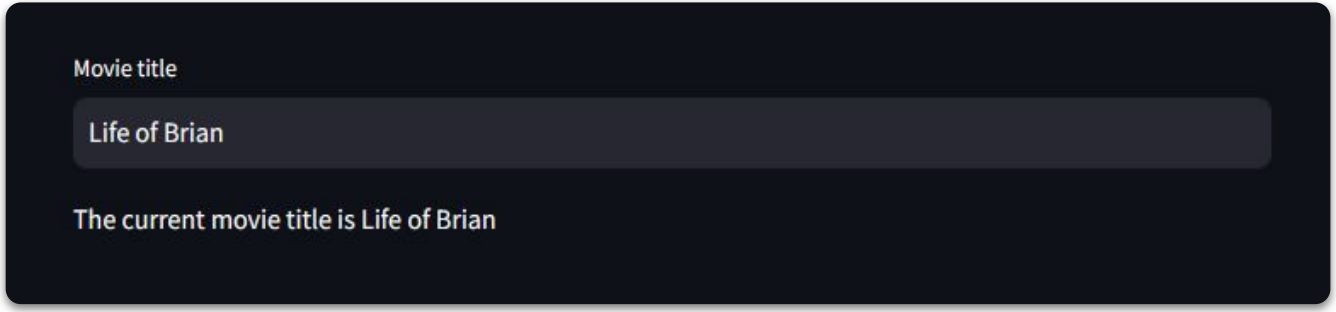
Components

Learn some of the available interactive elements

Text Input

The **`st.text_input`** displays a single-line text input widget.

```
import streamlit as st

title = st.text_input("Movie title", "Life of Brian")
st.write("The current movie title is", title)
```



Radio Buttons

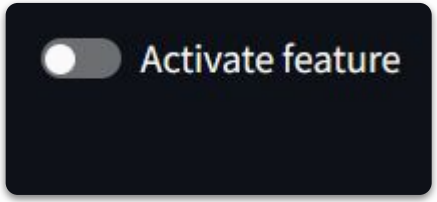
The **`st.radio`** displays a radio button widget

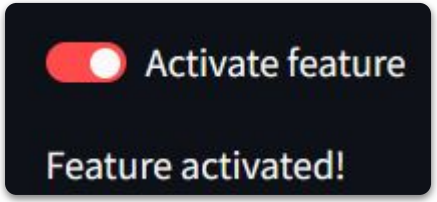
```
import streamlit as st

genre = st.radio(
    "What's your favorite movie genre",
    [":rainbow[Comedy]", "***Drama***", "Documentary :movie_camera:"],
    index=None,
)

st.write("You selected:", genre)
```

Toggle

The **`st.toggle`** displays a slider widget for integers, time, and datetime values

```
import streamlit as st

on = st.toggle("Activate feature")

if on:
    st.write("Feature activated!")
```

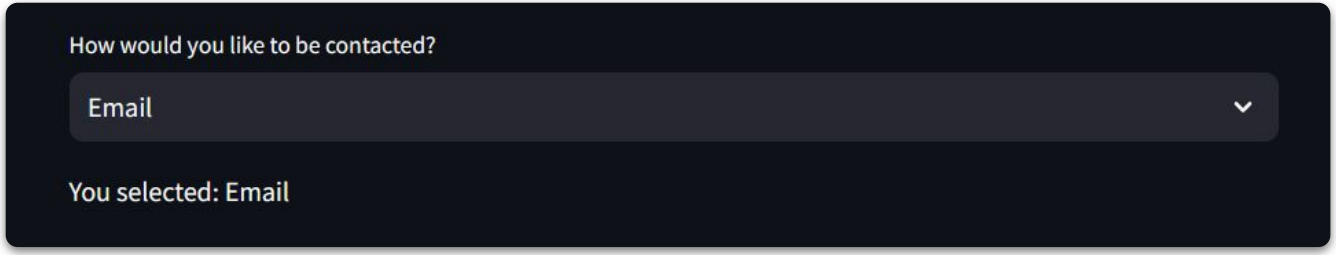




Select Box

The **`st.select_box`** displays a select widget for choosing a single value

```
import streamlit as st

option = st.selectbox(
    "How would you like to be contacted?",
    ("Email", "Home phone", "Mobile phone"),
)

st.write("You selected:", option)
```



