**<https://www.canva.com/design/DAGMge1Puzc/NAMY2SkA4eE8bydrwWRiFw/watch?utm_content=DAGMge1Puzc&utm_campaign=share_your_design&utm_medium=link&utm_source=shareyourdesignpanel>**

**Функции**

**1. Синтаксис:**

def имя\_функции():

действие\_функции

**2. Пример:**

def greet():

print("Привет, мир!")

greet()

**3. Локальные переменные**

def show\_number():

num = 100 # Локальная переменная

print(num)

show\_number()

print(num) # Ошибка, так как num не определена вне функции.

**4. Docstrings**

def greet():

"""

Эта функция выводит приветственное сообщение.

"""

print("Привет, мир!")

**5. Аргументы и возвращаемое значение**

def greet(name):

print(f"Привет, {name}!")

greet("Алиса")

**6. Пример с несколькими аргументами:**

def full\_name(first\_name, last\_name):

print(f"Ваше полное имя: {first\_name} {last\_name}")

full\_name("Алиса", "Селезнева")

**7. Возвращаемое значение**

def add(a, b):

return a + b

result = add(10, 5)

print(result)

**8. Необязательные аргументы и значения по умолчанию**

def greet(name="Гость"):

print(f"Привет, {name}!")

greet()

greet("Боб")

**9. Переменное количество аргументов:**

\*args позволяет передать любое количество позиционных аргументов.

def print\_all(\*args):

for arg in args:

print(arg)

print\_all(1, 2, 3, "a", "b")

**10. Передача именованных аргументов:**

def print\_data(\*\*kwargs):

for key, value in kwargs.items():

print(f"{key}: {value}")

print\_data(name="Анна", age=30, job="инженер")

\*\*kwargs позволяет передать любое количество именованных аргументов.

**11. Lambda-функции:**

multiply = lambda x, y: x \* y

print(multiply(3, 4))

**12. Замыкания (Closures)**

Замыкание – это функция, которая "запоминает" окружение, в котором была создана, даже если это окружение больше не доступно. Основное, что нужно понять о замыканиях: это функции, которые имеют доступ к переменным из объемлющей функции, где они были определены.

**12.1 Базовое понимание**

def outer\_function(x):

def inner\_function(y):

return x + y

return inner\_function

closure = outer\_function(10)

print(closure(5)) # Выведет: 15

В этом примере, inner\_function – это замыкание, которое "запоминает" значение x из outer\_function.

**12.2 Для чего нужны замыкания?**

1. Упаковка данных и функций в одну сущность
2. Используется для создания декораторов
3. Отложенное выполнение функций

**12.3 Замыкания в декораторах**

Замыкания часто используются для создания декораторов:

def decorator\_with\_arguments(arg):

def actual\_decorator(func):

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

print(f"Декоратор аргумент: {arg}")

return func(\*args, \*\*kwargs)

return wrapper

return actual\_decorator

@decorator\_with\_arguments("Пример")

def say\_hello():

print("Привет!")

say\_hello()

**12.4 Замыкания для инкапсуляции данных**

Замыкания можно использовать для скрытия или инкапсуляции данных:

def make\_counter():

count = 0

def counter():

nonlocal count

count += 1

return count

return counter

counter = make\_counter()

print(counter()) # Выведет: 1

print(counter()) # Выведет: 2

В заключение: замыкания – это мощный инструмент в Python, который позволяет нам сохранять состояние и данные локально в функции. Они играют ключевую роль в многих продвинутых темах, таких как декораторы, и дают большую гибкость при проектировании кода.

**13. Декораторы**

Декораторы в Python – это мощный и гибкий способ изменения или расширения функциональности функций или методов без изменения их кода. Декораторы представляют собой ещё один слой абстракции, который позволяет добавлять поведение или действия перед или после основного выполнения функции.

**13.1 Простейший декоратор:**

def simple\_decorator(func):

def wrapper():

print("Действие до вызова функции")

func()

print("Действие после вызова функции")

return wrapper

@simple\_decorator

def say\_hello():

print("Привет!")

say\_hello()

Когда вы вызываете say\_hello(), на самом деле вызывается wrapper(), который в свою очередь вызывает say\_hello().

