GRADAMIC

B.Tech–2nd Year Session 2023–24 Odd/Even Semester

Dr. A. P. J. Abdul Kalam Technical University
Lucknow, Uttar Pradesh

PYTHON PROGRAMMING SECURITY

(BCC 302 / BSS 302H / BCC 402 / BCC402H)

app.gradamic.com

UNIT – 3rd

UNIT - III

- Using string data type and string operations.
- Defining list and list slicing.
- Use of Tuple data type.
- String.
- List.
- Dictionary.
- Manipulations Building blocks of python programs.
- String manipulation methods.
- List manipulation.
- Dictionary manipulation.
- Programming using string.
- List and dictionary in-built functions.
- Python Functions.
- Organizing python codes using functions.

Using String Data Type & String Operations

The string data type in Python is used to store and manipulate sequences of characters. Python provides a rich set of operations and methods to work with strings. Here's a rundown of some common operations and methods:

Creating Strings

You can create strings using single quotes (' '), double quotes (" "), or triple quotes for multiline strings (""" """ or ''' '''):

 # Single line string

single_quoted_string = 'Hello, World!'

 # Double quoted string

double_quoted_string = "Python is awesome."

 # Multiline string

multiline_string = """This is a multiline

string spanning multiple lines."""

String Concatenation

You can concatenate strings using the `+` operator:

```
str1 = "Hello"
str2 = "World"
concatenated_string = str1 + ", " + str2 + "!"
print(concatenated_string)  # Output: Hello, World!
```

app.gradamic.com

String Indexing and Slicing

You can access individual characters of a string using indexing (0-based) and slice parts of strings using slicing notation (`[start:end:step]`):

Indexing refers to accessing a single character from a string by its position (index). Python uses zero-based indexing, meaning that the first character of a string is at index '0', the second character is at index '1', and so on.

Slicing allows you to extract a substring from a string by specifying a range of indices. The slice operation returns a new string that includes characters from the start index up to, but not including, the end index.

```
my_string = "Python"
print(my_string[0])    # Output: 'P'
print(my_string[1:4])  # Output: 'yth'
print(my_string[::2])  # Output: 'Pto' (every second character)
```

String Methods

Python strings have many built-in methods for various operations:

```
my_string = "   Hello, World!   "
# Removing leading and trailing whitespace
stripped_string = my_string.strip()
print(stripped_string)  # Output: 'Hello, World!'

# Splitting a string into a list of substrings
words = stripped_string.split(',')
print(words)  # Output: ['Hello', ' World!']

# Joining a list of strings into one string
new_string = '-'.join(words)
print(new_string)  # Output: 'Hello- World!'

# Finding substrings within a string
print(my_string.find('World'))  # Output: 8 (index of 'W' in 'World')
print(my_string.replace('World', 'Universe'))  # Output: '   Hello, Universe!   '
```

app.gradamic.com

String Formatting

Python provides several ways to format strings:

```
name = "Alice"
age = 30

# Using f-strings (formatted string literals)
formatted_string = f"My name is {name} and I am {age} years old."
print(formatted_string)  # Output: 'My name is Alice and I am 30 years old.'

# Using the format() method
formatted_string = "My name is {} and I am {} years old.".format(name, age)
print(formatted_string)  # Output: 'My name is Alice and I am 30 years old.'
```

Checking Substrings

You can check if a substring exists within a string using the `in` keyword:

```
my_string = "Hello, World!"

print('Hello' in my_string)  # Output: True
print('Python' in my_string)  # Output: False
```

String Operations

Python also supports various operations like checking the length of a string (`len()`), iterating over characters, and more.

```
my_string = "Python"
print(len(my_string))  # Output: 6
# Iterating over characters
for char in my_string:
    print(char)  # Output: each character on a new line
```

app.gradamic.com

Unicode Strings

Python 3 uses Unicode for strings by default, allowing you to work with characters from many languages and scripts seamlessly.

unicode_string = "こんにちは、世界！"

print(unicode_string) *# Output:* 'こんにちは、世界！'

These are some fundamental operations and methods you can perform with Python strings. Strings are versatile and form a crucial part of Python programming due to their extensive functionality and ease of use.

Defining List and List Slicing

In Python, a list is a built-in data structure that allows you to store a collection of items. Lists are mutable, meaning their elements can be changed after the list is created. Lists can contain elements of different types, including integers, floats, strings, and even other lists.

Defining a List:

You can define a list by enclosing elements in square brackets `[ ]`, separated by commas. Here's an example:

my_list = [1, 2, 3, 4, 5]

This creates a list named `my_list` containing five integers.

List Slicing:

List slicing is a powerful feature in Python that allows you to access a subset of elements from a list.

