Core Python

**TABLE OF CONTENTS**

Preface 2 Introduction 2

[Modules and Packages 2](#_Toc17188)

[Data Manipulation and Analysis 2](#_Toc17189)

[Data Visualization 2](#_Toc17190)

[Machine Learning 3](#_Toc17191)

[Web Development 3](#_Toc17192)

[Game Development 3](#_Toc17193)

[Installing New Packages 3](#_Toc17194)

[Data Types and Variables 4](#_Toc17195)

[Strings 4](#_Toc17196)

[Operators 4](#_Toc17197)

Sequence Unpacking 5

Conditional Statements 5 if statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5 if-else statement. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5 if-elif-else statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

Loops 5 for loop. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5 while loop . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

Functions 6 Function Definition. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

Function Call. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

Optional Arguments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

Variable-Length Arguments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

Keyword Arguments. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6 Default Argument Values. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

Anonymous Functions (Lambda Functions). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

Variable Scope . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

Lists 6 List methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

Dictionaries 7 Dictionary methods. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

Tuples 7 Sets 7

Set methods. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7 Input and Output 7 Input . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

Output. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

File Handling 7 Opening a file . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

Reading from a file . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

Writing to a file . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

Closing a file . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

Exception Handling 8

Classes and Objects 8 Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

@classmethod decorator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

Inheritance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

Magic methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

Comments and Docstrings. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

# Modules and Packages 10

Modules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

Packages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

|  |  |
| --- | --- |
| Lambda Functions | 10 |
| List Comprehensions | 10 |
| Generators | 10 |
| Decorators | 10 |
| Map, Filter, and Reduce | 11 |

Map. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

Filter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

Reduce . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

String Formatting 11

Regular Expressions 11 Functions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

Metacharacters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

|  |
| --- |
| **PREFACE** |

|  |
| --- |
| **INTRODUCTION** |

This cheatsheet is intended to serve as a quick reference guide for Python programming. It covers some of the most commonly used syntax and features of the language, including data types, control structures, functions, modules, and libraries.

Whether you are a beginner learning Python for the first time or an experienced programmer looking for a quick refresher, this cheatsheet should be a helpful resource for you.

Python is a high-level, interpreted programming language known for its simplicity, readability, and versatility. It was first released in 1991 by Guido van Rossum and has since become one of the most popular programming languages in the world.

Python’s syntax emphasizes readability, with code written in a clear and concise manner using whitespace and indentation to define blocks of code. It is an interpreted language, meaning that code is executed line-by-line rather than compiled into machine code. This makes it easy to write and test code quickly, without needing to worry about the details of low-level hardware.

Python is a general-purpose language, meaning that it can be used for a wide variety of applications, from web development to scientific computing to artificial intelligence and machine learning. Its simplicity and ease of use make it a popular choice for beginners, while its power and flexibility make it a favorite of experienced developers.

Python’s standard library contains a wide range of modules and packages, providing support for everything from basic data types and control structures to advanced data manipulation and visualization. Additionally, there are countless third-party packages available through Python’s package manager, pip, allowing developers to easily extend Python’s capabilities to suit their needs.

Overall, Python’s combination of simplicity, power, and flexibility makes it an ideal language for a wide range of applications and skill levels.

|  |
| --- |
| **MODULES AND PACKAGES** |

## DATA MANIPULATION AND ANALYSIS

#### NumPy

NumPy is a powerful package for scientific computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a wide range of mathematical functions to manipulate and analyze the data. NumPy is a fundamental package for scientific computing with Python and is widely used in the data science and machine learning communities.

#### Pandas

Pandas is a library for data manipulation and analysis. It provides data structures for efficiently storing and querying large data sets, along with a wide range of functions for cleaning, transforming, and analyzing data. Pandas is a key tool for data scientists and analysts working with tabular data.

#### SciPy

SciPy is a Python library for scientific computing and technical computing. It provides a large set of mathematical algorithms and functions for tasks such as optimization, integration, interpolation, signal processing, linear algebra, and more.

## DATA VISUALIZATION

#### Matplotlib

Matplotlib is a plotting library for Python. It provides a range of functions for creating highquality visualizations of data, including line charts, scatter plots, histograms, and more. Matplotlib is widely used in scientific computing, data visualization, and machine learning.

#### Seaborn

Seaborn is a Python data visualization library that is built on top of the popular visualization library, Matplotlib. It provides a high-level interface for creating beautiful and informative statistical graphics. Seaborn is designed to work with data in a Pandas DataFrame and provides a range of tools for visualizing relationships between variables.

