Cheatsheet

Programming with Python

Dr. Tobias Vlćek

Lecture I

Python Basics

- Python is an interpreted language - code is executed line by line
- Comments start with `#`
- Code blocks are defined by indentation

Variables

- Created using assignment operator `=`
- Must start with letter or underscore
- Case sensitive
- Cannot use reserved words

Data Types

1. Strings (str)
   - Enclosed in quotes: `"Hello"` or `'Hello'`
   - F-strings: `f"Value is {variable}"`
   - Format: `f"{variable:<width>.<precision>f}"`
2. Numbers
   - Integers (int): `1`, `-3`, `0`
   - Floats: `-4.78`, `0.1`, `1.23e2`
3. Booleans (bool)
   - `True` or `False`

Basic Operators

```
# Arithmetic
addition = 1 + 2 # 3
subtraction = 1 - 2 # -1
multiplication = 3 * 4 # 12
division = 7 / 4 # 1.75
floor_division = 7 // 4 # 1
exponentiation = 9 ** 0.5 # 3.0
modulo = 10 % 3 # 1
```

Common Functions

```
# Input/Output
print("Hello")                    # Display output
name = input("Enter name: ")      # Get user input

# Type Conversion
int_val = int("123")              # String to integer
float_val = float("12.34")        # String to float
str_val = str(123)                # Number to string

# Other
round(3.14159, 2)                 # Round to 2 decimals
len("Hello")                      # Get length
type(variable)                    # Get type of variable
```

Best Practices

1. Use meaningful variable names
2. Add comments to explain complex code
3. Follow Python's naming conventions
4. Use f-strings for string formatting
5. Be consistent with quote usage (" or ')

Lecture II

String Methods

```
text = "Hello, World!"
text.upper()      # Convert to uppercase: "HELLO, WORLD!"
text.lower()      # Convert to lowercase: "hello, world!"
text.title()      # Title case: "Hello, World!"
text.strip()      # Remove leading/trailing whitespace
text.replace("Hello", "Hi")  # Replace text: "Hi, World!"
text.find("World")          # Find substring index: 7
text.count("l")            # Count occurrences: 3
```

Indexing and Slicing

```
text = "Hello, World!"
text[0]      # First character: "H"
text[-1]     # Last character: "!"
text[7:12]   # Slice: "World"
text[::2]    # Every second character: "Hlo ol!"
text[::-1]   # Reverse string: "!dlroW ,olleH"
```

Comparison Operators

- `==` Equal to
- `!=` Not equal to
- `<` Less than

- `>` Greater than
- `<=` Less than or equal to
- `>=` Greater than or equal to

Logical Operators

```
# and: Both conditions must be True
x > 0 and x < 10    # True if x is between 0 and 10

# or: At least one condition must be True
x < 0 or x > 10     # True if x is outside 0-10

# not: Inverts the condition
not x == 10         # True if x is not 10
```

Membership Operators

```
"a" in "apple"          # True
"z" not in "apple"      # True
```

Control Structures

If Statements

```
if condition:
    # code if condition is True
elif other_condition:
    # code if other_condition is True
else:
    # code if all conditions are False
```

For Loops

```
# Loop with range
for i in range(5):          # 0 to 4
    print(i)

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

# Loop through string
for char in "Hello":
    print(char)
```

While Loops

```
# Basic while loop
i = 0
while i < 5:
    print(i)
    i += 1
```

```python
# While loop with break
while True:
    if condition:
        break    # Exit loop
```

Best Practices

1. Use clear and descriptive variable names
2. Maintain consistent indentation (4 spaces)
3. Use comments to explain complex logic
4. Avoid infinite loops
5. Keep code blocks focused and manageable

Lecture III

Functions

Basic Function Syntax

```python
def function_name(parameter1, parameter2):
    """Docstring explaining what the function does"""
    # Function body
    return result
```

Function Parameters

```python
# No parameters
def greet():
    print("Hello!")

# Multiple parameters
def greet(name, age):
    print(f"Hello {name}, you are {age} years old")

# Default parameters
def greet(name="Stranger"):
    print(f"Hello {name}")

# Keyword arguments
greet(name="Alice")  # Calling with named parameter
```

