FEDERAL UNIVERSITY OF TECHNOLOGY, AKURE
School of Computing | Department of Data Science

DTS 104 · 3 Units · Second Semester 2024/2025

Introduction to Python Programming

A Complete Study Notes Guide

From Basics to Functions, Classes, and Modules

- Python Overview & Basic Principles
- String Data & Operations
- Numeric Data & Type Conversion
- Lists, Tuples, and Dictionaries
- Input / Output & File Handling
- Programming: Assignments, Control Flow, Loops
- Functions, Scope, Lambda, map & filter
- Modules, OOP, and Exception Handling

DTS 104 | by Resolutefemi

Comprehensive Python Study Notes
Based on Introduction to Python Programming by Phil Spector, UC Berkeley

Table of Contents

Introduction to Python

Python is a high-level scripting language which can be used for a wide variety of text processing, system administration and internet-related tasks. Unlike many similar languages, its core language is very small and easy to master, while allowing the addition of modules to perform a virtually limitless variety of tasks. Python is a true object-oriented language, and is available on a wide variety of platforms.

Python was developed in the early 1990s by Guido van Rossum, then at CWI in Amsterdam, and currently at CNRI in Virginia. In some ways, Python grew out of a project to design a computer language which would be easy for beginners to learn, yet would be powerful enough for even advanced users. This heritage is reflected in Python's small, clean syntax and the thoroughness of the implementation of ideas like object-oriented programming, without eliminating the ability to program in a more traditional style.

> *Although pictures of snakes often appear on Python books and websites, the name is derived from Guido van Rossum's favorite TV show, "Monty Python's Flying Circus".*

The Very Basics of Python

Python statements do not need to end with a special character — the interpreter knows that you are done with an individual statement by the presence of a newline. If a statement spans more than one line, the safest course of action is to use a backslash (\) at the end of the line to let Python know that you are going to continue the statement on the next line.

Python uses indentation (the amount of white space before the statement itself) to indicate the presence of loops, instead of using delimiters like curly braces ({}) or keywords (like "begin" and "end") as in many other languages. The first statement of your programs must start in the very first column, and within each indented block, you must be consistent in the amount of indentation you use.

Indentation Example

```
# Example: indentation defines blocks
x = 10
if x > 5:
    print("x is greater than 5")
    print("This is inside the if block")
print("This is outside the if block")
```

Invoking Python

To run a Python program stored in a file, you can simply type "python" followed by the file name at the operating system prompt. After creating a file (say myprogram.py), you would make the file executable (through the UNIX command "chmod +x myprogram.py"), and then you could execute your program by simply typing "myprogram.py" at the UNIX prompt.

When running Python interactively, you can instruct Python to execute files containing Python programs with the execfile function:

```
execfile("myprog.py")
```

Basic Principles of Python

Python has many features that usually are found only in languages which are much more complex to learn and use. These features were designed into Python from its very first beginnings, rather than being accumulated into an end result.

Basic Core Language

The core language is intentionally small. It includes basic types (numbers, strings, lists, tuples, dictionaries), control structures (if, for, while), functions, and classes. Everything else is in modules that you import as needed.

Modules

A module is a file containing Python definitions and statements. Modules extend the core language with additional functions, classes, and variables. You import modules with the import statement:

```
import math
print(math.sqrt(16))   # 4.0
print(math.pi)          # 3.14159...
```

Object-Oriented Programming

Python is a true object-oriented language. Everything in Python is an object — numbers, strings, lists, functions, and even classes themselves. Python supports classes, inheritance, polymorphism, and encapsulation.

Namespaces and Variable Scoping

Python uses the LEGB rule for name resolution: Local (inside the current function), Enclosing (in enclosing function scopes), Global (at module level), Built-in (Python's built-in names).

Exception Handling

Python uses try/except blocks to handle errors gracefully. Exceptions can be caught, handled, and raised. The finally clause always runs, whether or not an exception occurred.

```
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
finally:
    print("This always runs.")
```

String Data

Strings are a collection of characters which are stored together to represent arbitrary text inside a Python program. You can create a string constant by surrounding text with either single quotes ('), double quotes (''), or a collection of three of either types of quotes (''' or """).

