This is the function header for the `print` function.

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

As you enter, type into the chat anything *interesting* you notice about this function header - all observations are welcome!

*Taken from: https://docs.python.org/3/library/functions.html

Functions

Functions

Functions

- Review of Functions
- Functions are Objects
- Namespaces and Scope
- Parameters
    - Parameter taxonomy
    - Parameter ordering
- Type Hints

Announcements!

Review of Functions

```
def f(x1, x2):
    # Do things
    return x3
```

```
def f(x1, x2):
    // Do things
    return x3
```

```
tup = (4, 3)
f(*tup)
```

```
def f(x1, x2):
    // Do things
    return x3
```

```
args = {"x1":4, "x2":3}
f(**args)
```

*When unpacking a dictionary, parameters are *bound to their names in the function header.*

Functions are Objects

Function Comments

- The first string literal inside a function body is the docstring.

```
def f(x1, x2):
    """
    Description: Does some things.

    Arguments:
    - x1 (int): The first x.
    - x2 (int): The second x.

    Returns:
    - int: Integer representing the third x.
    """
    # Does some things
    return x3
```

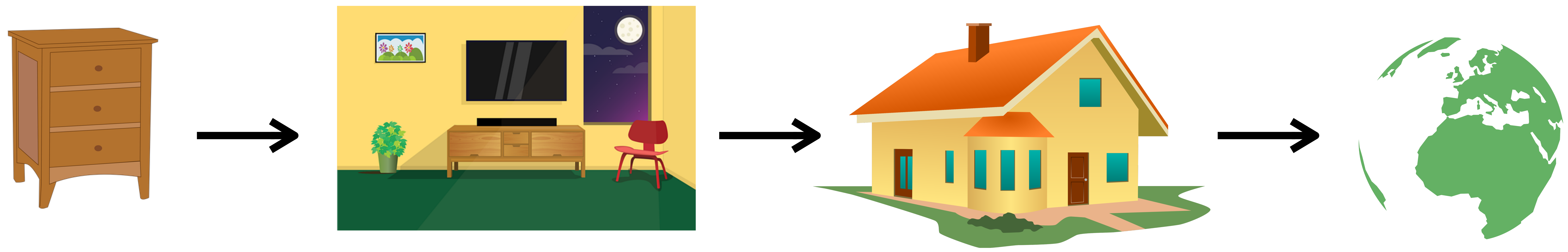
Namespaces and Scope

Namespace: a dictionary mapping names (strings) to objects within a certain *scope*.

`locals()` and `globals()` are both examples of namespaces.

Scope: the part of a program in which a certain namespace is valid (that is, where the name can be used to refer to the object).

Looking for my Keys



Local Scope → Enclosing Scope (if any) → Globals → Builtins

*If it's not in any of these places, Python raises a `NameError`.

Overwriting builtin function names (especially in the global scope) can be dangerous!

Why? (Type in chat!)

Parameter Taxonomy

What are Parameters?

Here's one!

Here's another one!

```
def f(x1, x2):
    // Do things
    return x3
```

Parameter Taxonomy

Parameter Taxonomy

- Positional-or-keyword arguments
- Positional-only arguments
- Keyword-only arguments
  - Default arguments
- Variadic positional arguments
- Variadic keyword arguments

```
def f(x1, x2):
    // Do things
    return x3
```

Positional Argument: when the function is called, this argument is bound to a name associated with a certain *position* in the function header.

Keyword Argument: when the function is called, this argument is bound to a name associated with the *name associated with it during the function call.*

```
f(3, x2=4)
```

`x1` is called by position; `x2` is called by name.

Positional-or-Keyword Arguments

Function Header: `f(x1, x2)`

Function Evaluation:

```
f(3, 2)
f(x1=3, x2=2)
f(3, x2=2)
```

Positional-Only Arguments

Function Header: `f(x1, x2, /)`

Function Evaluation: `f(3, 2)`

*Any arguments before the `/` are positional-only arguments!

Keyword-Only Arguments

Function Header: `f(*, x1, x2)`

Function Evaluation: `f(x1=3, x2=2)`

*Any arguments after the `*` are keyword-only arguments!