Return Values

```python
# Return single value
def multiply(a, b):
    return a * b

# Return None (implicit)
def greet(name):
```

```
    print(f"Hello {name}")
    # No return statement = returns None

# Multiple return points
def check_number(n):
    if n > 0:
        return "positive"
    elif n < 0:
        return "negative"
    return None  # if n == 0
```

Function Scope

```
# Global scope
global_var = 10

def function():
    # Local scope
    local_var = 20
    print(global_var)  # Can access global

    # Modify global variable
    global global_var
    global_var = 30
```

Classes

Basic Class Syntax

```
class ClassName:
    # Class attribute
    class_attribute = value

    # Constructor
    def __init__(self, parameter):
        # Instance attribute
        self.instance_attribute = parameter

    # Method
    def method_name(self):
        return self.instance_attribute
```

Class Example

```
class Student:
    # Class attribute
    school = "Python University"

    def __init__(self, name):
        # Instance attribute
        self.name = name
```

```
    def introduce(self):
        return f"Hi, I'm {self.name}"

# Create instance
student = Student("Alice")
print(student.introduce())  # "Hi, I'm Alice"
```

Inheritance

```
class Parent:
    def method(self):
        print("Parent method")

class Child(Parent):
    def method(self):
        print("Child method")
```

Best Practices

1. Use descriptive function and class names
2. Write clear docstrings
3. Keep functions focused on a single task
4. Use meaningful parameter names
5. Follow Python naming conventions:
   - function_name (snake_case)
   - ClassName (PascalCase)
   - variable_name (snake_case)

Lecture IV

Tuples

```
# Creating tuples
my_tuple = (1, 2, 3)           # Using parentheses
my_tuple = 1, 2, 3             # Using just commas
my_tuple = tuple([1, 2, 3])    # Using tuple() function

# Tuple operations
my_tuple[0]                    # Accessing elements
my_tuple[1:3]                  # Slicing
my_tuple + (4, 5, 6)          # Concatenation
my_tuple * 2                   # Repetition

# Tuple methods
my_tuple.count(2)              # Count occurrences
my_tuple.index(3)              # Find index of element

# Tuple unpacking
name, age, city = my_tuple     # Basic unpacking
name, *rest = my_tuple         # Using * for remaining elements
```

Lists

```
# Creating lists
my_list = [1, 2, 3]           # Using square brackets
my_list = list((1, 2, 3))     # Using list() function

# Common list methods
my_list.append(4)             # Add element to end
my_list.insert(0, 0)          # Insert at index
my_list.remove(1)             # Remove first occurrence
my_list.pop()                 # Remove and return last element
my_list.sort()                # Sort in place
my_list.reverse()             # Reverse in place
my_list.count(2)              # Count occurrences
my_list.index(3)              # Find index of element
```

Sets

```
# Creating sets
my_set = {1, 2, 3}            # Using curly braces
my_set = set([1, 2, 3])       # Using set() function

# Common set methods
my_set.add(4)                 # Add element
my_set.remove(1)              # Remove element (raises error if not found)
my_set.discard(1)             # Remove element (no error if not found)
my_set.pop()                  # Remove and return arbitrary element
my_set.update({4, 5, 6})      # Add multiple elements

# Set operations
set1.union(set2)              # Union of sets
set1.intersection(set2)       # Intersection of sets
set1.isdisjoint(set2)         # Check if sets have no common elements
set1.issubset(set2)           # Check if set1 is subset of set2
```

Dictionaries

```
# Creating dictionaries
my_dict = {"name": "John", "age": 30}    # Using curly braces
my_dict = dict(name="John", age=30)      # Using dict() function

# Dictionary operations
my_dict["name"]               # Access value by key
my_dict["city"] = "Hamburg"   # Add or update key-value pair
del my_dict["age"]            # Remove key-value pair
"name" in my_dict             # Check if key exists

# Dictionary methods
my_dict.keys()                # Get all keys
my_dict.values()              # Get all values
my_dict.items()               # Get all key-value pairs
my_dict.get("name")           # Safe way to get value
my_dict.pop("name")           # Remove and return value
```

File Handling

```
# Basic file operations
file = open("file.txt", "r")  # Open for reading
file = open("file.txt", "w")  # Open for writing
file = open("file.txt", "a")  # Open for appending

# Reading files
content = file.read()         # Read entire file
lines = file.readlines()      # Read lines into list

# Writing files
file.write("Hello")           # Write string to file
file.writelines(lines)        # Write list of strings

# Using with statement (recommended)
with open("file.txt", "r") as file:
    content = file.read()
```