String Constants

Creating Strings

```
name = 'Phil'
value = "$7.00"
helptext = """You can create long strings of text
spanning several lines by using triple quotes
at the beginning and end of the text"""
```

In the first two cases, the opening and closing quotes must appear on the same line. When you use triple quotes, your text can span as many lines as you like. The choice of which quote symbol to use is up to you — both single and double quotes have the same meaning in Python.

Special Characters and Raw Strings

The backslash (\) is the escape character in Python strings. Common escape sequences include \n (newline), \t (tab), \\ (literal backslash), \' (single quote), \" (double quote).

Raw strings (prefixed with r or R) treat backslashes as literal characters, which is useful for regular expressions and Windows file paths:

```
path = r'C:\\Users\\Phil\\Documents'
print(path)  # C:\\Users\\Phil\\Documents (backslashes preserved)
```

String Operations

Concatenation

The plus operator (+) joins strings together:

```
greeting = 'Hello' + ' ' + 'World'
print(greeting)  # Hello World
```

Repetition

The asterisk operator (*) repeats a string:

```
print('ab' * 3)  # abababab
```

Indexing and Slicing

Python strings are 0-indexed. Negative indices count from the end. Slicing uses [start:stop:step] where stop is exclusive:

```
s = 'Hello, World!'
print(s[0])      # H (first character)
print(s[-1])     # ! (last character)
print(s[0:5])    # Hello (indices 0-4)
print(s[7:])     # World! (from index 7 to end)
print(s[:5])     # Hello (from start to index 4)
print(s[::2])    # HloWrd (every 2nd char)
print(s[::-1])   # !dlroW ,olleH (reversed)
```

Functions and Methods for Strings

Numeric Data

Python supports four types of numeric objects: integers, long integers, floating point numbers, and complex numbers. In general, Python will not automatically convert numbers from one type to another, although it provides a complete set of functions to allow you to explicitly do these conversions.

Types of Numeric Data

Hexadecimal and Octal Constants

Hexadecimal constants are prefixed with 0x (or 0X). Octal constants begin with a leading zero (0).

```
print(0xFF)    # 255 (hexadecimal)
print(0o777)   # 511 (octal, Python 3 syntax)
```

Numeric Operators

Conversion of Scalar Types

Python provides functions for explicit type conversion between numeric types and strings:

Type Conversion Examples

```
x = int('42')       # string to integer: 42
y = float('3.14')    # string to float: 3.14
z = str(42)          # integer to string: '42'
w = int(3.99)        # float to integer: 3 (truncates)
v = float(5)         # integer to float: 5.0
```

Note: int() truncates toward zero — it does NOT round. Use round(3.99) → 4 if you want rounding.

Lists, Tuples, and Dictionaries

List Data

Lists provide a general mechanism for storing a collection of objects indexed by a number in Python. The elements of the list are arbitrary — they can be numbers, strings, functions, user-defined objects, or even other lists, making complex data structures very simple to express.

Creating Lists

```
mylist = [1, 7, 9, 13, 22, 31]
empty = []
mixed = [1, "hello", 3.14, [2, 4], len]
```

Python ignores spaces between the entries of a list. If you need to span multiple lines with a list entry, you can simply hit the return key after any comma in the list. The elements of a list need not be of the same type.

List Indexing and Slicing

List indexing and slicing work exactly the same way as string indexing — 0-based, negative indices from the end, [start:stop:step] with stop exclusive:

```
mylist = [10, 20, 30, 40, 50]
print(mylist[0])      # 10
print(mylist[-1])     # 50
print(mylist[1:4])    # [20, 30, 40]
print(mylist[::-1])   # [50, 40, 30, 20, 10]
```

List Operators

Concatenation (+)

```
[1, 2] + [3, 4]   # [1, 2, 3, 4]
```

Repetition (*)

```
[0] * 3          # [0, 0, 0]
```

The in operator

```
3 in [1, 2, 3]   # True
5 in [1, 2, 3]   # False
```

Functions and Methods for Lists

List Method Examples

```
nums = [3, 1, 4, 1, 5, 9, 2, 6]
nums.append(5)       # [3, 1, 4, 1, 5, 9, 2, 6, 5]
nums.sort()          # [1, 1, 2, 3, 4, 5, 5, 6, 9]
nums.reverse()       # [9, 6, 5, 5, 4, 3, 2, 1, 1]
print(len(nums))     # 9
print(max(nums))     # 9
print(sum(nums))     # 36
```

