- Running Python code
- Variables and data types
- Strings
- Lists
- Tuples
- Dictionaries

- Conditional statements
- Loops
- Functions
- Importing modules and standard library

Integers: These are whole numbers like we have three numbers: ten, zero, and negative fifty. In Python, you would write these as: ten comma zero comma or negative fifty. They're useful for counts or math operations.

Floats: These are decimal numbers like we have two floating point numbers: three point one four and negative forty-two point five. In Python, you would write these as: three point one four or negative forty-two point five They're also useful for math operations that require precision.

Strings: These represent text and are surrounded by single or double quotes. We can concatenate strings too: 'Hello ' plus the string 'world!'. In Python, you would write this as: quote Hello space single or double quotes plus single or double quotes world exclamation mark single or double quotes.gives us the string 'Hello world!'. In Python, you would write this as: quote Hello space world exclamation mark single or double quotes.

Booleans: These are True or False values representing logical or comparison operations.

Lists: These are ordered sequences enclosed in square brackets like we have a list of three numbers: one, two, and three. In Python, you would write this as: open square bracket one comma two comma three close square bracket. Lists can contain different data types.

Tuples: These are like lists but are immutable once declared.

```
name = "John" # This is a string
age = 25 # This is an integer
has_pet = True # This is a boolean

name = "Sarah"
age = "Thirty" # Type is now string instead of integer

Now let's go over the main data types we can store in variables:

- Integers are whole numbers like 10, 0, -50. Useful for counts or
math.

- Floats are decimal numbers like 3.14, -42.5. Also useful for math
operations.

- Strings represent text and are surrounded by quotes. We can concat
```

strings too: "Hello " + "world!"

- Booleans are True or False values representing logical or comparison operations.

- Lists are ordered sequences enclosed in square brackets like [1, 2, 3]. Lists can contain different data types.

- Tuples are like lists but are immutable once declared.

- Dictionaries use key-value pairs like {"name": "Mary", "age": 23}

```python
age = 16

if age < 0:
  print("Invalid age!")
elif age < 18:
  print("You are a minor.")
elif age >= 65:
  print("You are a senior citizen.")
```