Multiselect

The **`st.multiselect`** displays a multiselect widget

```
import streamlit as st

options = st.multiselect(
    "What are your favorite colors",
    ["Green", "Yellow", "Red", "Blue"],
    ["Yellow", "Red"],
)

st.write("You selected:", options)
```

Number Input

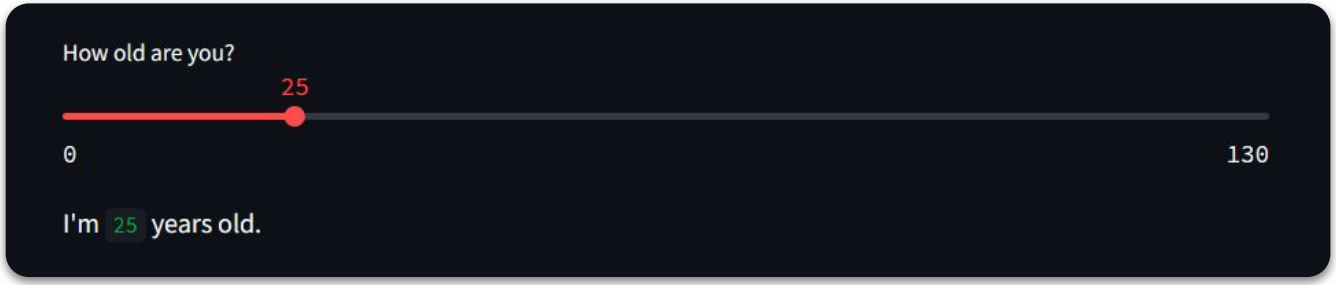
The **`st.number_input`** displays a numeric input widget

```python
import streamlit as st

number = st.number_input(
    "Insert a number", value=None, placeholder="Type a number..."
)
st.write("The current number is ", number)
```

Slider

The **`st.slider`** displays a slider widget for integers, time, and datetime values

```
import streamlit as st

age = st.slider("How old are you?", 0, 130, 25)
st.write("I'm ", age, "years old")
```



Submit Form

The **`st.form`** ensures that every input change doesn't refresh the page every time

```
import streamlit as st

with st.form("my_form"):
   st.write("Inside the form")
   my_number = st.slider('Pick a number', 1, 10)
   my_color = st.selectbox('Pick a color', ['red','orange','green','blue','violet'])
   st.form_submit_button('Submit my picks')

# This is outside the form
st.write(my_number)
st.write(my_color)
```

Data Handling

Process and visualize more data-intensive processes

Upload Files

Run the following on your chosen terminal to setup commits and remote connections

```
import streamlit as st

uploaded_files = st.file_uploader(
    "Choose a CSV file", accept_multiple_files=True
)
for uploaded_file in uploaded_files:
    bytes_data = uploaded_file.read()
    st.write("filename:", uploaded_file.name)
    st.write(bytes_data)
```