Tuple Objects

Tuples are very similar to lists, but they are immutable — once created, they cannot be modified. Tuples are created using parentheses instead of square brackets:

```
mytuple = (1, 2, 3)
single = (42,)       # note the comma for a single-element tuple
empty = ()
print(mytuple[0])     # 1 (indexing works like lists)
print(mytuple[1:])    # (2, 3) (slicing works like lists)
```

Because tuples are immutable, they have fewer methods than lists — only count() and index(). Tuples are useful for: fixed records that should not change (e.g., coordinates (lat, lon)); dictionary keys (lists cannot be dict keys, but tuples can); function return values when returning multiple items.

Dictionaries

Dictionaries (sometimes referred to as associative arrays or hashes) are similar to lists in that they can contain arbitrary objects and can be nested to any desired depth. But instead of being indexed by integers, they can be indexed by any immutable object, such as strings or tuples. The index is called the "key".

Dictionary Example

```
phonedict = {'Fred': '555-1231', 'Andy': '555-1195', 'Sue': '555-2193'}
print(phonedict['Sue'])    # 555-2193
phonedict['Bob'] = '555-9999'   # add a new entry
del phonedict['Fred']            # remove an entry
```

Functions and Methods for Dictionaries

Input and Output

The print Command

The fastest way to print the value of an object in Python is with the print command. You provide the command with a comma-separated list of objects, and it displays them with spaces between each object, and a newline added at the end.

```
print("Hello, World!")
print("Name:", "Phil", "Age:", 30)
print("No newline", end="")
print(" continues here")
```

Formatting Strings

String formatting is performed through overloading of the percent operator (%) for string values. On the left hand side of the operator, you provide a string which is a combination of literal text and formatting codes. Each appearance of a formatting code is matched with an object in a tuple, which appears on the right hand side of the percent sign.

String Formatting Examples

```
name = 'Phil'
age = 30
height = 5.9
print('Name: %s, Age: %d, Height: %.1f' % (name, age, height))
# Output: Name: Phil, Age: 30, Height: 5.9

# Python 3 also supports f-strings (preferred):
print(f'Name: {name}, Age: {age}, Height: {height:.1f}')
```

File Objects

To work with files, use the open() function, which returns a file object. The first argument is the file name; the second (optional) argument is the mode:

Methods for Reading

```
f = open('data.txt', 'r')
content = f.read()        # read entire file as one string
line = f.readline()        # read one line (includes \n)
lines = f.readlines()      # read all lines into a list
for line in f:             # iterate line by line (memory efficient)
    print(line.strip())
f.close()
```

Methods for Writing

```
f = open('output.txt', 'w')
f.write('Hello, World!\n')
f.write('Second line\n')
f.writelines(['a\n', 'b\n', 'c\n'])
f.close()
```

Always close files when done, or use the "with" statement for automatic closing: with open("file.txt") as f: data = f.read() — the file is closed automatically when the with block exits, even if an exception occurs.

Standard Input and Output Streams

The sys module provides access to standard input (sys.stdin), standard output (sys.stdout), and standard error (sys.stderr). The input() function reads a line from standard input:

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

import sys
for line in sys.stdin:    # read from piped input
    print(line.strip())
```

Programming Constructs

Assignments

The assignment statement associates a variable name with a value. The equal sign (=) is used to assign a value to a variable; the variable name is put on the left, and the value (any valid Python expression) is put on the right.

Assignment Examples

```
x = 3
y = "This is a string"
z = x + 3
vegies = ["broccoli", "peas", "carrots"]

# Chained assignment
count = number = check = 0

# Augmented assignment
x += 5    # equivalent to x = x + 5
x -= 2    # x = x - 2
x *= 3    # x = x * 3
x /= 2    # x = x / 2