## MACHINE LEARNING

#### Scikit-learn

Scikit-learn is a library for machine learning in Python. It provides a range of functions for classification, regression, clustering, and dimensionality reduction, along with tools for model selection and evaluation. Scikit-learn is a popular choice for building and deploying machine learning models in Python.

#### TensorFlow

TensorFlow is a library for building and training machine learning models, particularly deep learning models. It provides support for building and training neural networks, along with a range of tools for model evaluation and deployment. TensorFlow is widely used in the machine learning and data science communities.

## WEB DEVELOPMENT

#### Flask

Flask is a micro web framework for Python. It provides a range of tools for building web applications, including routing, templates, and request handling. Flask is a lightweight and flexible framework that is widely used for building web applications and APIs in Python.

#### Django

Django is a high-level web framework for Python that follows the Model-View-Controller (MVC) architectural pattern. It provides a powerful set of tools for building web applications, including a robust Object-Relational Mapping (ORM) system, a templating engine, and built-in support for handling user authentication and authorization.

## GAME DEVELOPMENT

#### Pygame

Pygame is a set of Python modules for creating video games and multimedia applications. It provides support for graphics, sound, input, and networking, making it easy to create games and interactive applications in Python.

#### Arcade

Arcade is a Python library for creating 2D arcadestyle video games. It is built on top of the Pygame library and provides an easy-to-use framework for building games with modern graphics and sound effects. Its cross-platform support (it works on Windows, Mac, and Linux), its support for both 2D and 3D graphics, and its active community of developers who are constantly creating new games and tools using the library.

## INSTALLING NEW PACKAGES

#### Using pip

pip is the package installer for Python. You can use it to install new libraries by running the following command in your terminal or command prompt: pip install <library-name>

Replace <library-name> with the name of the library you want to install. For example, to install the numpy library, run: pip install numpy

#### Using Anaconda

If you use Anaconda as your Python distribution, you can use the Anaconda Navigator or the command line interface conda to install new libraries. For example, to install the numpy library, run:

conda install numpy

#### Manually

You can download the source code of a library from its website or GitHub repository and install it manually by following the installation instructions provided by the library’s documentation.

Once you have installed a new library, you can import it into your Python code using the import statement. For example, to import the numpy library, you can use:

import numpy

Or you can use an alias to the library name for a shorter reference:

import numpy as np

# DATA TYPES AND VARIABLES

Python is a dynamically typed language. This means that the data type of a variable is determined at runtime based on the value that is assigned to it. In other words, you don’t need to specify the data type of a variable when you declare it, and you can assign values of different data types to the same variable.

x = 5 # x is an integer x = "hello" # x is now a string x = 3.14 # x is now a float

* Integers: whole numbers without decimal points. e.g. 1, 2, 3, 4, 5
* Floats: numbers with decimal points. e.g. 3.14,

4.5, 6.0

* Strings: sequences of characters. e.g. "hello",

'world'

* Booleans: True or False values.
* Variables: containers for storing values. e.g. x =

10, y = "hello"

## STRINGS

Strings are sequences of characters enclosed in single quotes ' or double quotes ". They are one of the fundamental data types in Python and are used to represent text and other types of data that can be represented as a sequence of characters.

Here are some basic operations that can be performed on strings:

# Creating a string

my\_string = "hello world"

# Getting the length of a string length = len(my\_string) # Output: 11

# Accessing individual characters first\_char = my\_string[0] # Output: "h" last\_char = my\_string[-1] # Output: "d"

# Slicing a string substring = my\_string[0:5] # Output: "hello"

# Concatenating strings new\_string = my\_string + "!" # Output: "hello world!"

# Repeating a string repeat\_string = my\_string \* 3 # Output: "hello worldhello worldhello world"

Strings in Python are immutable, which means that once you create a string, you cannot change its contents. However, you can create a new string that contains the modified content.

String formatting is another important feature in Python, which allows you to insert values into a string in a specific format.

name = "Alice" age = 30 greeting = "My name is {} and I am {} years old".format(name, age) print(greeting) # Output: "My name is Alice and I am 30 years old"

# OPERATORS

* Arithmetic operators: +, -, , /, % (modulus),\* (exponentiation)
* Comparison operators: == (equals), != (not equals), >, <, >=, ⇐

•

Logical operators:

and

,

or

,

not

**SEQUENCE UNPACKING**

Sequence unpacking is a feature in Python that

allows you to assign the elements of a sequence

such as a tuple or a list) to individual variables. It

(

provides a convenient way to assign multiple

values to multiple variables in a single statement.

my\_tuple = (1, 2, 3)

a, b, c = my\_tuple

print(a) # Output: 1

print(b) # Output: 2

print(c) # Output: 3

In this example, we define a tuple called

my\_tuple

that contains three elements. We then use sequence

unpacking to assign each element to a separate

variables a, b, and c.