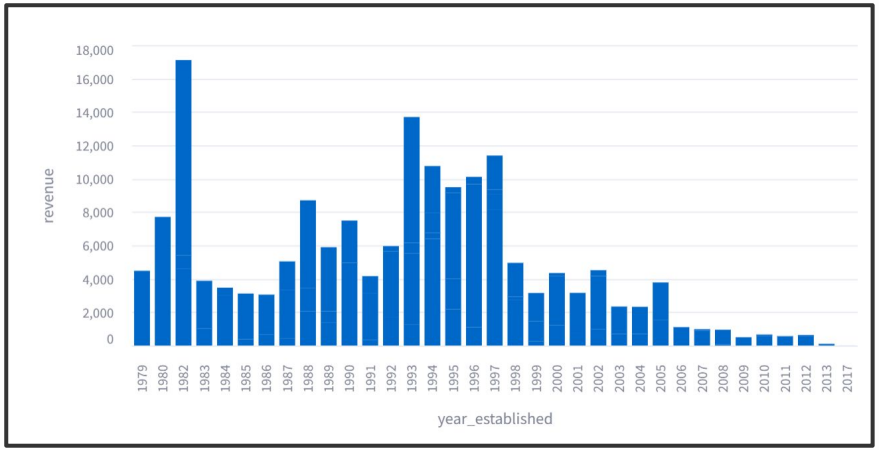
Read CSV and Excel File

Run the following on your chosen terminal to setup commits and remote connections

```
import streamlit as st
import pandas as pd

uploaded_file = st.file_uploader("File:", type=["csv", "xlsx", "xls"])

if uploaded_file is not None:
    st.write(f"Uploaded file: {uploaded_file.name}")

    if uploaded_file.name.endswith(".csv"):
        df = pd.read_csv(uploaded_file)
    elif uploaded_file.name.endswith((".xlsx", ".xls")):
        df = pd.read_excel(uploaded_file)

    st.write(df)
```

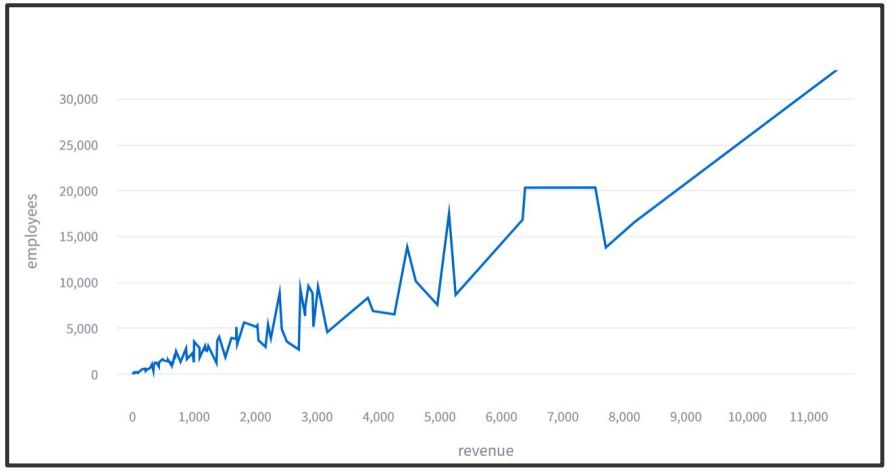
Bar Chart

```
1  import streamlit as st
2  import pandas as pd
3
4  df = pd.read_csv("data/sales/accounts.csv")
5  st.bar_chart(df, x="year_established", y="revenue")
```



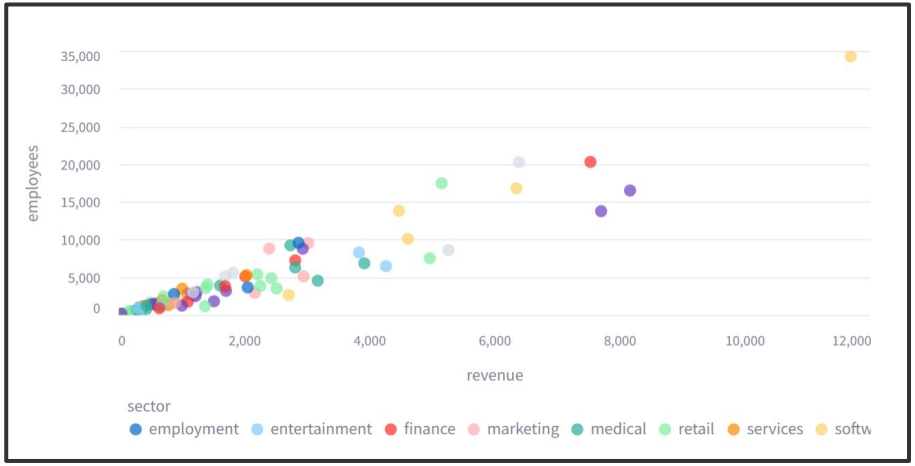
Line Plot

```
1  import streamlit as st
2  import pandas as pd
3
4  df = pd.read_csv("data/sales/accounts.csv")
5  st.line_chart(df, x="revenue", y="employees")
```