# Multiple assignment (tuple unpacking)
a, b, c = 1, 2, 3
a, b = b, a    # swap values
```

Indentation

Unlike most programming languages, where indentation is used simply to improve readability, Python uses indentation to signal the beginning and end of blocks of statements. The first statement of your programs must start in the very first column, and within each indented block, you must be consistent in the amount of indentation you use.

> *Python-aware editors like emacs, vim, IDLE, and PythonWin handle indentation automatically. Mixing tabs and spaces can cause IndentationError — configure your editor to use spaces (PEP 8 recommends 4 spaces per indent level).*

Truth, Falsehood and Logical Operators

The following values are considered "false" in a boolean context: False, 0, 0.0, empty string '', empty list [], empty tuple (), empty dictionary {}, and None. Everything else is "true".

Python uses keyword operators (not symbols) for logical operations:

The if Statement

if/elif/else Example

```
score = 85

if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
elif score >= 60:
    grade = "D"
else:
    grade = "F"

print(f"Grade: {grade}")   # Grade: B
```

for Loops

The for loop iterates over the elements of a sequence (list, tuple, string, dictionary, or range).

for Loop Examples

```
# Iterate over a list
for fruit in ["apple", "banana", "cherry"]:
    print(fruit)

# Iterate over a range
for i in range(5):        # 0, 1, 2, 3, 4
    print(i)

for i in range(2, 8):     # 2, 3, 4, 5, 6, 7
    print(i)

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

# enumerate for index + value
for i, fruit in enumerate(["apple", "banana"]):
    print(f"{i}: {fruit}")

# zip for parallel iteration
names = ["Phil", "Sue"]
ages = [30, 25]
for name, age in zip(names, ages):
    print(f"{name} is {age}")
```

while Loops

The while loop repeats a block as long as its condition is True. Use while when you don't know in advance how many iterations are needed.

while, break, continue

```
count = 0
while count < 5:
    print(count)
    count += 1

# break and continue
for i in range(10):
    if i == 3:
        continue    # skip 3
    if i == 7:
        break        # exit loop at 7
    print(i)
# Output: 0, 1, 2, 4, 5, 6
```

Functions

Functions allow you to encapsulate a block of code that performs a specific task, so you can reuse it without duplicating the code. Functions are defined using the def keyword.

Defining and Calling Functions

Basic Function

```
def greet(name):
    """Return a greeting string."""  # docstring
    return f"Hello, {name}!"

message = greet("Phil")
print(message)    # Hello, Phil!
```

Default Arguments

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

print(greet("Phil"))              # Hello, Phil!
print(greet("Phil", "Hi"))        # Hi, Phil!
```

Variable Arguments: *args and **kwargs

```
def f(*args, **kwargs):
    print("args:", args)        # tuple of positional arguments
    print("kwargs:", kwargs)    # dict of keyword arguments

f(1, 2, 3, x=10, y=20)
# args: (1, 2, 3)
# kwargs: {'x': 10, 'y': 20}
```

Variable Scope and the global Keyword

Variables defined inside a function are local to that function. Variables defined at module level are global. To modify a global variable from inside a function, declare it with the global keyword.

```
count = 0    # global variable

def increment():
    global count
    count += 1

increment()
increment()
print(count)    # 2
```

Lambda, map, and filter

The lambda keyword creates an anonymous (inline) function. map() applies a function to each element of a sequence. filter() returns only elements for which a function returns True.

Functional Programming Examples

```
# lambda
square = lambda x: x ** 2
print(square(5))    # 25

# map
nums = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, nums))
print(squared)      # [1, 4, 9, 16]

# filter
nums = [-3, 7, 12, -2, 19]
positives = list(filter(lambda x: x > 0, nums))
print(positives)    # [7, 12, 19]

# List comprehension (often preferred over map/filter)
squared = [x**2 for x in nums]               # like map
positives = [x for x in nums if x > 0]      # like filter
both = [x**2 for x in nums if x > 0]        # map + filter
```

Returning Multiple Values

```
def min_max(lst):
    return min(lst), max(lst)

lo, hi = min_max([3, 1, 4, 1, 5, 9])
print(f"Min: {lo}, Max: {hi}")    # Min: 1, Max: 9
```

Modules

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. To use a module, import it.