Sequence unpacking also works with lists:

my\_list = [4, 5, 6]

x, y, z = my\_list

print(x) # Output: 4

print(y) # Output: 5

print(z) # Output: 6

You can also use sequence unpacking to swap the

values of two variables without using a temporary

variable:

a = 1

b = 2

a, b = b, a

print(a) # Output: 2

print(b) # Output: 1

**CONDITIONAL STATEMENTS**

**IF STATEMENT**

Executes a block of code if a condition is

true

.

if condition:

# code to execute if condition

is true

**IF-ELSE STATEMENT**

Executes a block of code if a condition is

true

, and

another block if it’s

false

.

if condition:

# code to execute if condition

is true

else:

# code to execute if condition

is false

**IF-ELIF-ELSE STATEMENT**

Executes a block of code based on multiple

conditions.

if condition1:

# code to execute if condition1

is true

elif condition2:

# code to execute if condition2

is true

else:

# code to execute if all

conditions are false

**LOOPS**

**FOR LOOP**

Iterates over a sequence of values.

for value in sequence:

# code to execute for each value

in sequence

**WHILE LOOP**

Executes a block of code as long as a condition is

true

.

while condition:

# code to execute while

condition is true

**FUNCTIONS**

**FUNCTION DEFINITION**

Blocks of code that perform a specific task. A

function in Python is defined using the

def

keyword

followed by the function name and a set of

parentheses. Any input parameters or arguments

should be placed inside the parentheses. The

function body should be indented and can contain

one or more statements. The

return

statement is

used to return a value from the function to the

calling code.

def function\_name(parameters):

# code to execute

return value

**FUNCTION CALL**

result = function\_name(argument1,

argument2, ...)

**OPTIONAL ARGUMENTS**

result = def

function\_name(parameter1,

parameter2=default\_value, ...):

# function body

return value

**VARIABLE-LENGTH ARGUMENTS**

def function\_name(\*args):

# function body

return value

**KEYWORD ARGUMENTS**

def function\_name(parameter1,

parameter2, ..., keyword1=value1,

keyword2=value2, ...):

# function body

return value

**DEFAULT ARGUMENT VALUES**

def function\_name(parameter1,

parameter2=default\_value):

# function body

return value

**ANONYMOUS FUNCTIONS (LAMBDA**

**FUNCTIONS)**

lambda arguments: expression

**VARIABLE SCOPE**

# Global variable

variable\_name = value

def function\_name():

# Local variable

variable\_name = value

**LISTS**

Ordered collections of items.

my\_list = [item1, item2, item3]

**LIST METHODS**

my\_list.append(item) # adds an item

to the end of the list

my\_list.insert(index, item) #

inserts an item at a specific index

my\_list.pop() # removes and returns

the last item in the list

my\_list.remove(item) # removes the

first occurrence of an item

**DICTIONARIES**

Unordered collections of key-value pairs.

my\_dict = {"key1": value1, "key2":

value2, "key3": value3}

**DICTIONARY METHODS**

my\_dict.keys() # returns a list of

keys

my\_dict.values() # returns a list

of values

my\_dict.items() # returns a list of

key-value pairs

**TUPLES**

Ordered collections of items that cannot be changed

).

immutable

(

my\_tuple = (item1, item2, item3)

**SETS**

Unordered collections of unique items.

my\_set = {item1, item2, item3}

**SET METHODS**

my\_set.add(item) # adds an item to

the set

my\_set.remove(item) # removes an

item from the set

my\_set.union(other\_set) # returns a

new set with all unique items from

both sets

my\_set.intersection(other\_set) #

returns a new set with items that

are common to both sets

**INPUT AND OUTPUT**

**INPUT**

Allows a user to enter data into a program.

input\_string = input("Enter a value:

")

**OUTPUT**

Displays data to a user.

print("Hello, world!")

**FILE HANDLING**

**OPENING A FILE**

file = open("filename.txt", "r") #

open file for reading

file = open("filename.txt", "w") #

open file for writing

**READING FROM A FILE**

file\_contents = file.read() # reads

entire file

file\_contents = file.readline() #

reads one line of file

**WRITING TO A FILE**

file.write("Hello, world!") #

writes string to file

**CLOSING A FILE**

file.close() # closes file

**EXCEPTION HANDLING**

Errors that occur during program execution.

try:

# code that might raise an

exception

except ExceptionType:

# code to execute if exception

occurs

finally:

# code to execute regardless of

whether an exception occurred

**CLASSES AND OBJECTS**

**CLASSES**

In Python, a class is a blueprint for creating objects.

