Functions

Part 2

Python Functions

- There are two kinds of functions in Python.

  - Built-in functions that are provided as part of Python - print(), input(), type(), float(), int() ...

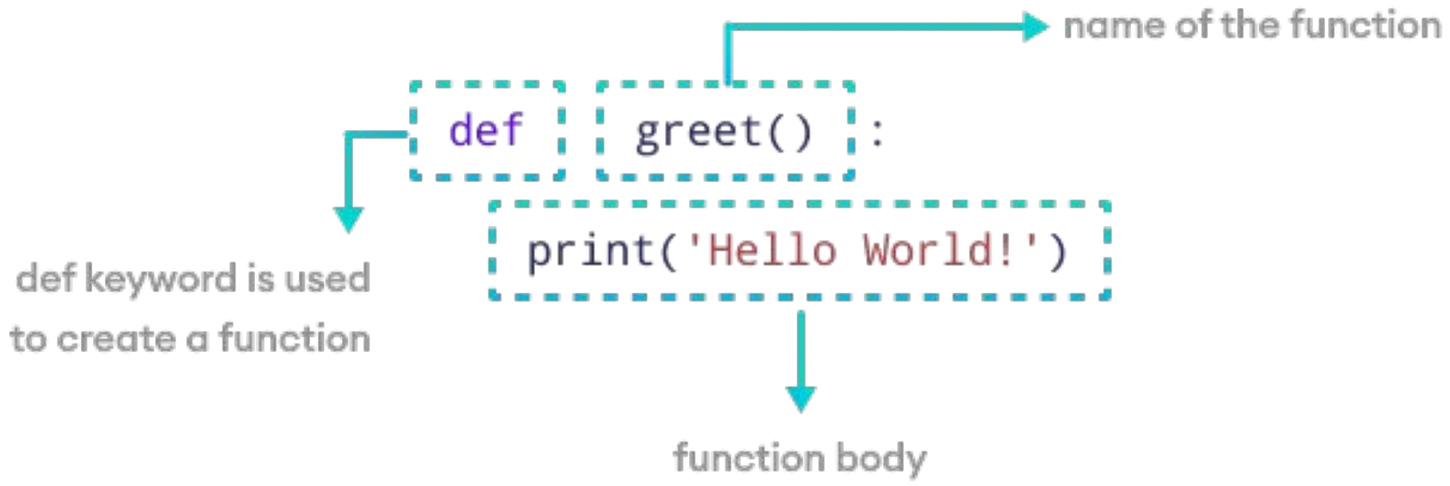
  - Functions that we define ourselves and then use

- We treat function names as “new” reserved words (i.e., we avoid them as variable names)

Function Definition

- In Python a function is some reusable code that takes arguments(s) as input, does some computation, and then returns a result or results
- We define a function using the def reserved word
- We call/invoke the function by using the function name, parentheses, and arguments in an expression

Python Functions



Argument

big = max('Hello world')

Assignment

'w'