Scatter Chart

```python
import streamlit as st
import pandas as pd

df = pd.read_csv("data/sales/accounts.csv")
st.scatter_chart(df, x="revenue", y="employees", color="sector")
```

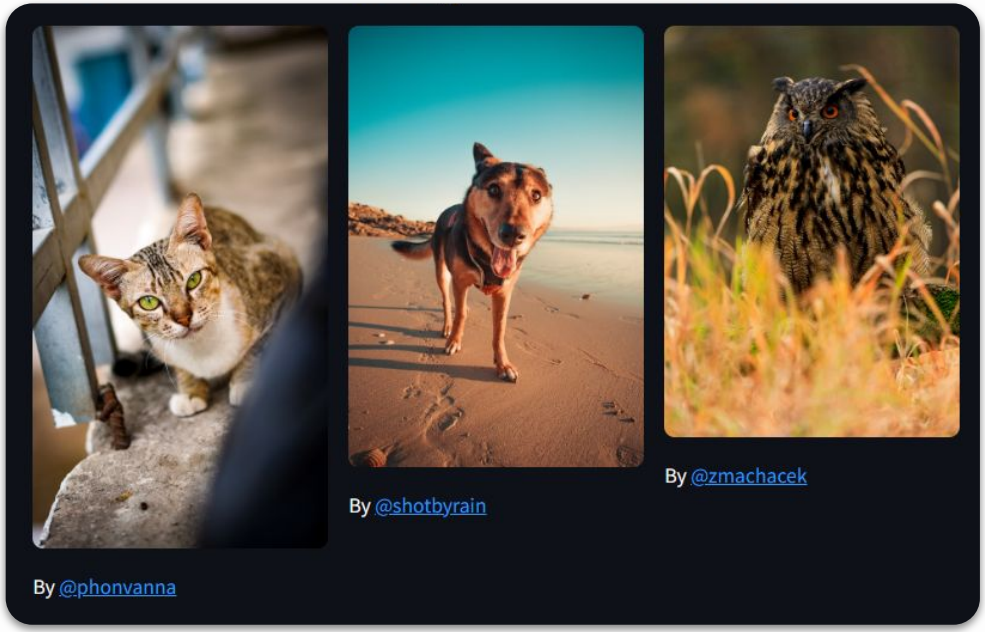


Modularization

High-level Streamlit code organization

Column Layouting

Streamlit supports multi-column layouts



Columns

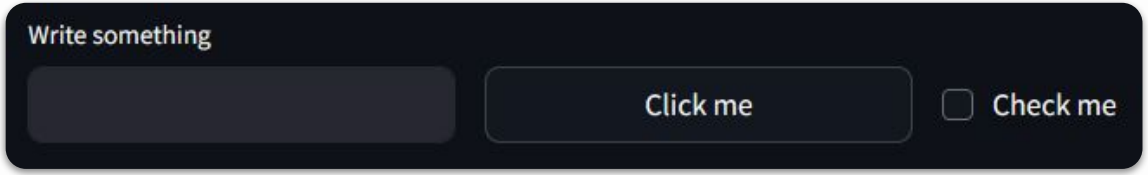
Using the context handler **`with`** syntax, content will be divided into separate columns

```
import streamlit as st

col1, col2, col3 = st.columns(3)

with col1:
    st.header("A cat")
    st.image("https://static.streamlit.io/examples/cat.jpg")

with col2:
    st.header("A dog")
    st.image("https://static.streamlit.io/examples/dog.jpg")

with col3:
    st.header("An owl")
    st.image("https://static.streamlit.io/examples/owl.jpg")
```