It defines a set of attributes and methods that the

objects will have. When you create an instance of a

class, you create a new object that has the same

attributes and methods as the class.

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def say\_hello(self):

print("Hello, my name is",

self.name, "and I'm", self.age,

"years old.")

**OBJECTS**

Instances of a class with specific values for their

attributes. To create an instance of the Person class,

we use the constructor method

*`init`*

which

initializes the object’s attributes. We then call the

say\_hello

method on each object to print the

greeting message.

person1 = Person("Alice", 25)

person2 = Person("Bob", 30)

person1.say\_hello()

person2.say\_hello()

**@CLASSMETHOD DECORATOR**

In Python, the

@classmethod

decorator is used to

define class methods. A class method is a method

that is bound to the class and not the instance of the

class. It can be called on the class itself, rather than

on an instance of the class.

class MyClass:

class\_var = "This is a class

variable"

def \_\_init\_\_(self, x):

self.x = x

@classmethod

def class\_method(cls):

print(cls.class\_var)

MyClass.class\_method()

To define a class method, we use the

@classmethod

decorator before the method definition. The first

parameter of a class method is always

cls

, which

refers to the class itself. You can use the

cls

parameter to access class variables and methods.

**INHERITANCE**

Classes can also inherit attributes and methods

from other classes.

class Student(Person):

def \_\_init\_\_(self, name, age,

grade):

super().\_\_init\_\_(name, age)

self.grade = grade

def say\_hello(self):

print(f"Hello, my name is

{

self.name}, I am {self.age} years

old, and I am in grade {self.grade}.")

#### MAGIC METHODS

Python also supports a number of special methods, called "magic methods" or "dunder methods" (short for "double underscore methods"), that allow you to customize the behavior of objects of a class. These methods have names that start and end with double underscores

* init(self[, …]) Initializes a new instance of the class
* str(self) Defines the string representation of an object
* repr(self) Defines the string representation of an object that can be used to recreate the object
* eq(self, other) Defines how two objects are compared for equality using the == operator
* lt(self, other) Defines how two objects are compared for less-than using the < operator
* len(self) Defines the behavior of the len() function for an object
* getitem(self, key) Defines how an object is accessed using square brackets, e.g. my\_object[key]
* setitem(self, key, value) Defines how an object is modified using square brackets, e.g. my\_object[key] = value
* delitem(self, key) Defines how an object is deleted using the del keyword and square brackets, e.g. del my\_object[key]
* getattr(self, name) Defines how an attribute that doesn’t exist on the object is accessed, e.g. my\_object.foo
* setattr(self, name, value) Defines how an attribute is set on the object, e.g. my\_object.foo

= 42

* delattr(self, name) Defines how an attribute is deleted from the object, e.g. del my\_object.foo
* call(self[, …]) Allows an object to be called like a function, e.g. my\_object()

#### COMMENTS AND DOCSTRINGS

Comments and docstrings are two important ways to document your code in Python. While comments are used to provide explanations for specific lines or blocks of code, docstrings are used to provide documentation for functions, classes, and modules.

##### Comments

Comments in Python start with the # symbol and can be used to provide explanations for specific lines of code:

# This is a comment x = 1 # This is another comment

In this example, we use comments to explain what the code does. Comments are ignored by the Python interpreter and are not executed as code.

##### Docstrings

Docstrings, on the other hand, are used to document functions, classes, and modules. They are enclosed in triple quotes """, and should describe what the function does, what arguments it takes, and what it returns:

def add\_numbers(a, b):

"""

This function adds two numbers and returns the result.

Parameters:

a (int): The first number to add. b (int): The second number to add.

Returns:

int: The sum of a and b.

""" return a + b

In this example, we define a function called add\_numbers and provide a docstring that describes what the function does, what arguments it takes, and what it returns. The docstring should be placed immediately after the function definition.

Docstrings are important because they make it

easier for other programmers (and your future self)

to understand what your code does and how to use

it. They can be accessed using the

help()

function or

by using the built-in

doc

attribute.

