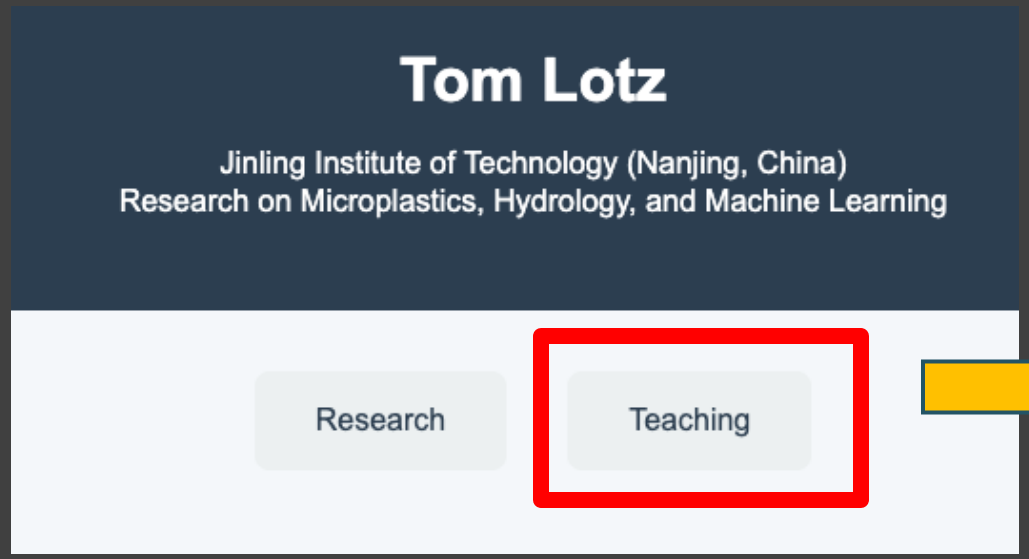
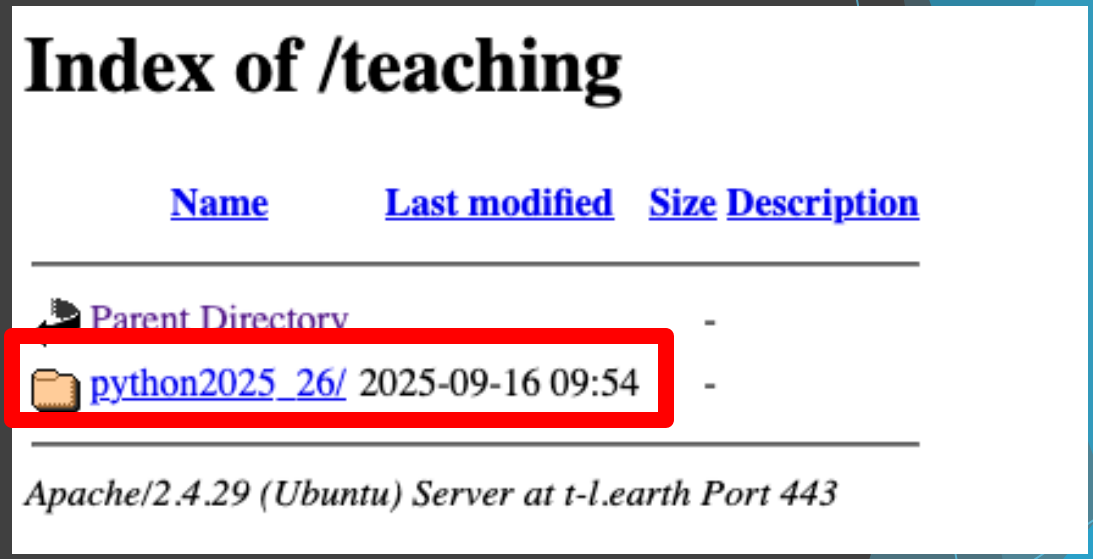
Download the cheat sheets and slides from here

t-l.earth





Python语言程序设计

Python Programming

2025/26

Session 06

Tom Lotz (tom.lotz@outlook.com)

Content

Brain Activation

Loop Through a List

- Make a list of three cities, then print each one.

```
cities = ["Nanjing", "Paris", "Tokyo"]
for city in cities:
    print(city)
```

Positive or Negative?