**13.2 Декораторы с аргументами:**

Если декорируемая функция принимает аргументы, необходимо использовать \*args и \*\*kwargs:

def decorator\_with\_args(func):

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

print("Действие до вызова функции")

func(\*args, \*\*kwargs)

print("Действие после вызова функции")

return wrapper

@decorator\_with\_args

def greet(name):

print(f"Привет, {name}!")

greet("Анна")

**13.3 Декораторы с параметрами:**

def repeat(num):

def actual\_decorator(func):

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

for \_ in range(num):

func(\*args, \*\*kwargs)

return wrapper

return actual\_decorator

@repeat(num=3)

def say\_hello():

print("Привет!")

say\_hello()

**13.4 Несколько декораторов на одну функцию:**

Функции можно применять несколько декораторов:

@decorator1

@decorator2

def some\_function():

pass

Это эквивалентно:

some\_function = decorator1(decorator2(some\_function))

**13.5 Сохранение метаданных при декорировании:**

При использовании декораторов метаданные (например, имя функции) могут быть потеряны. Для их сохранения можно использовать модуль functools.wraps:

from functools import wraps

def simple\_decorator(func):

@wraps(func)

def wrapper():

print("Действие до вызова функции")

func()

print("Действие после вызова функции")

return wrapper

**13.6 Заключение:**

Декораторы в Python – это мощный инструмент для изменения и расширения функциональности функций и классов без необходимости изменять их исходный код. Они предлагают чистый и элегантный способ добавления поведения к объектам и улучшения структуры вашего кода.

**14. Рекурсия**

Рекурсия – это метод, при котором функция вызывает саму себя. Рекурсивные функции могут быть мощным инструментом, но они также могут быть сложными и требуют осторожного использования.

**14.1 Простейший рекурсивный пример: вычисление факториала**

def factorial(n):

if n <= 1:

return 1

else:

return n \* factorial(n-1)

print(factorial(5)) # Выведет: 120

Здесь функция factorial вызывает саму себя для вычисления факториала числа.

Базовый случай: Важно определить базовый случай в рекурсии, чтобы предотвратить бесконечные вызовы. В примере выше базовым случаем является n <= 1.

**14.2 Продвинутая рекурсия:**

Рекурсия с несколькими базовыми случаями

Например, числа Фибоначчи:

def fibonacci(n):

if n <= 1:

return n

else:

return fibonacci(n-1) + fibonacci(n-2)

**14.3 Мемоизация**

Для оптимизации рекурсивных функций, которые многократно вычисляют одни и те же значения, можно использовать мемоизацию:

def fibonacci\_memo(n, memo={}):

if n in memo:

return memo[n]

if n <= 1:

return n

else:

value = fibonacci\_memo(n-1, memo) + fibonacci\_memo(n-2, memo)

memo[n] = value

return value

**14.4 Глубина рекурсии**

Python имеет ограничение на максимальную глубину рекурсии (обычно 1000 вызовов). Для изменения этого значения можно использовать модуль sys:

import sys

sys.setrecursionlimit(2000)

Однако будьте осторожны, увеличение глубины рекурсии может привести к зависанию программы или переполнению стека.

**14.5 Заключение:**

Рекурсия – это мощный инструмент, который позволяет элегантно решать многие задачи, особенно связанные со структурами данных, такими как деревья. Однако рекурсия требует понимания и осторожного использования, чтобы избежать потенциальных проблем, таких как бесконечные циклы или исчерпание стека вызовов.

**15. Функции высшего порядка**

Функции высшего порядка – это функции, которые принимают одну или несколько функций в качестве аргументов и/или возвращают функцию в качестве результата. В Python, благодаря его возможностям по работе с функциями как с объектами первого класса, функции высшего порядка легко реализовать и использовать.