**MODULES AND PACKAGES**

**MODULES**

Files containing Python code that can be imported

and used in other programs.

import my\_module

my\_module.my\_function() # calls a

function from the module

**PACKAGES**

Collections of related modules that can be imported

together.

import my\_package.my\_module

my\_package.my\_module.my\_function()

# calls a function from the module

in the package

**LAMBDA FUNCTIONS**

Anonymous functions that can be defined in a

single line of code.

my\_lambda = lambda x: x\*\*2 #

defines a lambda function that

squares its input

result = my\_lambda(3) # calls the

lambda function with input 3

**LIST COMPREHENSIONS**

Compact syntax for creating lists based on other

lists or sequences.

my\_list = [x\*\*2 for x in range(5)]

# creates a list of squares of

numbers 0-4

even\_numbers = [x for x in my\_list

if x % 2 == 0] # creates a list of

even numbers from my\_list

**GENERATORS**

Functions that use the

yield

keyword to return

values one at a time, instead of all at once.

def my\_generator():

yield 1

yield 2

yield 3

for value in my\_generator():

# code to execute for each value

returned by the generator

**DECORATORS**

Functions that modify the behavior of other

functions.

def my\_decorator(func):

def wrapper(\*args, \*\*kwargs):

# code to execute before the

original function

result = func(\*args,

\*\*kwargs) # call the original

function

# code to execute after the

original function

return result

return wrapper

@my\_decorator

def my\_function():

# code to execute

### MAP, FILTER, AND REDUCE

#### MAP

Applies a function to every element in a sequence and returns a new sequence with the results.

def square(x): return x\*\*2

my\_list = [1, 2, 3, 4] squared\_list = map(square, my\_list) # creates a new list with squares of the original values

#### FILTER

Applies a function to every element in a sequence and returns a new sequence with only the elements that pass a certain test.

def is\_even(x): return x % 2 == 0

my\_list = [1, 2, 3, 4] even\_list = filter(is\_even, my\_list) # creates a new list with only the even values from the original list

#### REDUCE

Applies a function to pairs of elements in a sequence and returns a single result.

from functools import reduce

def my\_function(x, y): return x + y

my\_list = [1, 2, 3, 4] result = reduce(my\_function, my\_list) # adds up all the values in the list to get a single result

### STRING FORMATTING

Allows values to be inserted into a string in a specific format.

name = "Alice" age = 30 greeting = "Hello, my name is {} and I am {} years old.".format(name, age) # creates a string with values inserted using curly braces

### REGULAR EXPRESSIONS

Regular expressions, also known as regex or regexp, are a sequence of characters that define a search pattern. They are powerful tools used to perform text manipulation and data extraction in programming languages such as Python. In Python, the re module provides support for regular expressions.

#### FUNCTIONS

* re.search(pattern, string): Searches for the first occurrence of the pattern in the string and returns a match object if found. If not found, it returns None.
* re.findall(pattern, string): Searches for all occurrences of the pattern in the string and returns a list of all matches found.
* re.sub(pattern, repl, string): Searches for all occurrences of the pattern in the string and replaces them with the repl string.

#### METACHARACTERS

* . (dot): Matches any character except a newline character.
* ^ (caret): Matches the start of a string.
* $ (dollar): Matches the end of a string.
* \* (asterisk): Matches zero or more occurrences of the preceding character.
* + (plus): Matches one or more occurrences of the preceding character.
* ? (question mark): Matches zero or one occurrence of the preceding character.

•

[]

(square brackets): Matches any one of the

characters enclosed in the brackets.

•

|

(pipe): Matches either the expression before

or after the pipe.

import re

# Search for a pattern in a string

text = "The quick brown fox jumps

over the lazy dog"

match = re.search(r"brown", text)

if match:

print("Match found!")

# Find all occurrences of a pattern

in a string

text = "The quick brown fox jumps

over the lazy dog"

matches = re.findall(r"the", text,

re.IGNORECASE)

print(matches)

# Replace all occurrences of a

pattern in a string

text = "The quick brown fox jumps

over the lazy dog"

new\_text = re.sub(r"the", "that",

text, flags=re.IGNORECASE)

print(new\_text)



**JCG deliver**

**s**

**over 1 million pages each month to more than 700K software**

**developers, architects and decision**

**makers. JCG offers something for everyone,**

**including news, tutorials, cheat sheets, research guides, feature articles, source code**

**and more.**

Copyright © 201

4

Exelixis Media P.C. All rights reserved. No part of this publication may be

reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic,

mechanical, photocopying, or otherwise, without prior written permission of the publisher.

**CHEATSHEET FEEDBACK**

**WELCOME**

support@javacodegeeks.com



**SPONSORSHIP**

**OPPORTUNITIES**

sales@javacodegeeks.com