Data Type Comparison

- Tuples: Immutable, ordered, allows duplicates
- Lists: Mutable, ordered, allows duplicates
- Sets: Mutable, unordered, no duplicates
- Dictionaries: Mutable, unordered, unique keys

Best Practices

1. Use tuples for immutable sequences
2. Use lists when order matters and items need to be modified
3. Use sets for unique collections
4. Use dictionaries for key-value relationships
5. Always use `with` statement for file operations
6. Close files after use if not using `with`

Lecture V

Common Built-in Exceptions

- ValueError: Wrong value type (e.g., converting “hello” to int)
- TypeError: Wrong operation for type (e.g., “hello” + 5)
- NameError: Variable not found
- IndexError: List index out of range
- KeyError: Dictionary key not found
- FileNotFoundError: File/directory not found
- ZeroDivisionError: Division by zero
- AttributeError: Object has no attribute/method
- ImportError: Module import fails
- SyntaxError: Invalid Python syntax
- IndentationError: Incorrect indentation

- RuntimeError: Generic runtime error

Try-Except Blocks

```
# Basic try-except
try:
    result = risky_operation()
except Exception as e:
    print(f"Error occurred: {e}")

# Multiple exception handling
try:
    result = risky_operation()
except ValueError as e:
    print(f"Value error: {e}")
except TypeError as e:
    print(f"Type error: {e}")
except Exception as e:
    print(f"Other error: {e}")
```

Raising Exceptions

```
# Basic raise
def validate_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative")
    return age

# Custom exception
class CustomError(Exception):
    pass

def custom_operation():
    if error_condition:
        raise CustomError("Custom error message")
```

Assertions

```
# Basic assertions
assert condition, "Error message"
assert x > 0, "x must be positive"
assert isinstance(x, int), "x must be integer"

# Common assertion patterns
def process_list(lst):
    assert isinstance(lst, list), "Input must be a list"
    assert all(isinstance(x, int) for x in lst), "All elements must be
integers"
    assert len(lst) > 0, "List cannot be empty"
```

Debugging Tips

1. Print Debugging

```
print(f"Variable x = {x}")
print(f"Type of x: {type(x)}")
print(f"Debug: Entering function {function_name}")
```

2. Assertions for Debugging

```
assert x == expected_value, f"x should be {expected_value}, but got {x}"
```

3. IDE Debugging

- Set breakpoints
- Step through code
- Inspect variables
- Use watch windows

Best Practices

1. Always handle specific exceptions before generic ones
2. Use meaningful error messages
3. Don't catch exceptions without handling them
4. Use assertions for debugging and testing
5. Include relevant information in error messages
6. Clean up resources in try-finally blocks

Common Debugging Workflow

1. Identify the error (error message or unexpected behavior)
2. Locate the source of the error
3. Add print statements or use debugger
4. Test the fix
5. Add error handling if needed

Lecture VI

Modules

Importing Modules

```
# Basic import
import module_name
module_name.function_name()

# Import specific items
from module_name import function_name, another_function
function_name()

# Import with alias
import module_name as alias
alias.function_name()
```

Common Built-in Modules

Module	Description	Common Functions/Constants
`math`	Mathematical functions	`pi`, `sqrt()`, `cos()`
`random`	Random number generation	`random()`, `randint()`
`datetime`	Date and time handling	`datetime`, `timedelta`
`os`	Operating system interaction	`listdir()`, `path.exists()`
`csv`	CSV file operations	`reader()`, `writer()`
`re`	Regular expressions	`search()`, `findall()`

Random Module

```
import random

random.random()           # Float between 0 and 1
random.uniform(1, 10)     # Float between 1 and 10
random.randint(1, 10)     # Integer between 1 and 10
random.choice(list)       # Random item from list
random.shuffle(list)      # Shuffle list in place
```

OS Module

```
import os

os.listdir('path')            # List directory contents
os.path.exists('path')        # Check if path exists
os.path.isfile('path')        # Check if path is file
os.makedirs('path')           # Create directories
os.getcwd()                   # Get current working directory
os.path.join('dir', 'file')   # Join path components
```