The syntax for slicing a list is `list[start:end:step]`, where:

- `start` is the index where the slicing starts (inclusive).
- `end` is the index where the slicing ends (exclusive).
- `step` is the stride or step size.

Here are some examples of list slicing:

1. **Basic Slicing:**

```
my_list = [1, 2, 3, 4, 5]
sliced_list = my_list[1:4]  # Returns [2, 3, 4]
```

In this example, `my_list[1:4]` slices elements starting from index 1 (2nd element) up to index 4 (5th element), but not including index 4.

app.gradamic.com

2. Slicing with Steps:

```
my_list = [1, 2, 3, 4, 5]
sliced_list = my_list[::2]  # Returns [1, 3, 5]
```

Here, `my_list[::2]` slices the list with a step size of 2, meaning it takes every second element starting from the beginning.

3. Negative Indices:

```
my_list = [1, 2, 3, 4, 5]
sliced_list = my_list[::-1]  # Returns [5, 4, 3, 2, 1]
```

Using a negative step (`[::-1]`) reverses the list.

Some Key Points-

- List slicing in Python is flexible and allows you to create new lists from existing lists efficiently.
- You can omit `start`, `end`, or `step` in list slicing. Omitting `start` defaults to the beginning of the list, omitting `end` defaults to the end of the list, and omitting `step` defaults to 1.
- Slicing does not modify the original list; it creates a new list with the specified elements.

Use of Tuple Data Type

In Python, a tuple is another built-in data type similar to a list, but with some key differences:

Definition and Characteristics of Tuples:

1. **Definition:**
   - A tuple is defined using parentheses `()` and elements are separated by commas.
   - **Example:**

```
my_tuple = (1, 2, 3, 4, 5)
```

2. **Immutability:**
   - Tuples are immutable, meaning once they are created, their elements cannot be changed or modified.
   - **Example:**

```
my_tuple = (1, 2, 3)
my_tuple[0] = 5  # This will raise a TypeError
```

app.gradamic.com

3. Use Cases:

- **Fixed Collection:** Tuples are suitable for representing fixed collections of items, especially when you want to ensure the data remains unchanged.

- **Functions Returning Multiple Values:** Functions can return tuples to conveniently return multiple values as a single unit.

- **Efficient Packing and Unpacking**: Tuples can be used to pack multiple values together and can be unpacked into individual variables.

- **Dictionary Keys:** Tuples can be used as keys in dictionaries if they contain immutable elements like strings or numbers.

Example Use Cases:

- **Returning Multiple Values:**

```
def get_point():
    x = 10
    y = 20
    return x, y

point = get_point()
print(point)  # Output: (10, 20)
```

- **Packing and Unpacking :**

```
coordinates = (3, 4)
x, y = coordinates
print(f"x = {x}, y = {y}")  # Output: x = 3, y = 4
```

- **Dictionary Keys:**

```
location_count = {
    (10, 20): "Location A",
    (30, 40): "Location B"
}
print(location_count[(10, 20)])  # Output: Location A
```

app.gradamic.com

Some Key Points-

- Tuples are immutable sequences, which makes them useful for representing fixed data that should not change.
- They support the same sequence operations as lists, such as indexing, slicing, and concatenation.
- Tuples are generally faster than lists when accessed and iterated over.
- The immutability of tuples provides a level of data integrity and safety, ensuring that data intended to be constant remains unchanged.

String

In Python, a string is a sequence of characters enclosed within either single quotes `' '` or double quotes `" "`. Strings are one of the most fundamental and versatile data types in Python and are used to represent textual data.

Creating Strings:

You can create strings in Python using single quotes, double quotes, or even triple quotes for multiline strings:

```python
# Single quotes
str1 = 'Hello, World!'

# Double quotes
str2 = "Python Programming"

# Multiline string using triple quotes
str3 = """This is a multiline
string in Python."""

# Strings with escape characters
str4 = "He said, \"Hello!\""
```

String Operations and Methods

Python provides a rich set of operations and methods to manipulate strings:

1. **Concatenation:**

```python
str1 = "Hello"
str2 = "World"
result = str1 + " " + str2  # Concatenation
print(result)  # Output: Hello World
```

app.gradamic.com

2. Indexing and Slicing:

```python
my_string = "Python"
print(my_string[0])    # Output: P (indexing)
print(my_string[2:5])  # Output: tho (slicing)
```

3. String Methods:

```python
my_string = "Python Programming"
print(my_string.lower())     # Output: python programming
print(my_string.upper())     # Output: PYTHON PROGRAMMING
print(my_string.startswith("Python"))   # Output: True
print(my_string.split())     # Output: ['Python', 'Programming']
```

4. Formatting Strings:

Python supports multiple ways to format strings, including using f-strings (formatted string literals), `.format()` method, and `%` formatting:

```python
name = "Alice"
age = 30
print(f"My name is {name} and I am {age} years old.")  # Output: My name is Alice and I am 30 years old.
```

5. String Immutability:

Strings in Python are immutable, meaning once a string is created, its contents cannot be changed. Any operation that modifies a string actually creates a new string object.