We can also combine multiple conditions using logical operators like and, or, and not:

```python
is_citizen = True
age = 19

if is_citizen and age >= 18:
  print("You are eligible to vote!")
```

```python
# Here's an example of a list:
my_list = [1, 2, 3]

# Indexing/Slicing:
my_list = [10, 20, 30, 40]
print(my_list[0])  # prints 10
print(my_list[:2])  # prints [10, 20]

# Append/Insert:
my_list.append(50)
print(my_list)  # prints [10, 20, 30, 40, 50]
my_list.insert(1, 15)
print(my_list)  # prints [10, 15, 20, 30, 40, 50]

# Delete/Pop:
del my_list[1]
print(my_list)  # prints [10, 20, 30, 40, 50]
```

```
last_element = my_list.pop()
print(last_element)  # prints 50

# Sort/Reverse:
my_list.sort()
print(my_list)  # prints [10, 20, 30, 40]
my_list.reverse()
print(my_list)  # prints [40, 30, 20, 10]

# Iterate Over Contents:
for element in my_list:
    print(element)  # prints each element on a new line

# Convert to Other Types:
my_tuple = tuple(my_list)
print(my_tuple)  # prints (40, 30, 20, 10)
my_set = set(my_list)
print(my_set)  # prints {40, 10, 20, 30}

# Here's an example of a tuple:
my_tuple = (1, 2, 3)

# Indexing/Slicing:
print(my_tuple[0])  # prints 1

# Iterating Over Contents:
for item in my_tuple:
    print(item)  # prints each item on a new line

# Convert to Other Types:
my_list = list(my_tuple)
print(my_list)  # prints [1, 2, 3]

# Creating a dictionary and accessing values:
my_dict = {'name': 'John Doe'}
print(my_dict['name'])  # prints 'John Doe'

# Changing Values:
my_dict['age'] = 31
print(my_dict)  # prints {'name': 'John Doe', 'age': 31}

# Adding Items:
my_dict['profession'] = 'Engineer'
print(my_dict)  # prints {'name': 'John Doe', 'age': 31, 'profession':
'Engineer'}

# Removing Items:
my_dict.pop('profession')
print(my_dict)  # prints {'name': 'John Doe', 'age': 31}

# Iterating Through a Dictionary:
```

```python
for key in my_dict:
    print(key, my_dict[key])  # prints each key-value pair

# clear() Method:
my_dict = {'name': 'John Doe', 'age': 30}
my_dict.clear()
print(my_dict)  # prints {}

# get() Method:
value = my_dict.get('name')
print(value)  # prints 'John Doe'

# items() Method:
items = my_dict.items()
print(items)  # prints dict_items([('name', 'John Doe'), ('age', 30)])

# keys() Method:
keys = my_dict.keys()
print(keys)  # prints dict_keys(['name', 'age'])

# values() Method:
values = my_dict.values()
print(values)  # prints dict_values(['John Doe', 30])

# update() Method:
my_dict.update({'profession': 'Engineer'})
print(my_dict)  # prints {'name': 'John Doe', 'age': 30, 'profession':
'Engineer'}

# Here's an example of a list:
my_list = [1, 2, 3]

# Indexing/Slicing:
my_list = [10, 20, 30, 40]
print(my_list[0])  # prints 10
print(my_list[:2])  # prints [10, 20]

# String Operations:
str1 = "Hello, "
str2 = "World!"
str3 = str1 + str2
print(str3)  # prints 'Hello, World!

# Repetition:
repeat_str = "Python! " * 3
print(repeat_str)  # prints 'Python! Python! Python!

# Common String Methods:
print(my_string.lower())  # prints 'hello, world!'
print(my_string.upper())  # prints 'HELLO, WORLD!'
```

```
plit_str = my_string.split()
print(split_str)  # prints ['Hello,', 'World!']
new_str = ' '.join(split_str)
print(new_str)  # prints 'Hello, World!'

new_str = " ".join(split_str)
print(new_str)  # prints 'Hello, World!'

# Concatenation:
greeting = "Hello"
name = "John"
message = greeting + " " + name
print(message)  # Output: "Hello John"

# Length:
text = "Python is amazing"
length = len(text)
print(length)  # Output: 1

# Indexing:
word = "Python"
first_letter = word[0]
print(first_letter)  # Output: "P"

text = "Hello, World!"
new_text = text.replace("Hello", "Hi")
print(new_text)  # prints 'Hi, World!'

# strip():
# The strip() method removes any leading or trailing whitespace from a
string. For example:
text = "   Python   "
print(text.strip()) # Output: "Python"

abs(x) # : Returns the absolute value of x.
len(s): Returns the length of s.
s.index(x): Returns the index of the first occurrence of x in s.

x = -10
print(abs(x))  # prints '10'

s = "Hello, World!"
print(len(s))  # prints '13'

s = "Hello, World!"
x = "World"
print(s.index(x))  # prints '7'

text = "Hello, World!"
new_text = text.replace("Hello", "Hi")
print(new_text)  # prints 'Hi, World!'
```

```
name = "John"
age = 25
message = f"My name is {name} and I'm {age} years old."
print(message)  # prints 'My name is John and I'm 25 years old.'

fruits = ["apple", "banana", "mango"]

for fruit in fruits:
  print(fruit)

fruits = ["apple", "banana", "mango"]

for i in range(len(fruits)):
  print(i, fruits[i])



for i in range(len(fruits)-1, -1, -1):
  print(i, fruits[i])

names = ["John", "Mary", "Bob"]
ages = [25, 34, 18]

for name, age in zip(names, ages):
  print(name, age)

tuples = [(1,2), (3,4), (5,6)]

for a, b in tuples:
  print(a, b)

dict = {"a": 1, "b": 2}

for key, value in dict.items():
  print(key, value)

This lets us conveniently access the items as variables without
indexing.

Next, let's discuss the start, stop, and step arguments we can pass to
range() to customize our loops.

for i in range(0, 10, 2):
  print(i) # Prints 0 2 4 6 8


This loops from 0 to 10 stepping by 2 each iteration.

We can also use start/stop/step to loop backwards:

python
for i in range(10, 0, -1):
```

```
  print(i) # Prints 10 9 8 7 ... 3 2 1


Being able to customize range() is useful when we want more control