Default Arguments

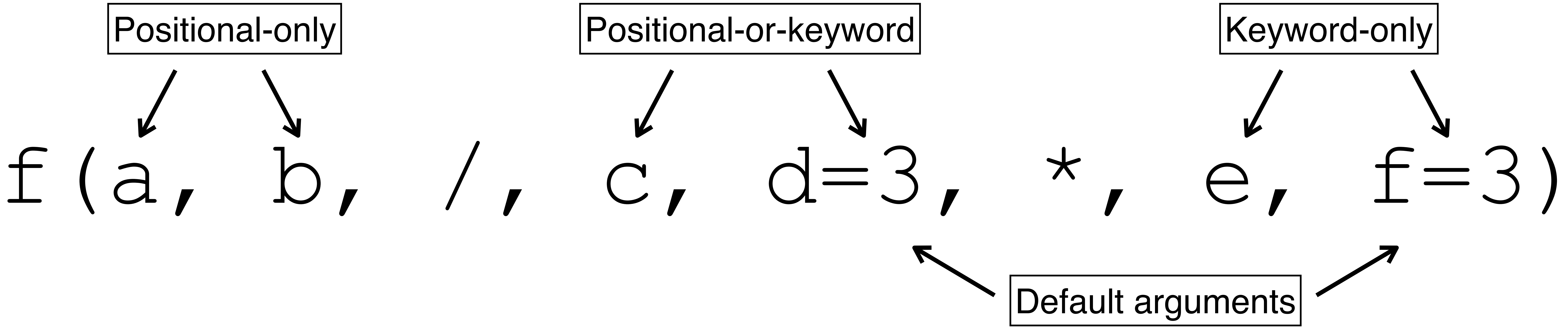
Assigns a "default" value to arguments during a function call.

Function Header: `f(x1, x2=2)`

Function Evaluation:

```
f(3)
f(3, 2)
f(x1=3, x2=2)
```

Name Each Type of Argument



```
f(a, b, /, c, d=3, *, e, f=3)
```

Variadic Arguments

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

```
print(1)
# => 1
print(1, 2)
# => 1 2
print(1, 2, 3, 4, 5, 6)
# => 1 2 3 4 5 6
```

How… many arguments does `print` accept?

Variadic Positional Arguments

Function Header: `f(*args)`

Function Evaluation:

```
f(3, 2)
f(3, 2, 1, 2, 3, 2, 1)
f(*(3, 1, 4, 1, 5, 9))
```

*The parameter can be named whatever you please: but `*args` is conventional.