Simple Column Layout

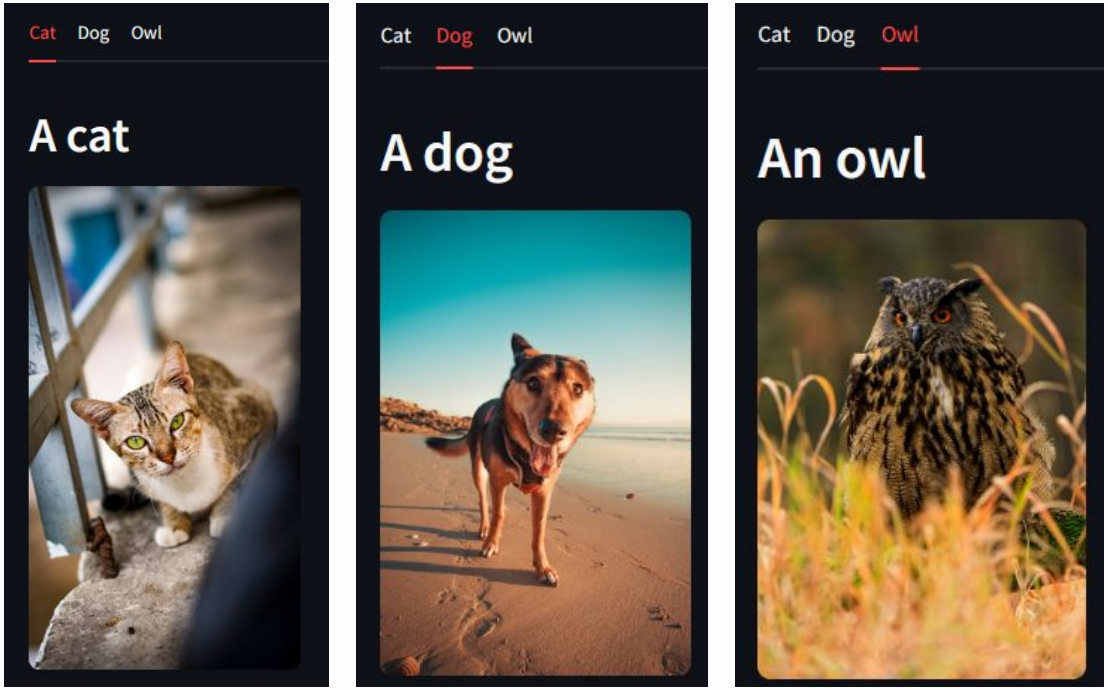
For simple columns, **`st`** can be replaced with the given column name

```
import streamlit as st

left, middle, right = st.columns(3, vertical_alignment="bottom")

left.text_input("Write something")
middle.button("Click me", use_container_width=True)
right.checkbox("Check me")
```



Tabs

Streamlit also supports tab layouts to prevent cluttering the page



Tabs

Using the context handler **`with`** syntax, content will be divided into separate tabs

```
import streamlit as st

tab1, tab2, tab3 = st.tabs(["Cat", "Dog", "Owl"])

with tab1:
    st.header("A cat")
    st.image("https://static.streamlit.io/examples/cat.jpg", width=200)
with tab2:
    st.header("A dog")
    st.image("https://static.streamlit.io/examples/dog.jpg", width=200)
with tab3:
    st.header("An owl")
    st.image("https://static.streamlit.io/examples/owl.jpg", width=200)
```

Multiple Pages

Multiple subpages are easy to implement in Streamlit. Place subpages in the **pages/** folder

```
.
└── project_name/
    ├── ...
    └── src/
        ├── pages/
        │   ├── subpage1.py
        │   ├── subpage2.py
        │   └── subpage3.py
        └── main.py
```

Beautiful Soup

Getting data from the web

Parsing a string

Beautiful soup can handle string files directly

```
1  from bs4 import BeautifulSoup
2
3  soup = BeautifulSoup("<html>a web page</html>", 'html.parser')
```