over our iteration.

Another construct that can be helpful is the for-else loop:

for i in range(5):
  if i == 3:
    break
else:
  print("Loop completed without breaking")

The else block executes after the for loop finishes if it doesn't
encounter a break statement.

When mutating a list as we iterate through it, we need to be careful
to avoid errors.

For example, this will skip items:


values = [1, 2, 3, 4]

for v in values:
  values.remove(v)

To fix, we can loop over a copy of the list:


for v in values[:]:
  values.remove(v)


Or iterate backwards:


for v in reversed(values):
  values.remove(v)


There are a few solutions to safely modify lists during iteration.

Another common pattern is filtering lists during iteration:


numbers = [1, 2, 3, 4, 5, 6]

even_numbers = []
for n in numbers:
```

```
  if n % 2 == 0:
    even_numbers.append(n)


We can use a list comprehension here as a more Pythonic approach.
We'll come back to those later.

Now let's look at how we can use break and continue statements inside
loops:

break exits the loop immediately:


for i in range(10):
  if i == 5:
    break
  print(i)


continue skips to the next iteration:


for i in range(10):
  if i % 2 != 0:
    continue
  print(i)


These statements give us more control over the loop execution.

Looping over dictionaries is also common. We can use:


dict = {"a": 1, "b": 2}

for key in dict:
  print(key) # Prints keys

for key, value in dict.items():
  print(key, value) # Prints keys and values


items() lets us access both the key and value.

When should we use range() and len() vs enumerate()?

enumerate() gives us index and value:


for i, fruit in enumerate(fruits):
  print(i, fruit)
```

```
range() and len() is useful when we just need the index:


for i in range(len(fruits)):
  print(fruits[i])


Prefer enumerate() when feasible.

zip() is helpful for looping over multiple lists in parallel:


names = ["John", "Mary"]
ages = [25, 32]

for name, age in zip(names, ages):
  print(name, age)


The itertools module provides many useful iterator functions:


from itertools import product, permutations

for item in product([1,2], [3,4]):
  print(item) # Cartesian product

for item in permutations([1,2,3]):
  print(item) # Order permutations


Very useful for advanced looping scenarios.
```

```python
squared = []
for x in range(10):
  squared.append(x**2)

# The equivalent list comprehension:
squared = [x**2 for x in range(10)]

# And a dictionary comprehension:
dict = {x: x*2 for x in range(10)}
print("squared = :", squared)
print("dict = :", dict)

# The structure of a while loop looks like:
"""
while condition:
  # Do something
```

```
  # Update condition
"""
count = 0
while count < 5:
  print(count)
  count += 1

# This will print the numbers 0 through 4.

# We can use comparison operators like <, >, ==:
x = 0
while x < 10:
  print(x)
  x += 1


keep_going = True
while keep_going and x < 10:
  print(x)
  x += 1

# An idiomatic pattern is while True with conditional break:
"""
while True:
  x = get_next_value()
  if check_termination_condition(x):
    break

  process(x)
"""

# We can use else blocks after while loops:
found = False
i = 0
my_list = [...]  # Replace this with your list
item = ...  # Replace this with the item you're searching for
while i < len(my_list) and not found:
  if my_list[i] == item:
    found = True
  else:
    i += 1
if not found:
  # Didn't find item
  print("Item not found")

# The else block runs if the loop exits normally without hitting
break: Avoiding infinite loops
#Make sure to update loop conditions properly:
"""
x = 0
while x < 5:
```

```
  print(x)

"""
# Forgot to update x, so infinite loop

# Set a max iteration count to terminate:
max_iterations = 100
i = 0
while i < max_iterations:
  i += 1

# Incrementing/Decrementing
#Use a pattern like:
"""
while value < max_value:
  value += 1
"""

#Rather than:
"""
value = 0
while value < max_value:
  value = value + 1 # Don't do this!

"""

# do while Pattern:
#Perform at least one iteration using do while:
"""
do:
  print("Loop body")
while condition

**Break and Continue**
"""
# As we saw earlier, use break to exit loop and continue to skip to
next iteration.

# Looping Over Collections**
# While loops can loop over lists, dicts etc:

list = [1, 2, 3]
i = 0
while i < len(list):
  print(list[i])
  i += 1

# Prefer for loops though when iterating over collections.

# Counter Loop:
# Use a counting variable to complete a certain number of iterations:
```

```
count = 0
while count < 10:
  print(count)
  count += 1

# Input Loop:
# Loop while user input is valid:
"""
while True:
  input = get_input()
  if is_valid(input):
    process(input)
  else:
    break
"""

# Event Loop:
# Loop while waiting for events:
"""
while True:
  event = get_next_event()
  if event is None:
    break
  process_event(event)
"""
# This is very common in GUIs.

# Conditional Loop:
# Loop while some condition holds true:
"""
while temp > threshold:
  cooler.run()
  temp = read_temp()
"""

#Combining Conditions:
# Use and/or to require multiple conditions:
while x < 10 and y > 3:
  x += 1
  y -= 1
print(x,y)

# Iterative Approach:
def factorial(n):
  result = 1
  for i in range(2, n+1):
    result *= i
  return result
```

```python
# And recursion Approach:

def factorial(n):
  if n == 1:
    return 1
  return n * factorial(n-1)
```