Importing Modules

```
# Import the entire module
import math
print(math.sqrt(16))    # 4.0
print(math.pi)           # 3.14159...

# Import specific names
from math import sqrt, pi
print(sqrt(25))          # 5.0
print(pi)                # 3.14159...

# Alias a module
import numpy as np
arr = np.array([1, 2, 3])

# Import all names (discouraged — pollutes namespace)
from math import *
```

Common Standard Library Modules

Creating Your Own Module

Creating a Module

```
# File: mymath.py
def square(x):
    return x ** 2

def cube(x):
    return x ** 3

PI = 3.14159
```

Using Your Module

```
# File: main.py
import mymath
print(mymath.square(5))     # 25
print(mymath.cube(3))       # 27
print(mymath.PI)            # 3.14159

from mymath import square, cube
print(square(7))            # 49
```

Object-Oriented Programming

Python is a true object-oriented language. Everything in Python is an object — numbers, strings, lists, functions, and even classes themselves are objects. Python supports classes, inheritance, polymorphism, and encapsulation.

Defining Classes

Class Definition Example

```
class Dog:
    species = "Canis familiaris"   # class attribute

    def __init__(self, name, age):  # constructor
        self.name = name            # instance attribute
        self.age = age

    def bark(self):                 # instance method
        return f"{self.name} says Woof!"

    def __str__(self):              # string representation
        return f"Dog({self.name}, {self.age} years old)"

# Create instances
rex = Dog("Rex", 5)
buddy = Dog("Buddy", 3)

print(rex.bark())      # Rex says Woof!
print(rex.species)     # Canis familiaris
print(rex)             # Dog(Rex, 5 years old)
```

Inheritance

Inheritance allows a new class (child) to inherit attributes and methods from an existing class (parent). The child class can override or extend the parent's behavior.

Inheritance Example

```
class Puppy(Dog):   # Puppy inherits from Dog
    def __init__(self, name, age, toy):
        super().__init__(name, age)   # call parent constructor
        self.toy = toy

    def play(self):
        return f"{self.name} plays with {self.toy}"

    def bark(self):   # override parent method
        return f"{self.name} says Yip!"

puppy = Puppy("Max", 1, "ball")
print(puppy.bark())    # Max says Yip!
print(puppy.play())    # Max plays with ball
print(puppy.age)       # 1 (inherited from Dog)
```

Key OOP Concepts

- **Class:** A blueprint for creating objects. Defined with the class keyword.
- **Instance:** A specific object created from a class. Created by calling the class like a function: Dog("Rex", 5).
- **Attribute:** A variable belonging to an object (instance attribute) or class (class attribute).
- **Method:** A function belonging to a class. The first parameter is conventionally self.
- **Constructor (__init__):** A special method called when an instance is created.
- **Inheritance:** A child class inherits attributes and methods from a parent class.
- **Encapsulation:** Bundling data and methods that operate on that data within a class.
- **Polymorphism:** Different classes can implement the same method differently (e.g., bark() in Dog vs Puppy).

Exception Handling

Exceptions are errors that occur during program execution. Python uses try/except blocks to handle them gracefully, allowing your program to recover from errors instead of crashing.

try/except Blocks

Complete Exception Handling

```
try:
    x = int(input("Enter a number: "))
    result = 10 / x
    print(f"10 / {x} = {result}")
except ValueError:
    print("That's not a valid number!")
except ZeroDivisionError:
    print("Cannot divide by zero!")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
else:
    print("No exception occurred.")
finally:
    print("This always runs (cleanup).")
```

Common Built-in Exceptions

Raising Exceptions

```
def set_age(person, age):
    if age < 0:
        raise ValueError("Age cannot be negative")
    person["age"] = age

try:
    set_age({"name": "Phil"}, -5)
except ValueError as e:
    print(e)    # Age cannot be negative
```

Best Practices

- Catch specific exceptions, not bare except (which catches everything including KeyboardInterrupt).
- Handle the exception — don't silently pass (empty except block hides bugs).
- Use finally for cleanup (closing files, releasing resources) or use the "with" statement.
- Raise meaningful exceptions with descriptive messages.
- Create custom exception classes by subclassing Exception for domain-specific errors.

Key Terms and Definitions

A condensed glossary of the Python terms covered in this guide. Memorise the definition and a canonical example for each.