Result

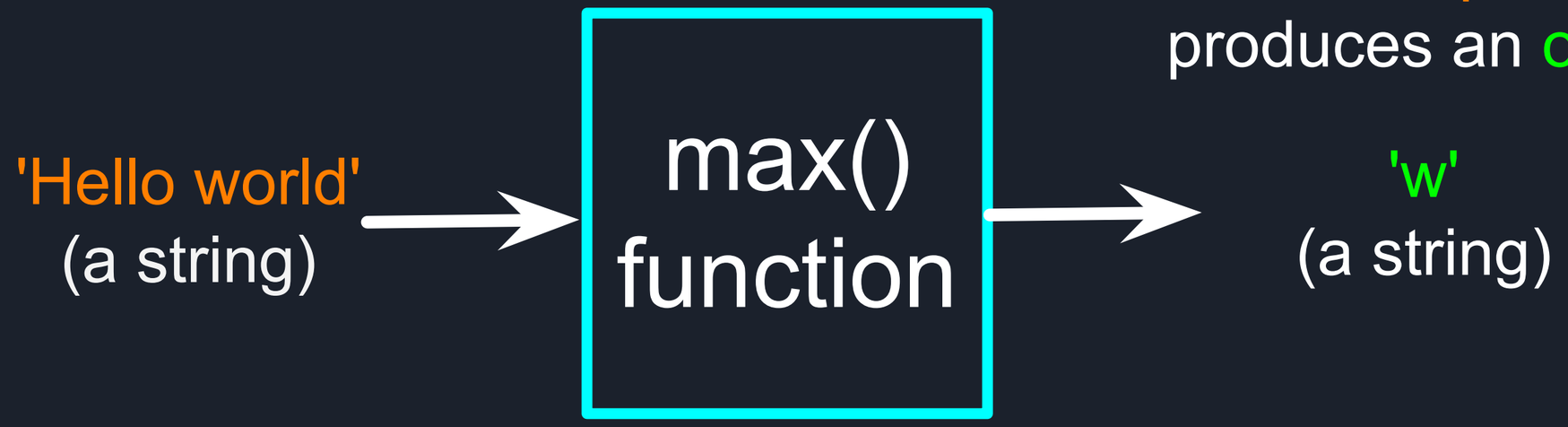
```
>>> big = max('Hello world')
>>> print(big)
w
>>> tiny = min('Hello world')
>>> print(tiny)

>>>
```

Max Function

```
>>> big = max('Hello world')
>>> print(big)
w
```

A function is some stored code that we use. A function takes some input and produces an output.

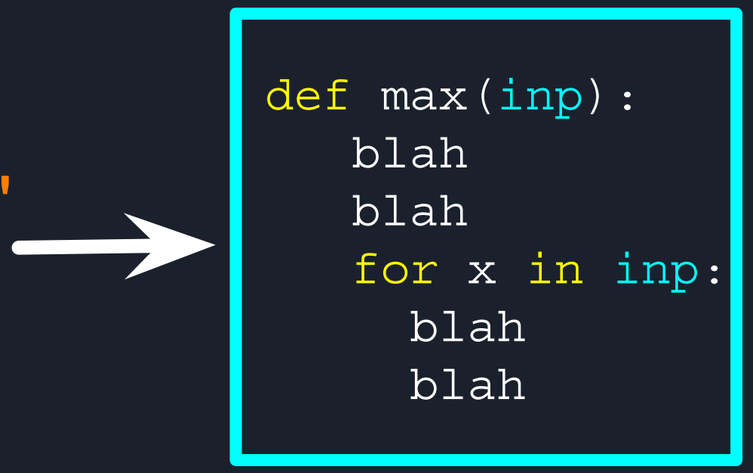


Max Function

```
>>> big = max('Hello world')
>>> print(big)
w
```

A function is some stored code that we use. A function takes some input and produces an output.

'Hello world'
(a string)



'w'
(a string)

Functions of our own…

Building our Own Functions

- We create a new function using the def keyword followed by optional parameters in parentheses
- We indent the body of the function
- This defines the function but does not execute the body of the function

```
def print_lyrics():
    print("I'm okay.")
    print('I sleep all night and I work all day.')
```

print_lyrics():

```
print("I'm okay.")
print('I sleep all night and I work all day.')
```

```
x = 5
print('Hello')

def print_lyrics():
    print("I'm okay.")
    print('I sleep all night and I work all day.')

print('Hi')
x = x + 2
print(x)
```

Output:

Hello
Hi
7

Definitions and Uses

- Once we have defined a function, we can call (or invoke) it as many times as we like
- This is the store and reuse pattern

```
x = 5
print('Hello')

def print_lyrics():
    print("I'm okay.")
    print('I sleep all night and I work all day.')

print('Yo')
print_lyrics()
x = x + 2
print(x)
```

```
Hello
Yo
I'm okay.
I sleep all night and I work all day.
7
```

Arguments

- An argument is a value we pass into the function as its input when we call the function
- We use arguments so we can direct the function to do different kinds of work when we call it at different times
- We put the arguments in parentheses after the name of the function

```
big = max('Hello world')
```

Argument

Parameters

A parameter is a variable which we use in the function definition. It is a “handle” that allows the code in the function to access the arguments for a particular function invocation.