Some Key Points-

- Strings are sequences of characters and are treated as immutable in Python.
- They can be created using single, double, or triple quotes.
- Strings support various operations such as indexing, slicing, concatenation, and a wide range of methods for manipulation.

app.gradamic.com

- Python's string handling capabilities are extensive and include support for Unicode characters, making it powerful for handling text data in different languages and contexts.

List

In Python, a list is a versatile and mutable data structure that allows you to store a collection of items.

Lists are defined using square brackets `[ ]` and can contain elements of different data types, including integers, floats, strings, and even other lists.

Here's an overview of lists in Python:

Creating Lists:

You can create a list by enclosing elements in square brackets `[ ]`, separated by commas:

```
# List of integers
my_list = [1, 2, 3, 4, 5]
# List of strings
fruits = ["apple", "banana", "cherry"]
# Mixed data types
mixed_list = [1, "hello", 3.14, True]
```

Basic Operations on Lists:

1. **Accessing Elements:** Lists are indexed starting from 0. You can access elements using square bracket notation:

```
print(my_list[0])   # Output: 1
print(fruits[1])    # Output: banana
```

2. **List Length:** You can find the number of elements in a list using the `len()` function:

```
print(len(my_list))   # Output: 5
```

3. **Appending and Extending Lists:** You can add elements to a list using `append()` to add one element at a time or `extend()` to add elements from another list:

```
my_list.append(6)          # Adds 6 to the end of my_list
fruits.extend(["orange", "grape"])  # Adds multiple elements to fruits
```

app.gradamic.com

4. **Slicing Lists:** You can extract a portion of a list using slicing, which creates a new list:

```
sliced_list = my_list[1:4]  # Returns [2, 3, 4]
```

5. **Modifying Elements:** Lists are mutable, so you can change individual elements:

```
fruits[0] = "pear"  # Changes "apple" to "pear"
```

6. **Removing Elements:** You can remove elements by value with `remove()` or by index with `del` or `pop()`:

```
fruits.remove("banana")  # Removes "banana" from fruits
del my_list[0]           # Removes the element at index 0
popped_element = my_list.pop()  # Removes and returns the last element
```

7. **Concatenating Lists:** You can concatenate two lists using the `+` operator:

```
combined_list = my_list + fruits
```

List Comprehensions

List comprehensions provide a concise way to create lists. They consist of an expression followed by a `for` clause and can optionally include `if` clauses:

```
squares = [x**2 for x in range(10)]  # Creates a list of squares of numbers from 0 to 9
```

Some Key Points-

- Lists in Python are ordered, mutable collections of elements.
- They can contain elements of different data types and support various operations such as indexing, slicing, appending, extending, and more.
- List comprehensions offer a powerful way to create lists based on existing lists or iterable objects.
- Understanding lists is essential for manipulating collections of data in Python efficiently.

List & Dictionary

In Python, lists and dictionaries are both fundamental data structures, but they serve different purposes and have distinct characteristics. Here's a comparison of lists and dictionaries based on their features and usage:

Lists:

1. **Definition:**

Lists are ordered collections of items, where each item is identified by its position or index. Elements in a list can be of any data type, and different data types can be mixed within the same list.

ALL COPYRIGHTS 2024 RESERVED BY: GRADAMIC

2. Mutable:

Lists are mutable, meaning you can change, add, or remove elements after the list is created.

- Example:

```
my_list = [1, 2, 3, "apple", "banana"]
my_list[0] = 10  # Modifying an element
my_list.append("cherry")  # Adding an element
del my_list[3]  # Removing an element
```

3. Accessing Elements:

Elements in a list are accessed using indexing (starting from 0) and slicing:

```
print(my_list[1])    # Output: 2
print(my_list[2:4])  # Output: [3, "apple"]
```

4. Ordered:

Lists maintain the order of elements as they are inserted. The order of elements is preserved unless explicitly changed.

5. Use Cases:

Lists are commonly used when the order of elements matters, such as maintaining sequences of data or when you need a mutable collection of items.

Dictionaries:

1. Definition:

Dictionaries are unordered collections of items, where each item is stored as a key-value pair.