Variadic Keyword Arguments

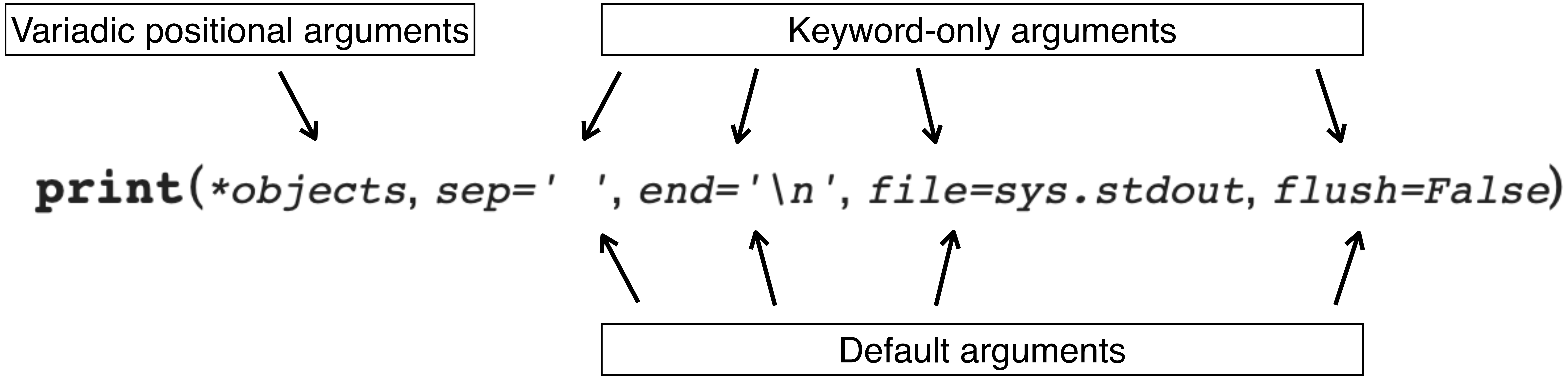
Function Header: `f(**kwargs)`

Function Evaluation: `f(a=3, b=2, CS="41")`

`f(**{"a":3, "b":2, "CS":"41"})`

*The parameter can be named whatever you please: but `**kwargs` is conventional.

Returning to `print`



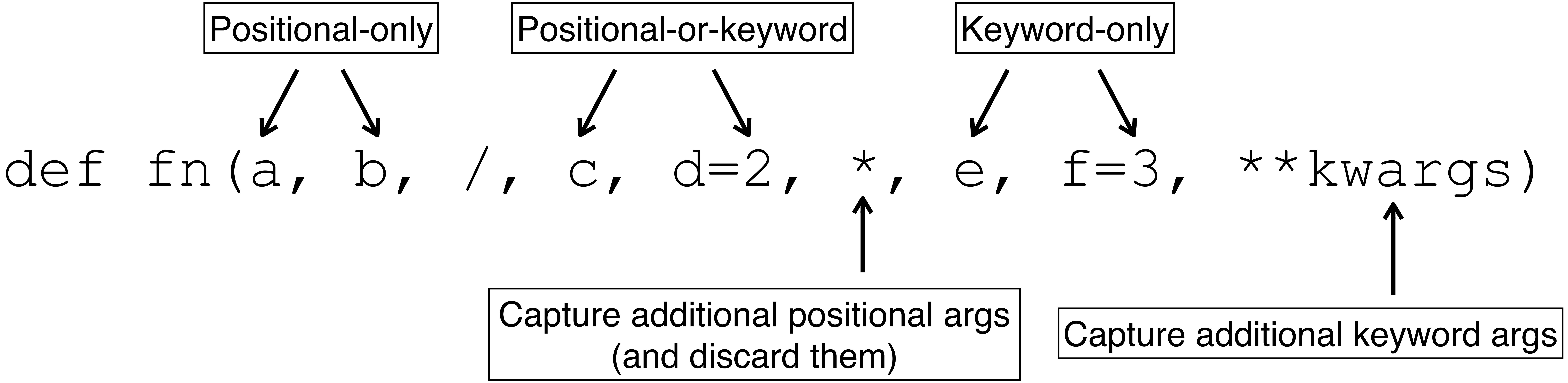
```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Parameter Ordering Rules

Parameter Ordering Rules

1. Keyword arguments follow positional arguments.
    - Default arguments (of each type) must follow non-default arguments of that type.
2. All arguments must identify some parameter. (Even positional ones!)
3. No parameter may receive a value more than once.

Parameter Rules in Action



```python
def fn(a, b, /, c, d=2, *, e, f=3, **kwargs)
```

Positional-only
Positional-or-keyword
Keyword-only
Capture additional positional args (and discard them)
Capture additional keyword args

The Universal Function Header

Function Header: `f(*args, **kwargs)`

Function Evaluation: `f(a=3, b=2, CS="41")`

`f(**{"a":3, "b":2, "CS":"41"})`

Type Hints

Duck Typing

"If it walks like a duck, and it quacks like a duck, it must be a duck."

- Python's philosophy toward objects: the type of an object is less important than the methods it defines.
    - E.g. you can use `+` on any object that defines an `__add__` method.

```
def f(x1: int, x2: int) -> int:
    # Do things
    return x3
```

Why Type Hints?

- Readability
- Optional strong-typing (type checking with packages like `mypy`!)

Functions

Functions

- Review of Functions
- Functions are Objects
- Namespaces and Scope
- Parameters
  - Parameter taxonomy
  - Parameter ordering
- Type Hints