- Use an if statement to test each number in a list to see if it is positive or negative and print the result.

```
numbers = [3, -1, 7]
for n in numbers:
    if n > 0:
        print(n, "is positive")
    else:
        print(n, "is negative")
```

Mini Dictionary Recap

- Create a small dictionary describing one student.

```
student = {"name": "Ali", "age": 20}
print(f"My name is {student['name']} and I am {student['age']} years old.")
```

Review

Why Dictionaries?

- Lists: store items by position.

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0]) # 'apple'
```

- Dictionaries: store items by name (key).

```
person = {"name": "Ali", "age": 20}
print(person["name"]) # 'Ali'
```

Dictionary Syntax

- A dictionary is wrapped in curly braces { }. Each key is connected to a value with a colon :. Multiple pairs are separated by commas.

```
dict = {"key 1": "value 1", "key 2": "value 2"}
```

- Keys: must be unique.
- Values: can be numbers, strings, lists, or even other dictionaries.

Dictionary Syntax

- Values are accessed by their key.

```
person = {"name": "Ali", "age": 20}
print(person["name"]) # 'Ali'
```

- There is no index!

Adding and Modifying Data

- Dictionaries are dynamic – you can add or change data anytime.

```
student = {"name": "Ali", "age": 20}
student["major"] = "Software Engineering" # add
student["age"] = 21 # modify
print(student) # {'name': 'Ali', 'age': 21, 'major': 'Software Engineering'}
```

Deleting Key-Value Pairs

- Use del to permanently remove a pair.

```
alien = {"color": "green", "points": 5}
del alien["points"]
print(alien) # {'color': 'green'}
```

Looping through key-value pairs

- Use .items() to get both the key and its value:

```
student = {"name": "Ali", "age": 21, "major": "Software Engineering"}
for key, value in student.items():
    print(key, value)
```

```
name Ali
age 21
major Software Engineering
```

Looping through keys only

- If you only need the keys, use .keys():

```
student = {"name": "Ali", "age": 21, "major": "Software Engineering"}
for key in student.keys():
    print(key)
```

```
name
age
major
```

Looping through values only

- Use .values() to get all values:

```
student = {"name": "Ali", "age": 21, "major": "Software Engineering"}
for value in student.values():
    print(value)
```

```
Ali
21
Software Engineering
```

Avoiding Repeated Values with set()

- If some values repeat, we can make them unique using set():

```
favorite_languages = {
'jen': 'python',
'sarah': 'c',
'edward': 'rust',
'phil': 'python'
}

for language in set(favorite_languages.values()):
    print(language.title())
```

```
Python
Rust
C
```

Defining and Calling Functions

What Is a Function?

- A function is a named block of code that performs one task.
- We use it to:
  - Reuse code instead of writing the same thing many times.
  - Keep programs organized and readable.
- Example idea: a machine that does one job when you press its button.

```
machine()
```

What Is a Function?

- We have already used some functions in the past:
  - len()
  - sorted()
  - range()
- These are built-in functions. But we can also make our own functions!

Defining a Function

- Basic syntax:

```
def greet_user():
    print("Hello!")
```

- def introduces a function.
- The name (greet_user) describes its job.
- The colon starts an indented block (function body).
- To run it: you must call it. Before that, nothing happens.

```
greet_user()
```

Mini Task 1

- Create and call your own simple function.
- Example:

```
def greet_user():
    print("Hello!")
```

```
greet_user()
```

Adding a Parameter

- Functions can accept input values.

```
def greet_user(name):
    print("Hello, ", name)


greet_user("Lina")
greet_user("Maxi")
```

Arguments vs Parameters

- Parameter: variable name inside the function definition.
- Argument: actual value passed when calling the function.

- Example:

```
def greet_user(name):  #name = Parameter
    print("Hello, ", name)

greet_user("Lina")     # "Lina" = Argument
```

Mini Task 2

- Adjust your own function definition with a parameter and call it with an argument.
- Example:

```
def greet_user(name):  #name = Parameter
    print("Hello, ", name)

greet_user("Lina")     # "Lina" = Argument
```

Functions with Multiple Parameters

- You can pass more than one piece of information. Functions can be designed with any number of parameters.