Keys in a dictionary must be unique and immutable (such as strings, numbers, or tuples), while values can be of any data type.

2. Mutable:

Like lists, dictionaries are mutable, allowing you to change, add, or remove key-value pairs.

- Example:

```
my_dict = {"name": "Alice", "age": 30, "city": "New York"}
my_dict["age"] = 31  # Modifying a value
my_dict["gender"] = "Female"  # Adding a new key-value pair
del my_dict["city"]  # Removing a key-value pair
```

app.gradamic.com

3. Accessing Elements:

Elements in a dictionary are accessed using keys rather than indices:

```
print(my_dict["name"])   # Output: Alice
```

4. Unordered:

Dictionaries do not maintain the order of elements as they are inserted. The order of items may vary between iterations.

5. Use Cases:

Dictionaries are suitable for scenarios where you need to store and retrieve data based on keys rather than positions, or when you need to associate data with specific labels or identifiers.

Comparison:

- **Access Method:** Lists use numerical indices to access elements, while dictionaries use keys.
- **Order:** Lists maintain the order of elements, whereas dictionaries do not.
- **Purpose:** Lists are ideal for sequences and collections where the order of elements matters, while dictionaries excel at storing and retrieving data based on keys.
- **Flexibility:** Lists allow for heterogeneous elements and are more straightforward for sequential data access. Dictionaries are efficient for associative data access and are useful for representing structured information.

Manipulations Building Blocks of Python Programs

In Python, there are several fundamental building blocks and techniques that form the foundation for building programs. These building blocks include variables, control structures (like loops and conditionals), functions, data structures (such as lists and dictionaries), and modules. Let's explore each of these in the context of Python programming:

1. Variables and Data Types:

Variables are used to store data values.

In Python, variables are dynamically typed, meaning you don't need to declare the type of a variable explicitly. Some common data types in Python include:

- **Integers:** Whole numbers (`int`).
- **Floating-point numbers**: Decimal numbers (`float`).
- **Strings**: Sequence of characters (`str`).
- **Booleans:** True or False values (`bool`).
- **Lists:** Ordered collections of items.

app.gradamic.com

- **Dictionaries:** Unordered collections of key-value pairs.
- **Tuples**: Immutable ordered collections.
- **Sets:** Unordered collections of unique items.

Example:

```
name = "Alice"
age = 30
is_student = False
numbers = [1, 2, 3, 4]
```

2. Control Structures:

Control structures allow you to control the flow of execution in your program:

- **Conditionals (`if`, `elif`, `else`):** Execute code based on conditions.

```
if age >= 18:
    print("Adult")
else:
    print("Minor")
```

- **Loops (`for` loops, `while` loops):** Iterate over sequences or execute code repeatedly.

```
for number in numbers:
    print(number)

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

3. Functions:

Functions allow you to encapsulate reusable pieces of code:

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

message = greet("Alice")
print(message)  # Output: Hello, Alice!
```

app.gradamic.com

4. Data Structures:

Python provides built-in data structures to store and manipulate collections of data:

- **Lists:** Mutable ordered collections.
- **Dictionaries**: Unordered collections of key-value pairs.
- **Tuples**: Immutable ordered collections.
- **Sets:** Unordered collections of unique items.

Example:

```
student = {"name": "Alice", "age": 20, "courses": ["Math", "History"]}
grades = (85, 90, 78)
```

5. Modules and Packages:

Python modules are files containing Python code that define functions, classes, and variables. Packages are directories of modules:

- **Importing Modules:**

```
import math
print(math.sqrt(25))  # Output: 5.0
```

- **Creating Modules:**

```
# mymodule.py
def say_hello(name):
    return f"Hello, {name}”
```

- **Using Packages:**

```
# main.py
from mypackage import mymodule
print(mymodule.say_hello("Alice"))  # Output: Hello, Alice!
```

6. Exception Handling:

Handling errors and exceptions in Python using `try`, `except`, `finally`, and `raise`:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
finally:
    print("Execution completed")
```

app.gradamic.com

7. Input and Output:

Reading input from users and writing output to the console or files:

```
name = input("Enter your name: ")
print(f"Hello, {name}!")
```

8. Object-Oriented Programming (OOP):

Python supports OOP principles, allowing you to define classes and create objects:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        return f"Hello, my name is {self.name} and I am {self.age} years old."

alice = Person("Alice", 30)
print(alice.greet())  # Output: Hello, my name is Alice and I am 30 years old.
```

String Manipulation Methods

In Python, there are several built-in string manipulation methods that allow you to perform various operations on strings. Here are some commonly used string methods:

1. **Basic Methods:**

- **`str.upper()`:** Returns a copy of the string with all characters converted to uppercase.
- **`str.lower()`:** Returns a copy of the string with all characters converted to lowercase.
- **`str.capitalize()`:** Returns a copy of the string with the first character converted to uppercase and the rest to lowercase.
- **`str.title()`:** Returns a copy of the string with the first character of each word converted to uppercase and all other characters to lowercase.