Loops Best Practices:

```
Initialize loop variables properly.
Check for infinite loops.
Keep loop bodies clean and short.
Store results outside loop.
Use efficient data structures.
Profile code to optimize.
Prefer for loops for fixed iteration.
Use recursion for elegance, not efficiency.
```

Initialize loop variables before the loop

It's better to initialize counters or accumulators before the loop:

```python
# Do this:
index = 0
total = 0

while index < 10:
  total += index
  index += 1

# Not this:
while index < 10:
  total = 0 # Don't initialize here
  total += index
  index = 0 # Set initial value here
  index += 1
```

Initializing upfront makes the code cleaner.

Use descriptive variable names

For loop counters and other temporary vars, use names like i, j, etc:

```python
i = 0
while i < 10:
  print(i)
  i += 1
```

But use more descriptives names for accumulators:

```
total_sum = 0
while more_data:
  total_sum += next_data()
```

Check for infinite loops

Make sure your loop will terminate:

```
while True:
  print("Infinite Loop!")
```

Set a max iteration count or check conditions that will become false.

Avoid too many loop conditions

Having many complex conditions makes loops harder to reason about:

```
while x > 10 and y < 3 or x == 7 and z != 8:
  do_something()
```

Break conditions into steps before/inside loop if needed.

Use break/continue carefully

break and continue can make control flow confusing:

```
while True:
  if cond1:
    continue
  if cond2:
    break

  do_something()
```

Keep use of break/continue simple and minimize nesting.

Keep loop bodies clean and short

Loop bodies should do focused unit of work:

```
while x < 10:
  do_step1()
  do_step2()
  log_info()

  # This is too much for one loop iteration
  do_complex_step3()
  update_visuals()
  save_results()
```

Move extraneous logic outside loop if needed.

Store loop results outside loop

Accumulate results in outer scope variables:

```
# Good
total = 0
while x < 10:
  total += x

print(total)

# Avoid
while x < 10:
  print(total)
  total += x
```

This avoids re-processing each iteration.

Let's also discuss some performance optimizations:

Move loop invariant calculations outside loop

Calculate unchanging values once before loop:

```
data = load_data() # Only load once

while data:
  process(data)
```

Instead of:

```
while True:
  data = load_data() # Don't reload each time
  process(data)
```

Use efficient data structures

Some types like deque are faster than lists for queues/stacks.

Preallocate arrays

If appending to lists in loop, allocate list size upfront:

```
data = [None] * 10000
for i in range(10000):
   data[i] = calculate()
```

To avoid expensive resizing as list grows.

Use subroutines for repetitive logic

Move repetitive loop logic into functions:

```
# Factor out into function
def process_item(item):
  # do some work

while True:
  item = get_next()
  process_item(item)
```

This avoids rewriting code.

Profile code to identify slow areas

Use profiling tools to find optimization opportunities.

Now let's discuss some alternatives to while loops:

Recursion

Recursion can express repetitive logic elegantly:

```
def factorial(n):
  if n == 1:
    return 1
  else:
    return n * factorial(n-1)
```

But recursive logic is less efficient than loops in many cases.

For loops

Prefer for loops when you know number of iterations needed:

```
for i in range(10):
   print(i)
```

Use while loops when number of repetitions is unknown.

1. Defining Functions: In Python, we define functions using the `def` keyword, followed by the function name and parentheses. Let's see an example:

```
def greet():
    print("Hello, everyone!")

greet()  # Output: Hello, everyone!
```

In the example above, we define a function named `greet()` that prints a greeting message. We can call the function by using its name followed by parentheses.

1. Function Parameters: Functions can accept parameters, which are placeholders for values passed into the function. Parameters are defined inside the parentheses of the function definition. Let's see an example:

```
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")  # Output: Hello, Alice!
greet("Bob")  # Output: Hello, Bob!
```

In the example above, we define a function named `greet()` that takes a `name` parameter. We can pass different values to the function when calling it.