Parameter

```
>>> def greet(lang):
...     if lang == 'es':
...         print('Hola')
...     elif lang == 'fr':
...         print('Bonjour')
...     else:
...         print('Hello')
...
>>> greet('en')
Hello
>>> greet('es')
Hola
>>> greet('fr')
Bonjour
>>>
```

Argument

Return Values

Often a function will take its arguments, do some computation, and return a value to be used as the value of the function call in the calling expression. The return keyword is used for this.

```
def greet():
    return "Hello"

print(greet(), "Glenn")
print(greet(), "Sally")
```

```
Hello Glenn
Hello Sally
```

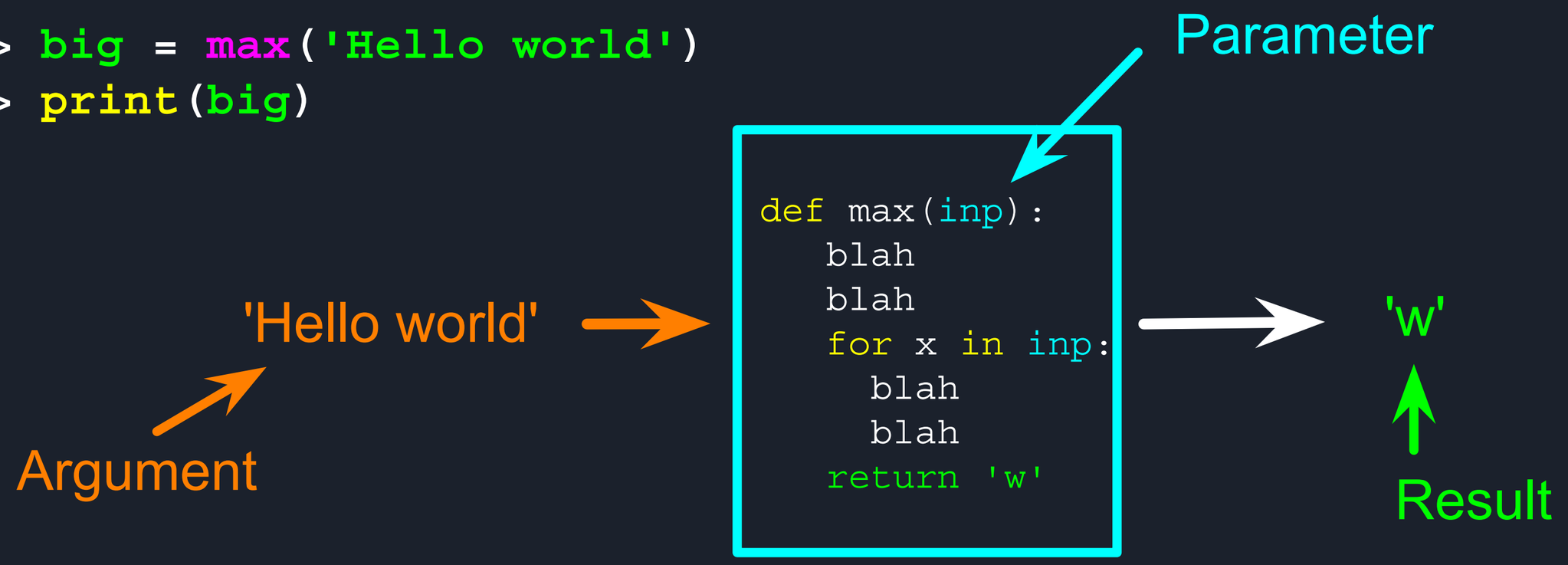
Return Value

- A “fruitful” function is one that produces a result (or return value)
- The return statement ends the function execution and “sends back” the result of the function

```
>>> def greet(lang):
...     if lang == 'es':
...         return 'Hola'
...     elif lang == 'fr':
...         return 'Bonjour'
...     else:
...         return 'Hello'
...
>>> print(greet('en'),'Glenn')
Hello Glenn
>>> print(greet('es'),'Sally')
Hola Sally
>>> print(greet('fr'),'Michael')
Bonjour Michael
>>>
```

Arguments, Parameters, and Results

```
>>> big = max('Hello world')
>>> print(big)
w
```



Multiple Parameters / Arguments

- We can define more than one parameter in the function definition
- We simply add more arguments when we call the function
- We match the number and order of arguments and parameters

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)
```

```
8
```

Void (non-fruitful) Functions

- When a function does not return a value, we call it a “void” function
- Functions that return values are “fruitful” functions
- Void functions are “not fruitful”

Example

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("inside is_even")
    return i%2 == 0
```

run some commands

keyword

expression to evaluate and return

To function or not to function...