2. **Search and Replace:**

- **`str.find(substring)`:** Returns the lowest index in the string where substring `substring` is found. Returns `-1` if not found.

app.gradamic.com

- **`str.replace(old, new)`:** Returns a copy of the string with all occurrences of substring `old` replaced by `new`.

- **`str.startswith(prefix)`:** Returns `True` if the string starts with the specified prefix; otherwise, returns `False`.

- **`str.endswith(suffix)`:** Returns `True` if the string ends with the specified suffix; otherwise, returns `False`.

3. **Whitespace Removal:**

- **`str.strip()`:** Returns a copy of the string with leading and trailing whitespace removed.

- **`str.lstrip()`:** Returns a copy of the string with leading whitespace removed.

- **`str.rstrip()`:** Returns a copy of the string with trailing whitespace removed.

4. **Splitting and Joining:**

- **`str.split(sep)`:** Returns a list of words in the string using `sep` as the delimiter. If `sep` is not specified, any whitespace is a separator.

- **`str.join(iterable)`:** Returns a string which is the concatenation of the strings in `iterable`, with the original string `str` used as a separator.

5. **Checking and Formatting:**

- **`str.isdigit()`:** Returns `True` if all characters in the string are digits; otherwise, returns `False`.

- **`str.isalpha()`:** Returns `True` if all characters in the string are alphabetic; otherwise, returns `False`.

- **`str.isalnum()`:** Returns `True` if all characters in the string are alphanumeric (either letters or numbers); otherwise, returns `False`.

- **`str.format(*args, **kwargs)`:** Formats the string into a nicer output using positional and keyword arguments.

6. **Character and Length:**

- **`str.len()`:** Returns the length of the string (number of characters).

These are just a selection of the most commonly used string methods in Python. Depending on your specific needs, there are many more methods available in Python's string manipulation toolkit.

Dictionary Manipulation

Dictionary manipulation in Python involves various operations such as adding or removing elements, accessing values, iterating through items, and merging dictionaries.

Here’s a rundown of some common operations:

app.gradamic.com

Creating a Dictionary

```python
# Method 1: Using curly braces {}
my_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}
# Method 2: Using dict() constructor
my_dict = dict(name='Alice', age=30, city='New York')
```

Accessing Elements

```python
# Accessing by key
print(my_dict['name'])  # Output: 'Alice'

# Accessing with get() method (handles key errors gracefully)
print(my_dict.get('age'))  # Output: 30
print(my_dict.get('address', 'Not Found'))  # Output: 'Not Found'
```

Adding or Updating Elements

```python
# Adding a new key-value pair
my_dict['email'] = 'alice@example.com'
# Updating an existing value
my_dict['age'] = 31
```

Removing Elements

```python
# Removing a key-value pair
del my_dict['city']
# Removing all elements
my_dict.clear()
```

appgradamic.com

Iterating Through a Dictionary

```python
for key, value in my_dict.items():
    print(key, value)
```

Checking if Key Exists

```python
if 'name' in my_dict:
    print('Name is', my_dict['name']) ```

### Dictionary Methods

```python
# Get all keys
keys = my_dict.keys()
# Get all values
values = my_dict.values()
# Copy a dictionary
new_dict = my_dict.copy()
# Merge dictionaries (Python 3.9+)
my_dict.update({'phone': '555-1234'})
```

Dictionary Comprehension

```python
my_dict = {key: value for key, value in my_dict.items() if key != 'age'}
```

Handling Nested Dictionaries

```python
users = {
    'user1': {'name': 'Alice', 'age': 30},
    'user2': {'name': 'Bob', 'age': 25}
}
```

app.gradamic.com
```

```
# Accessing nested values
print(users['user1']['name'])  # Output: 'Alice'
```

Programming Using String

Programming with strings in Python involves manipulating text data, formatting strings, searching within strings, and performing various operations such as concatenation, slicing, and formatting. *Here's a comprehensive overview:*

Creating Strings

```
# Using single quotes
my_string = 'Hello, World!'
# Using double quotes
my_string = "Hello, World!"

# Using triple quotes for multiline strings
multiline_string = """This is a
                      multiline string."""
```

String Operations

Concatenation

```python
str1 = 'Hello'
str2 = 'World'
concatenated = str1 + ' ' + str2  # Result: 'Hello World'
```