```
def describe_pet(animal, name):
    print("I have a ", animal, " named ",name)


describe_pet("dog", "Max")
```

Order matters!

- The arguments must be provided in the same order as the function definition expects them.

```
describe_pet("dog", "Max")
describe_pet("Max", "dog")
```

```
I have a  dog  named  Max
I have a  Max  named  dog
```

Mini Task 3

- Add one more parameter to your own function and call it with two arguments.

Wrap-up

- Why Use Functions?
  - Avoid repetition – write once, use many times.
  - Improve structure – break big programs into small tasks.
  - Make code readable – each function has a clear name and job.
- Define a function using def and a name.
- Use parentheses () for parameters.
- Call functions to make them run.

Passing Information to Functions

Positional Arguments

- The arguments we have used so far are called positional arguments.

```
def describe_pet(animal, name):
    print("I have a ", animal, " named ",name)


describe_pet("dog", "Max")
```

Positional Arguments

- The arguments we have used so far are called positional arguments.

```
def describe_pet(animal, name):
    print("I have a ", animal, " named ",name)



describe_pet("dog", "Max")
```

- The order is absolute!

Keyword Arguments

- The alternative is to use keywords (argument names) when calling the function.

```
def describe_pet(animal, name):
    print("I have a ", animal, " named ",name)

describe_pet(animal="dog", name="Max")
describe_pet(name="Max", animal="dog")
```

```
I have a  dog  named  Max
I have a  dog  named  Max
```

Mini Task 4

- Call your function by using keyword arguments.

Default Values

- You can provide default values to parameters.
  - In the definition, default parameters come after required parameters.

```
def describe_pet(name, animal = "dog" ):
    print("I have a ", animal, " named ",name)

describe_pet(name="Max", animal="dog")
describe_pet(name="Max")
describe_pet(name="Max", animal="cat")
```

```
I have a  dog  named  Max
I have a  dog  named  Max
I have a  cat  named  Max
```

Mini Task 5

- Add a default parameter to your function and call it.
- Example:

```
def describe_pet(name, animal = "dog" ):
    print("I have a ", animal, " named ",name)

describe_pet(name="Max", animal="dog")
describe_pet(name="Max")
describe_pet(name="Max", animal="cat")
```

Avoiding argument errors

- Arguments without default values need to be added during a function call, otherwise we receive an argument error.

```
def greet_user(name):
    print("Hello, ", name)


greet_user() # -> Argument Error
```

```
TypeError: greet_user() missing 1 required positional argument: 'name'
```

Optional Values

- We can use an empty string “” or None to make an argument optional.

```
def print_full_name(first, last, middle=""):
    if middle:
        print(first, middle, ",", last)
    else:
        print(first, ",", last)

print_full_name("John", "Doe") # John , Doe
print_full_name("John", "Doe", "Lee") # John Lee , Doe
```

Optional Values

- We can use an empty string “” or None to make an argument optional.

```
def print_full_name(first, last, middle=None):
    if middle:
        print(first, middle, ",", last)
    else:
        print(first, ",", last)

print_full_name("John", "Doe") # John , Doe
print_full_name("John", "Doe", "Lee") # John Lee , Doe
```

Wrap-up

- Functions can take multiple inputs in flexible ways.
- Use positional, keyword, and default arguments effectively.
- Add optional parameters for versatility.

Returning Values

Returning values

- So far, we have always printed the result of our functions using a direct print().

```
def add(a, b):
    print(a + b)

add(1, 2)
```

Returning values

- Usually, the result of a function call is returned as a value.
- With return, the result can be stored or reused.

```
def add(a, b):
    return a + b
```

Capturing Return Values

- We can capture the return value of a function with an assignment.

```
def add(a, b):
    return a + b

c = add(1, 2)

print(c)
```

Returning text

- We can write a function to repeat common tasks

```
def get_full_name(first, last):
    full = first + " " + last
    return full.title()


musician = get_full_name("jimi", "hendrix")
print(musician) # Jimi Hendrix
```

Mini Task 6

- Write your own function that accepts arguments and returns a value instead of printing it. You can use text or number arguments, as you wish.

- Example:

```
def get_full_name(first, last):
    full = first + " " + last
    return full.title()


musician = get_full_name("jimi", "hendrix")
print(musician) # Jimi Hendrix
```

Optional Arguments with Return

- Use default or optional parameters in return functions.