1. Return Statement: Functions can also return values using the `return` statement. The returned value can be stored in a variable or used directly. Let's see an example:

```
def add(a, b):
    return a + b

result = add(2, 3)
print(result)  # Output: 5
```

In the example above, we define a function named `add()` that takes two parameters, `a` and `b`, and returns their sum. We assign the returned value to the `result` variable and print it.

Great job! In this lecture, we explored the basics of functions in Python. We learned how to define functions, use parameters to accept input, and use the `return` statement to provide output. Functions are a powerful tool for organizing and reusing code, and they play a crucial role in writing clean and modular programs.

1. Understanding Recursion: Recursion is a process in which a function calls itself during its execution. It allows us to solve problems by dividing them into smaller and simpler versions of the same problem. A recursive function consists of two parts: the base case(s) and the recursive case(s).

2. Base Case: The base case is the condition that determines when the recursive function should stop calling itself. It provides a termination condition to prevent infinite recursion. Without a base case, the function would keep calling itself indefinitely. Let's see an example:

```
def countdown(n):
    if n == 0:
        print("Blastoff!")
    else:
        print(n)
        countdown(n - 1)

countdown(5)  # Output: 5 4 3 2 1 Blastoff!
```

In the example above, we define a recursive function named `countdown()` that prints numbers from `n` to 1, and finally, "Blastoff!" when `n` reaches 0. The base case is when `n` equals 0, and the function stops calling itself.

1. Recursive Case: The recursive case is the part of the function where it calls itself with a modified input. It breaks down the problem into a smaller version of itself. Let's see an example:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

result = factorial(5)
print(result)  # Output: 120
```

In the example above, we define a recursive function named `factorial()` that calculates the factorial of a number `n`. The recursive case multiplies `n` with the factorial of `n - 1` until `n` reaches 0, which is the base case.

Recursion allows us to solve complex problems by expressing them in terms of simpler subproblems. However, it's important to ensure that the base case is reached to avoid infinite recursion.

```
1. Function Decorators:
Function decorators are a way to modify the behavior of a function
without changing its source code. They are defined using the
`@decorator_name` syntax above the function definition. Let's see an
example:

```python
def uppercase_decorator(func):
    def wrapper():
        result = func().upper()
        return result
    return wrapper

@uppercase_decorator
def greet():
    return "hello"

print(greet())  # Output: HELLO
```

In the example above, we define a decorator named
`uppercase_decorator` that converts the result of the decorated
function to uppercase. We apply the decorator to the `greet()`
function using the `@` syntax.

2. Higher-Order Functions:
```

```
Higher-order functions are functions that take one or more functions as arguments or return functions as their results. They enable us to write more flexible and reusable code. Let's see an example:

```python
def multiply_by(n):
    def multiplier(x):
        return x * n
    return multiplier

double = multiply_by(2)
triple = multiply_by(3)

print(double(5))  # Output: 10
print(triple(5))  # Output: 15
```

Function Caching and Memoization: These techniques allow us to optimize the performance of functions by storing and reusing previously computed results. Let's dive in!

1. Function Caching: Function caching is a technique where the results of a function call are stored in memory for future use. If the samefunction is called again with the same arguments, the cached result is returned instead of recomputing the function. Let's see an example using the `functools` module:

```python
import functools

@functools.lru_cache()
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(10))  # Output: 55
print(fibonacci(15))  # Output: 610
```

In the example above, we define a recursive function `fibonacci()` that calculates the Fibonacci sequence. By applying the `@functools.lru_cache()` decorator, the function's results are cached, allowing subsequent calls with the same arguments to be retrieved from the cache.

1. Memoization: Memoization is a specific form of caching that stores the results of expensive function calls and returns the cached result when the same inputs occur again. Let's see an example using a memoization decorator:

```python
def memoize(func):
    cache = {}

    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
```
```

```
    return wrapper

@memoize
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)

print(factorial(5))  # Output: 120
print(factorial(10))  # Output: 3628800
```

In the example above, we define a memoization decorator `memoize()` that stores the results of the decorated function `factorial()` in a cache dictionary. If the same arguments are encountered again, the cached result is returned.

Function caching and memoization are powerful techniques to optimize the performance of functions, especially when dealing with computationally expensive tasks or recursive algorithms.

```
```