String Length

```python
length = len(my_string)  # Length of the string
```

Accessing Characters

```python
char = my_string[0]  # Accessing first character
```

app.gradamic.com

Slicing

```python
substring = my_string[7:12]  # Slicing 'World' from 'Hello, World!'
```

String Methods

Conversion and Case Conversion

```python
lower_case = my_string.lower()
upper_case = my_string.upper()
```

Splitting and Joining

```python
words = my_string.split(',')  # Splits at commas into a list of words
joined_string = ' '.join(words)  # Joins list into a single string with spaces
```

Finding and Replacing

```python
index = my_string.find('World')  # Finds the index of 'World'
new_string = my_string.replace('Hello', 'Hi')  # Replaces 'Hello' with 'Hi'
```

Checking Substrings

```python
contains = 'Hello' in my_string  # Checks if 'Hello' is in my_string
```

Stripping Whitespace

```python
stripped = my_string.strip()  # Removes leading and trailing whitespace
```

Formatting Strings (f-strings)

```
name = 'Alice'
age = 30
formatted_string = f'My name is {name} and I am {age} years old.'
```

app.gradamic.com

String Formatting (Older Methods)

Using `.format()`

```python
formatted_string = 'My name is {} and I am {} years old.'.format(name, age)
```

Using `%` Formatting (Older)

```python
formatted_string = 'My name is %s and I am %d years old.' % (name, age)
```

Escaping Characters

```python
escaped_string = 'He\'s going to the park.'  # Using backslash (\) to escape
```

Unicode Strings

```python
unicode_string = 'Hello, \u03A9'  # Unicode character Ω
```

Raw Strings

```python
raw_string = r'C:\Users\Documents\file.txt'  # Ignores escape sequences
```

Checking String Types

```python
is_alpha = my_string.isalpha()  # Checks if all characters are alphabetic
is_digit = my_string.isdigit()  # Checks if all characters are digits
```

String Interpolation

```
name = 'Alice'
age = 30
formatted_string = 'My name is %s and I am %d years old.' % (name, age)
```

app.gradamic.com

List and Dictionary In-Built Functions

Certainly! Python provides a rich set of built-in functions for both lists and dictionaries that make them powerful data structures to work with. Here's a comprehensive list of commonly used built-in functions for lists and dictionaries:

List Built-in Functions

1. ****len()****- Returns the number of items in the list.

```python
my_list = [1, 2, 3, 4, 5]
length = len(my_list)  # Returns 5
```

2. ****append()****- Adds an element to the end of the list.

```python
my_list.append(6)  # Appends 6 to the end of my_list
```

3. ****extend()****- Extends the list by appending elements from another iterable.

```python
another_list = [7, 8, 9]
my_list.extend(another_list)  # Appends [7, 8, 9] to my_list
```

4. ****insert()****- Inserts an element at a specified position in the list.

```python
my_list.insert(2, 'inserted')  # Inserts 'inserted' at index 2
```

5. ****remove()****- Removes the first occurrence of a value from the list.

```python
my_list.remove(3)  # Removes the first occurrence of 3
```

6. ****pop()**** - Removes and returns the element at a specified index (default is the last element).

```python
popped_element = my_list.pop(0)  # Removes and returns the element at index 0
```

app.gradamic.com

```

7. **index()** - Returns the index of the first occurrence of a value in the list.

   ```python
   idx = my_list.index(4)  # Returns the index of the first occurrence of 4
   ```

8. **count()**- Returns the number of occurrences of a value in the list.

   ```python
   count = my_list.count(5)  # Returns the number of occurrences of 5
   ```

9. **sort()**- Sorts the list in place.

   ```python
   my_list.sort()  # Sorts the list in ascending order
   ```

10. **reverse()** - Reverses the elements of the list in place.

    ```python
    my_list.reverse()  # Reverses the order of elements in the list
    ```

11. **copy()**

    - Returns a shallow copy of the list.

    ```python
    new_list = my_list.copy()  # Copies the contents of my_list to new_list
    ```

12. **clear()**

    - Removes all elements from the list.

    ```python
    my_list.clear()  # Clears all elements from my_list
    ```

Dictionary Built-in Functions

1. **len()**- Returns the number of key-value pairs in the dictionary.

   ```python
```

app.gradamic.com
```

```
my_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}
length = len(my_dict)  # Returns 3
```

2. **keys()**- Returns a view object that displays a list of all keys in the dictionary.

```python
keys_list = my_dict.keys()  # Returns a view object of keys ('name', 'age', 'city')
```

3. **values()**- Returns a view object that displays a list of all values in the dictionary.

```python
values_list = my_dict.values()  # Returns a view object of values ('Alice', 30, 'New York')
```