```
def return_full_name(first, last, middle=None):
    if middle:
        full_name = first + " " + middle + ", " + last
    else:
        full_name = first + ", " + last

    return full_name

full_name = return_full_name("John", "Doe")
print(full_name) # John, Doe
full_name = return_full_name("John", "Doe", "Lee")
print(full_name) # John Lee, Doe
```

Returning a Dictionary

- Functions can build and return structured data.

```
def build_person(first, last, age):
    person = {"first": first, "last": last}
    person["age"] = age
    return person

musician = build_person("jimi", "hendrix", 27)
print(musician) # {'first': 'jimi', 'last': 'hendrix', 'age': 27}
print(musician["first"]) # jimi
```

Mini Task 7

- Adjust your own function to return a dictionary instead of a single value.
- Remember:

```
dict1 = {}
dict1["val1"] = 1
dict1["val2"] = "text"
```

Returning from Conditional Logic

- Functions can decide what value to return based on conditions.

```
def pos_or_neg(number):
    if number < 0:
        return "Negative"
    elif number > 0:
        return "Positive"

r = pos_or_neg(3)
print(r) # Positive
```

Using Returned Values in Loops

- What do you think result will be?

```
def square(n):
    return n ** 2


numbers = [1, 2, 3, 4]
results = []
for n in numbers:
    results.append(square(n))


print(results)
```

Using Returned Values in Loops

- What do you think result will be?

```
def square(n):
    return n ** 2

numbers = [1, 2, 3, 4]
results = []
for n in numbers:
    results.append(square(n))

print(results)
```

```
[1, 4, 9, 16]
```

Wrap-up

- Use return to send results back to the caller.
- Store and reuse those results later.
- Use print() in the function only to inform the user about a result.

Passing Lists and Using *args / kwargs

Moving Data Between Lists

- We can also pass a list as an argument.

```
def move(src):
    dst = []
    while src:
        dst.append(src.pop())
    return dst


old = ['a', 'b']
new = move(old[:]) # use copy !!!
print(new) # ['b', 'a']
print(old) # ['a', 'b'] stays unchanged
```

Moving Data Between Lists

- We can (of course) also pass a list as an argument.

```
def move(src):
    dst = []
    while src:
        dst.append(src.pop())
    return dst


old = ['a', 'b']
new = move(old[:]) # use copy !!!
print(new) # ['b', 'a']
print(old) # ['a', 'b'] stays unchanged
```

Mini Task 8

- Make any function that accepts a list as argument and does something with the list.

Arbitrary Arguments

- If we do not know the number of arguments, we can work with arbitrary arguments.
- The arguments will be passed as a tuple.

```
def pizza(*args):
    print('Pizza with:', args)

pizza('cheese') # Pizza with: ('cheese',)
pizza('mushroom', 'onion') # Pizza with: ('mushroom', 'onion')
```

Tuple ???

- A tuple is similar to a list, but it can’t be changed after creation.
- A tuple uses parentheses instead of brackets.

```
t = ("cheese","mushroom","onion") # immutable
l = ["cheese","mushroom","onion"] # mutable
```

Mixing normal arguments with *args

- We can mix normal arguments with *args by defining normal parameters first before adding *args.

```
def pizza(size, *t):
    print(size, 'inch:', t)

pizza(12, 'cheese', 'olives')
```

Mini Task 9

- Write a function sandwich(*args) that prints all provided toppings.
- Reminder:

```
def pizza(*args):
    print('Pizza with:', args)
```

Arbitrary Keyword Args **kwargs

- Using **kwargs, we can pack arbitrary arguments into a dictionary that is passed to the function.

```
def car(make, model, **x):
    x['make'] = make
    x['model'] = model
    return x

car1 = car('BMW', 'X3', color='red')
print(car1) # {'color': 'red', 'make': 'BMW', 'model': 'X3'}
```

Wrap-up

- We can use lists as arguments and return lists.
- *args packs all arbitrary arguments into a tuple and passes them to the function.
- **kwargs packs all arbitrary arguments into a dictionary (with keywords) and passes them to the function.

Exercises

Pizza making

- Build one pizza-ordering function that grows with each task.
- You can find the task list in the CheatSheet_06.py file on the server.
- Try to finish as many tasks as possible!