Parsing a document

Beautiful soup can also parse or open html files

```
1  from bs4 import BeautifulSoup
2
3  with open("index.html") as file:
4      soup = BeautifulSoup(file, 'html.parser')
```

Parsing a website online

Using the requests library, beautiful soup can also directly parse live websites

```
1  from bs4 import BeautifulSoup
2  import requests
3
4  url = "https://www.bbc.com/news"
5  response = requests.get(url)
6
7  if response.status_code == 200:
8      soup = BeautifulSoup(response.text, "html.parser")
```

Tags

Every detected component in the parser is a **Tag** object

```
1  from bs4 import BeautifulSoup
2
3  soup = BeautifulSoup(
4      '<b class="boldest">Extremely bold</b>',
5      'html.parser'
6  )
7  tag = soup.b
8  print(type(tag))
```

Tag Name

To access the HTML tag of the object, use the **name** field

```
1  from bs4 import BeautifulSoup
2
3  soup = BeautifulSoup(
4      '<b class="boldest">Extremely bold</b>',
5      'html.parser'
6  )
7  tag = soup.b
8  print(tag.name)
```

Tag String

To access the string contents the object, use the **`string`** field

```
1  from bs4 import BeautifulSoup
2
3  soup = BeautifulSoup(
4      '<b class="boldest">Extremely bold</b>',
5      'html.parser'
6  )
7  tag = soup.b
8  print(tag.string)
```

Finding tags (Explicit)

Soup and Tag objects have a find method that can be used to search for HTML tags.

```
1  from bs4 import BeautifulSoup
2  import requests
3
4  url = "https://www.bbc.com/news"
5  response = requests.get(url)
6
7  if response.status_code == 200:
8      soup = BeautifulSoup(response.text, "html.parser")
9      print(soup.find("head"))
```

Finding tags (Implicit)

Soup and Tag objects can also find tags using attribute access. It returns None if not found.

```
1  from bs4 import BeautifulSoup
2  import requests
3
4  url = "https://www.bbc.com/news"
5  response = requests.get(url)
6
7  if response.status_code == 200:
8      soup = BeautifulSoup(response.text, "html.parser")
9      print(soup.head)
```

Finding tags (Nested)

Tag finding can be nested using attribute access

```
1  from bs4 import BeautifulSoup
2  import requests
3
4  url = "https://www.bbc.com/news"
5  response = requests.get(url)
6
7  if response.status_code == 200:
8      soup = BeautifulSoup(response.text, "html.parser")
9      print(soup.body.h1)
```

Multiple Finding

To check for a tag in a soup or existing tag, use the `findAll` method

```
1  from bs4 import BeautifulSoup
2  import requests
3
4  url = "https://www.bbc.com/news"
5  response = requests.get(url)
6
7  if response.status_code == 200:
8      soup = BeautifulSoup(response.text, "html.parser")
9      print(soup.findAll('a'))
```

Multiple Finding (Chained Conditions)

The `findAll` method can also accept additional inputs to narrow down a search

```
1  from bs4 import BeautifulSoup
2  import requests
3
4  url = "https://www.bbc.com/news"
5  response = requests.get(url)
6
7  if response.status_code == 200:
8      soup = BeautifulSoup(response.text, "html.parser")
9      print(soup.findAll('a','head'))
```

Finding using ID's

Finally, the method can also find components using their id

```
1  from bs4 import BeautifulSoup
2  import requests
3
4  url = "https://www.bbc.com/news"
5  response = requests.get(url)
6
7  if response.status_code == 200:
8      soup = BeautifulSoup(response.text, "html.parser")
9      print(soup.findAll(id='a'))
```

Finding using Classes

Additionally, the `findAll` method can also find components using their id

```
1  from bs4 import BeautifulSoup
2  import requests
3
4  url = "https://www.bbc.com/news"
5  response = requests.get(url)
6
7  if response.status_code == 200:
8      soup = BeautifulSoup(response.text, "html.parser")
9      print(soup.findAll("p", class_="body"))
```

Python: Bonus

Anthology of additional topics