- Organize your code into “paragraphs” - capture a complete thought and “name it”
- Don’t repeat yourself - make it work once and then reuse it
- If something gets too long or complex, break it up into logical chunks and put those chunks in functions
- Make a library of common stuff that you do over and over - perhaps share this with your colleagues...

Type Hints

- Annotate function signatures with precise DataFrame input/output types: df: pd.DataFrame -> pd.DataFrame

```python
def average_list(values: List[float]) -> float:
    """
    Compute the average of a list of floats.

    :param values: list of floats
    :return: their average as a float
    """
    return sum(values) / len(values)


def average_series(values: pd.Series) -> float:
    """
    Compute the average of a pandas Series of numbers.

    :param values: pandas Series of numeric values
    :return: their mean as a float
    """
    return values.mean()
```

Error Handling

- Use `try/except` to catch specific or broad exceptions
- Raise meaningful errors: `raise ValueError("...")`, use `raise from` for context

```python
# 1. Specific vs. broad exception catching
def safe_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        print("Cannot divide by zero.")
        return None
    except Exception as e:
        print(f"Error: {e}")
        return None

# 2. Raising meaningful errors and exception chaining
def get_positive_int(value):
    try:
        ivalue = int(value)
    except ValueError as e:
        # Chain original exception for traceback
        raise ValueError(f"Invalid integer input: {value}") from e
    if ivalue <= 0:
        raise ValueError(f"Value must be positive, got {ivalue}")
    return ivalue
```

Error Handling

Types of errors: many

Most likely:

- **NameError** - Occurs when you try to use a variable or function name that hasn’t been defined.
- **TypeError** - Happens when an operation or call is applied to an object of inappropriate type.
- **ValueError** - Raised when a function receives an argument of the right type but an inappropriate value.

Default Parameters

- Functions can supply default values for parameters.
- Supply default values using `param=default` in the signature
- Override defaults via positional or keyword arguments
- Use None (or sentinel) for mutable defaults to avoid shared-state bug

However: default values may lead to confusing function behavior.

```python
def clean_col(df, col, strip=True, lower=True, repl=None, inplace=False):
    """
    Clean a DataFrame text column.

    Parameters:
    df : pd.DataFrame
    col : Name of the column to process.
    strip : Remove leading/trailing whitespace if True.
    lower : Convert text to lowercase if True.
    repl : Mapping for value replacement; applied if provided.
    inplace : If True, modify df in place; otherwise, work on a copy.

    """
    df = df if inplace else df.copy()
    s = df[col].str
    if strip:
        df[col] = s.strip()
    if lower:
        df[col] = s.lower()
    if repl:
        df[col] = df[col].replace(repl)
    return df

# Usage examples:
df1 = clean_col(df, 'name')
df2 = clean_col(df, 'name', lower=False)
df3 = clean_col(df, 'name', repl={'mr':'mister'}, inplace=True)
```

Task 1

Write a function that would convert Fahrenheit to Celsius.

$$C = (F - 32) * 5/9$$

Task 2

Rewrite the salary computation with time-and-a-half for overtime and create a function called computepay which takes two parameters (hours and rate).

Enter Hours: 45
Enter Rate: 10

Pay: 475.0

475 = 40 * 10 + 5 * 15

```
40 * rate + overtime * rate * 1.5
```

Calling a function from someplace else

Step 1: Ensure both Python files are in the same directory.

In utils.py, define your function normally.

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

Step 2: At the top of your main file, import the function or the module:

```
# Option A: Import specific function
from utils import greet
# Option B: Import entire module
import utils
```

Step 3: Call on the function

```
greet("Alice") # or utils.greet(“Alice”)
```