4. **items()** - Returns a view object that displays a list of key-value tuple pairs in the dictionary.

```python
items_list = my_dict.items()  # Returns a view object of items (('name', 'Alice'), ('age', 30), ('city', 'New York'))
```

5. **get()**- Returns the value for a specified key.

```python
name = my_dict.get('name')  # Returns 'Alice'
```

6. **pop()**- Removes the element with the specified key and returns its value.

```python
age = my_dict.pop('age')  # Removes 'age' key and returns its value (30)
```

7. **popitem()**- Removes and returns the last inserted key-value pair.

```python
last_item = my_dict.popitem()  # Removes and returns the last key-value pair inserted
```

8. **clear()**- Removes all elements from the dictionary.

```python
my_dict.clear()  # Clears all key-value pairs from my_dict
```

app.gradamic.com

```

**9. **copy()****- Returns a shallow copy of the dictionary.

```python
new_dict = my_dict.copy()  # Copies the contents of my_dict to new_dict
```

10. **update()**- Updates the dictionary with the key-value pairs from another dictionary or iterable.

```python
another_dict = {'email': 'alice@example.com'}
my_dict.update(another_dict)  # Adds {'email': 'alice@example.com'} to my_dict
```

Python Functions

Functions in Python are blocks of organized, reusable code designed to perform a specific task.

They allow you to break down your program into smaller, modular pieces, which makes your code more organized, easier to understand, and easier to debug.

Here's a comprehensive overview of Python functions:

Defining a Function

In Python, you define a function using the `def` keyword followed by the function name and parentheses `( )`.

You can optionally specify parameters inside the parentheses.

```python
def greet():
    print("Hello, welcome to Python Functions!")
```

Function Parameters

Parameters (or arguments) are values that you can pass into a function. A function can have multiple parameters separated by commas.

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

app.gradamic.com
```

Function Arguments

Arguments are actual values passed into a function when it is called.

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

Return Statement

Functions can optionally return a value using the `return` statement. If no return statement is used, the function returns `None` by default.

```python
def add_numbers(x, y):
    return x + y
result = add_numbers(3, 5)  # result will be 8
```

Default Parameters

You can specify default values for parameters. These parameters become optional when the function is called.

```python
def greet(name="Guest"):
    print(f"Hello, {name}!")
greet()  # Output: Hello, Guest!
greet("Alice")  # Output: Hello, Alice!
```

Keyword Arguments

You can pass arguments to a function by explicitly naming them, which allows you to change the order of arguments and makes the function call more readable.

```python
def describe_pet(animal_type, pet_name):
    print(f"I have a {animal_type}. Its name is {pet_name}.")


describe_pet(animal_type="cat", pet_name="Whiskers")
```

app.gradamic.com

Arbitrary Number of Arguments

If you do not know how many arguments will be passed into your function, you can use `*args` and `**kwargs` to handle variable numbers of arguments.

```python
def greet_multiple(*names):
    for name in names:
        print(f"Hello, {name}!")

greet_multiple("Alice", "Bob", "Charlie")
```

Lambda Functions (Anonymous Functions)

Lambda functions are small, anonymous functions defined with the `lambda` keyword. They can have any number of arguments, but only one expression.

```python
multiply = lambda x, y: x * y
print(multiply(3, 4))  # Output: 12
```

Docstrings

Docstrings are used to describe the purpose and usage of functions. They are enclosed in triple quotes (`""" """`) and are optional but highly recommended.

```python
def add_numbers(x, y):
    """
    Adds two numbers together and returns the result.
    """
    return x + y
```

Scope of Variables

Variables defined inside a function are local to that function and cannot be accessed outside it unless explicitly declared as global.

```python
```

app.gradamic.com

```
def my_function():
    x = 10  # Local variable
    print(x)
my_function()
print(x)  # This will raise a NameError because x is not defined globally
```

Function Calling Another Function

Functions can call other functions within them to achieve modularity and avoid redundancy.

```python
def square(x):
    return x ** 2
def cube(x):
    return square(x) * x
result = cube(3)  # result will be 27
```

Decorators

Decorators are a powerful way to modify or enhance the behavior of functions or methods without changing their definition.

```python
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper
@my_decorator
def say_hello():
    print("Hello!")
say_hello()
```

app.gradamic.com

Built-in Functions Related to Functions

Python also provides several built-in functions that are related to functions, such as `map()`, `filter()`, `reduce()`, `sorted()`, etc., which can be used for functional programming.

Functions are fundamental in Python programming, enabling code reuse, abstraction, and modularization. They are a powerful tool for structuring and organizing code effectively.