Term	Definition
Variable	A name associated with a value, created by assignment (x = 5).
String	A sequence of characters, created with quotes: 'hello' or "hello" or """multi-line""".
Integer (int)	A whole number: 42, -17, 0. Range approx -2B to +2B on most computers.
Float	A number with a decimal point or exponential notation: 3.14, 1e3.
Boolean (bool)	A truth value: True or False (capitalised).
List	An ordered, mutable collection: [1, 2, 3]. Elements need not be the same type.
Tuple	An ordered, immutable collection: (1, 2, 3). Useful for fixed records.
Dictionary (dict)	A key-value collection: {"name": "Phil", "age": 30}. Indexed by keys.
Set	An unordered collection of unique elements: {1, 2, 3}.
Indexing	Accessing an element by position: mylist[0] returns the first element.
Slicing	Extracting a sub-sequence: s[1:4] returns elements at indices 1, 2, 3.
Method	A function that belongs to an object: "hello".upper() returns "HELLO".
Function	A reusable block of code defined with def: def add(a, b): return a + b.
Parameter	A variable in a function definition that receives a value (argument) when called.
Argument	A value passed to a function when calling it: add(3, 4) — 3 and 4 are arguments.
Return	The value a function produces: return x + y. Functions without return return None.
Module	A file containing Python code, imported with import: import math.
Class	A blueprint for creating objects, defined with class: class Dog:.
Object / Instance	A specific entity created from a class: rex = Dog("Rex", 5).
Constructor (__init__)	A special method called when an instance is created.
Inheritance	A child class inheriting from a parent: class Puppy(Dog):.
Exception	An error that occurs during execution, handled with try/except.
Indentation	Whitespace at the start of a line; Python uses it to define code blocks.
Comment	Text preceded by # that Python ignores: # This is a comment.
Docstring	A triple-quoted string documenting a function/class, accessible via __doc__.

Term	Definition
Lambda	An anonymous inline function: lambda x: x**2.
List Comprehension	A concise way to create lists: [x**2 for x in range(5)].
Global Variable	A variable defined at module level, accessible throughout the module.
Local Variable	A variable defined inside a function, accessible only within that function.
None	Python's null value, representing the absence of a value.

Table 11 — Glossary of Python Terms

Summary and Quick Review

Use this page as a final revision sheet. Each item is one sentence — the absolute minimum you must remember.

Concepts in One Line Each

- **Python** = high-level, interpreted, object-oriented scripting language with small clean syntax.
- **Indentation** = Python uses whitespace (not braces) to define code blocks; be consistent.
- **Strings** = sequences of characters; created with ' ', " ", or ''' '''; 0-indexed; [start:stop:step] slicing.
- **Numeric types** = int (whole), long (arbitrary precision), float (decimal), complex (a+bj).
- **Lists** = ordered, mutable, heterogeneous; [ ]; append, extend, insert, remove, pop, sort, reverse.
- **Tuples** = ordered, immutable; ( ); useful for fixed records and dict keys.
- **Dictionaries** = key-value pairs; { }; indexed by immutable keys; O(1) lookup.
- **print()** = displays values; f-strings (f'...') for formatting.
- **Files** = open(path, mode); read/readline/readlines; write; close; use "with" for auto-close.
- **Assignment** = (=) assigns; chained (a=b=c=0); augmented (+=, -=, *=, /=); tuple unpacking (a,b=b,a).
- **Booleans** = True, False; falsy: 0, '', [], (), {}, None; operators: and, or, not (keywords).
- **if/elif/else** = conditional; colon (:) introduces indented block.
- **for loop** = iterates sequences; range(n)=0..n-1; enumerate for index+value; zip for parallel.
- **while loop** = repeats while condition True; break exits; continue skips.
- **Functions** = def name(params): body; default args (y=10); *args, **kwargs; return multiple as tuple.
- **Scope** = LEGB (Local, Enclosing, Global, Built-in); global keyword to modify globals.
- **lambda** = inline function; map(f, lst) applies f; filter(f, lst) keeps True; list comprehensions preferred.
- **Modules** = import module; from module import name; math, sys, os, random, json, re common.
- **Classes** = class Name:; __init__ constructor; self refers to instance; inheritance: class Child(Parent):.
- **Exceptions** = try/except/else/finally; catch specific types; raise manually; finally for cleanup.

Key Code Snippets

End of DTS 104 Python study notes. Best of luck in your examination.