**15.1 Простой пример функции высшего порядка:**

def apply(func, value):

return func(value)

def double(x):

return x \* 2

result = apply(double, 5) # Результат: 10

**15.2 Встроенные функции высшего порядка: map, filter, reduce**

# map применяет функцию ко всем элементам списка

numbers = [1, 2, 3, 4]

doubled = list(map(double, numbers)) # Результат: [2, 4, 6, 8]

# filter возвращает элементы, для которых функция возвращает True

def is\_even(x):

return x % 2 == 0

evens = list(filter(is\_even, numbers)) # Результат: [2, 4]

# reduce применяет функцию последовательно к элементам списка, сворачивая их в одно значение

from functools import reduce

def add(x, y):

return x + y

sum = reduce(add, numbers) # Результат: 10

**15.3 Функции, возвращающие функции:**

def power(exponent):

def inner(base):

return base \*\* exponent

return inner

square = power(2)

cube = power(3)

print(square(4)) # Результат: 16

print(cube(2)) # Результат: 8

**15.4** Функции высшего порядка предоставляют мощные абстракции для создания гибких и выразительных программ на Python. Они позволяют управлять поведением функций, комбинировать их и расширять их функциональность без изменения исходного кода. Это основа многих паттернов проектирования и техник программирования в функциональном стиле.

**16. Подробное изучение аргументов:**

**16.1 Для изменяемых объектов (например, списков, словарей):**

def modify\_list(lst):

lst.append(4)

my\_list = [1, 2, 3]

modify\_list(my\_list)

print(my\_list) # Вывод: [1, 2, 3, 4]

16.1 Использование \* и \*\* при вызове функции

\* – распаковка итерируемых объектов

def print\_numbers(a, b, c):

print(a, b, c)

args = [1, 2, 3]

print\_numbers(\*args) # Вывод: 1 2 3

\*\* – распаковка словарей

def print\_names(first, middle, last):

print(first, middle, last)

names = {"first": "John", "middle": "W.", "last": "Doe"}

print\_names(\*\*names) # Вывод: John W. Doe

**17. Обработка исключений внутри функций**: Как обрабатывать ошибки и исключения внутри функций с помощью try, except, finally.

**17.1 Основы обработки исключений**

В Python исключения могут быть вызваны различными ошибками или нештатными ситуациями в коде. Обработка исключений позволяет управлять этими ошибками и продолжать выполнение программы.

Базовый пример try и except:

def divide(x, y):

try:

result = x / y

except ZeroDivisionError:

print("Ошибка: деление на ноль!")

result = None

return result

Если функция вызывается с y = 0, то будет поймано исключение ZeroDivisionError и функция вернёт None.

**17.2 Обработка нескольких исключений**

Можно обрабатывать разные типы исключений, указывая несколько блоков except.

def fetch\_data(dictionary, key):

try:

value = dictionary[key]

except KeyError:

print(f"Ошибка: ключ '{key}' не найден!")

value = None

except TypeError:

print("Ошибка: аргумент должен быть словарём!")

value = None

return value

**17.3 Использование else и finally**

else: Этот блок будет выполнен, если исключения в блоке try не были пойманы.

finally: Этот блок будет выполнен в любом случае, независимо от того, были ли пойманы исключения.

def open\_file(filename):

try:

file = open(filename, "r")

content = file.read()

except FileNotFoundError:

print("Ошибка: файл не найден!")

content = None

else:

print("Файл успешно прочитан!")

finally:

if file:

file.close()

print("Файл закрыт!")

return content

**17.4 Вызов исключений с помощью raise**

Можно вызывать собственные исключения с помощью ключевого слова raise.

def set\_age(age):

if age < 0:

raise ValueError("Возраст не может быть отрицательным!")

return age

**17.5 Заключение:**

Обработка исключений – ключевой аспект программирования на Python. Это позволяет создавать устойчивые к ошибкам приложения и обрабатывать неожиданные ситуации на лету. Использование try, except, else и finally позволяет обеспечивать гибкое управление ошибками и обеспечивать корректное завершение функций даже при возникновении проблем.