Organizing Python Codes Using Functions

Organizing Python code using functions is crucial for improving readability, maintainability, and reusability of your codebase.

Here are several key strategies and best practices for effectively organizing Python code using functions:

1. Modularization

Break down your code into smaller, cohesive modules. Each module should focus on a specific task or functionality, and functions should be grouped logically within these modules.

Example:

```python
# math_operations.py
def add(x, y):
    return x + y
def subtract(x, y):
    return x - y
# file_operations.py
def read_file(filename):
    with open(filename, 'r') as f:
        return f.read()
def write_file(filename, data):
    with open(filename, 'w') as f:
        f.write(data)
```

2. Encapsulation

Encapsulate related functionality into functions, hiding implementation details. Functions should have clear inputs (parameters) and outputs (return values), with minimal side effects.

Example:

```python
# user_operations.py
def create_user(username, password):
    # Logic to create a new user in the system
    pass
```

app.gradamic.com

```
def authenticate_user(username, password):
    # Logic to authenticate a user
    pass
```

3. Avoiding Code Duplication

Identify repetitive code patterns and refactor them into reusable functions. This reduces redundancy and makes your codebase more maintainable.

Example:

```python
# Before refactoring
def process_data(data):
    # Processing logic here
    Pass
# After refactoring
def process_data(data):
    # Reusable processing logic
    pass
def main():
    data1 = process_data(data1)
    data2 = process_data(data2)
```

4. Use of Helper Functions

Break down complex tasks into smaller, manageable helper functions. This improves code readability and allows you to focus on specific aspects of functionality within a larger process.

Example:

```python
# data_processing.py
def preprocess_data(data):
    # Preprocessing steps
    pass
def analyze_data(data):
    # Analysis logic
    pass
def visualize_data(data):
    # Visualization logic
    pass
def process_data(data):
    data = preprocess_data(data)
    data = analyze_data(data)
    visualize_data(data)
```

gradamic.com

5. Function Naming and Documentation

Use descriptive and meaningful names for functions that clearly indicate their purpose. Add docstrings to explain what each function does, its parameters, and return values. This improves code understanding and maintainability.

****Example:****

```python
def calculate_average(numbers):
    """
    #Calculate the average of a list of numbers.
    Args:
      numbers (list): A list of numbers.
    Returns:
      float: The average of the numbers.
    """
    if not numbers:
        return 0.0
    return sum(numbers) / len(numbers)
```

6. Organizing Functions within Scripts or Modules

Arrange functions within scripts or modules in a logical order, such as grouping related functions together or following a flow of execution. This makes it easier for developers to navigate and understand your code.

****Example:****

```python
# data_processing.py
def load_data(filename):
    # Loading data from file
    pass
def preprocess_data(data):
    # Preprocessing steps
    pass
def analyze_data(data):
    # Analysis logic
    pass
def visualize_data(data):
    # Visualization logic
    pass
def main():
    data = load_data('data.csv')
    data = preprocess_data(data)
    analyze_data(data)
    visualize_data(data)
```

ALL COPYRIGHTS 2024 RESERVED BY: GRADAMIC

```
if __name__ == "__main__":
    main()
```

7. Testing Functions

Separate functions responsible for business logic from those handling input/output or interactions with external systems. *This facilitates unit testing of core functionalities.*

Example:

```python
# business_logic.py
def calculate_profit(revenue, expenses):
    # Business logic to calculate profit
    return revenue - expenses
# test_business_logic.py
import unittest
from business_logic import calculate_profit
class TestCalculateProfit(unittest.TestCase):
    def test_calculate_profit(self):
        self.assertEqual(calculate_profit(1000, 500), 500)
        self.assertEqual(calculate_profit(500, 1000), -500)

if __name__ == "__main__":
    unittest.main()
```

8. Using Decorators and Higher-Order Functions

Utilize decorators and higher-order functions to add behavior or modify existing functions without changing their core implementation. This promotes code reuse and separation of concerns.

Example:

```python
def logging_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"Calling function '{func.__name__}' with args {args} and kwargs {kwargs}.")
        return func(*args, **kwargs)
    return wrapper

@logging_decorator
def add(x, y):
    return x + y

result = add(3, 5)  # Output: Calling function 'add' with args (3, 5) and kwargs {}.
```

app.gradamic.com

Conclusion

Organizing Python code using functions is essential for building maintainable and scalable applications.

By breaking down tasks into smaller functions, encapsulating functionality, and using best practices such as naming conventions and documentation, you can create clear, understandable, and efficient code that is easier to debug, test, and maintain over time.

app.gradamic.com

ALL COPYRIGHTS 2024 RESERVED BY: GRADAMIC