CSV Module

```
import csv

# Writing CSV
with open('file.csv', 'w') as file:
    writer = csv.writer(file)
    writer.writerow(['header1', 'header2'])
    writer.writerow(['data1', 'data2'])

# Reading CSV
with open('file.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Regular Expressions (re)

```
import re

# Basic patterns
re.search(pattern, string)    # Search for pattern
re.findall(pattern, string)   # Find all occurrences
re.sub(pattern, repl, string) # Replace pattern
re.split(pattern, string)     # Split string by pattern

# Common special characters
. # Any character
* # Zero or more
+ # One or more
? # Zero or one
[] # Character set
\d # Any digit
\w # Word character
\s # Whitespace
```

Package Management

```
# Installing packages
pip install package_name
pip install package1 package2

# Upgrading packages
pip install --upgrade package_name

# List installed packages
pip list
```

Best Practices

1. Import modules at the beginning of the file
2. Use specific imports instead of importing everything
3. Use meaningful aliases when needed
4. Keep virtual environments project-specific
5. Document package dependencies
6. Use regular expressions carefully and test them thoroughly

---

Lecture VII

NumPy Basics

Creating Arrays

```
import numpy as np

# Basic array creation
arr = np.array([1, 2, 3, 4, 5])
```

```
arr_2d = np.array([[1, 2], [3, 4]])

# Pre-filled arrays
zeros = np.zeros((3, 3))        # Array of zeros
ones = np.ones((2, 2))          # Array of ones
rand = np.random.rand(3, 3)     # Random values
arange = np.arange(0, 10, 2)    # Values from 0 to 10, step 2
linspace = np.linspace(0, 1, 5) # 5 evenly spaced values
```

Array Operations

```
# Basic operations
arr + 1                  # Add 1 to all elements
arr * 2                  # Multiply all elements by 2
arr > 3                  # Boolean comparison

# Array methods
arr.sort()               # Sort array
arr.reshape(2, 3)        # Reshape array
arr.flatten()            # Convert to 1D array
arr.transpose()          # Transpose array
arr.squeeze()            # Remove single-dimensional entries

# Joining arrays
np.concatenate((arr1, arr2))  # Join arrays
np.vstack((arr1, arr2))       # Vertical stack
np.hstack((arr1, arr2))       # Horizontal stack
```

Data Types

```
# Common dtypes
'i'    # integer
'b'    # boolean
'f'    # float
'S'    # string
'U'    # unicode

# Setting dtype
arr = np.array([1, 2, 3], dtype='f')
arr = arr.astype('i')         # Convert type
```

Best Practices

1. Use NumPy for numerical computations
2. Keep array types homogeneous for better performance
3. Use appropriate data types to optimize memory
4. Prefer vectorized operations over loops

Lecture VIII

Pandas Basics

Creating DataFrames

```
import pandas as pd

# From dictionary
df = pd.DataFrame({
    'Name': ['John', 'Anna'],
    'Age': [25, 28]
})

# From CSV/Excel
df = pd.read_csv('file.csv')
df = pd.read_excel('file.xlsx')
```

Basic Operations

```
# Viewing data
df.head()               # First 5 rows
df.tail()               # Last 5 rows
df.info()              # DataFrame info
df.describe()          # Summary statistics

# Accessing data
df['column_name']      # Access column
df.iloc[0]            # Access by position
df.loc['label']       # Access by label

# Filtering
df[df['Age'] > 25]    # Filter by condition
```

Grouping and Aggregation

```
# Basic grouping
df.groupby('column').mean()
df.groupby(['col1', 'col2']).sum()

# Common aggregations
.sum()      # Sum of values
.mean()     # Mean of values
.max()      # Maximum value
.min()      # Minimum value
.count()    # Count of values
```

Reshaping Data

```
# Melting (wide to long)
pd.melt(df, id_vars=['ID'])

# Combining DataFrames
```

```
pd.concat([df1, df2])           # Concatenate
df1.join(df2)                   # Join on index
df1.merge(df2, on='column')     # Merge on column
```

Excel Operations

```
# Reading Excel
df = pd.read_excel('file.xlsx', sheet_name='Sheet1')

# Writing to Excel
df.to_excel('output.xlsx',
            sheet_name='Sheet1',
            index=False)
```

Best Practices

1. Use pandas for structured data analysis
2. Always check data types and missing values