Python Programming: Advanced

Lesson Objectives

After this lesson, you will be able to…

- Review all topics to this point.
- Use keyword arguments in functions.

Review: Functions

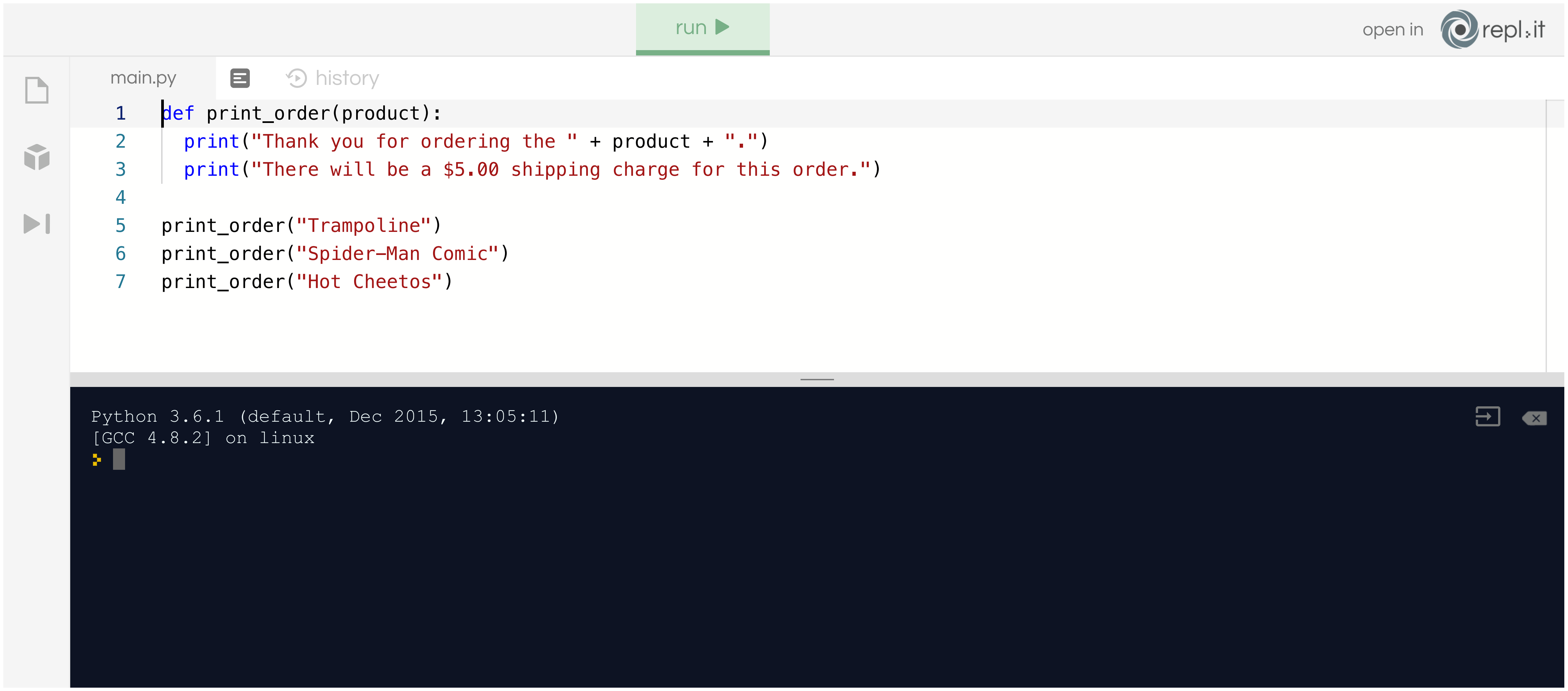
Main points:

- Define functions using the `def` keyword.
- A function must be **called** before the code in it will run!
- You will recognize function calls by the `()` at the end.

```python
# This part is the function definition!
def say_hello():
    print("hello world!")

# This part is actually calling/running the function!
say_hello()
```

Review: Function Arguments

- Provide an argument to a function when you need something small to vary.



```python
def print_order(product):
  print("Thank you for ordering the " + product + ".")
  print("There will be a $5.00 shipping charge for this order.")

print_order("Trampoline")
print_order("Spider-Man Comic")
print_order("Hot Cheetos")
```

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
```

Multiple Parameters

Functions can have…

```
# No parameters
def add_2_and_3():
    x = 2 + 3
    print(x)


# One parameter
def add_2(x):
    print(x + 2)


# Multiple parameters
def add(x, y, z):
```

Discussion: Print vs Return

Why doesn't this do anything?

```
def add(x, y, z):
    return x + y + z


add(1, 2, 3) # does nothing!
```

We Do: Review Exercises

Locally, let's create a file called `function_practice.py`.

- We'll define a function named `areBothEven`.
- It will accept two parameters: `num1` and `num2`.
- Inside the function, we'll return `True` if `num1` and `num2` are both even but `False` if they are not.
- We'll test this with `print(areBothEven(1, 4))`, `print(areBothEven(2, 4))`, and `print(areBothEven(2, 3))`.

We Do: Another Review Exercise!

In our file, we'll define another function named `lightOrDark` that takes the parameter `hour`.

- If `hour` is greater than 24, the function will print "That's not an hour in the day!" and **return nothing.**
- If `hour` is less than 7 or greater than 17, the function will return "It's dark outside!"
- Otherwise, the function will return "It's light outside!"
- We'll test this with `print(lightOrDark(4))`, `print(lightOrDark(26))`, and `print(lightOrDark(10))`.

Discussion: Arguments

Now, let's make functions a little more sophisticated.

What do you think the following code does?

```
def multiply(x, y):
    print(x * y)

multiply(1, 2, 3) # Too many arguments! What happens?
```

What if we want all of these to work?

```
def multiply(x, y):
    print(x * y)

multiply(4, 5, 6)
multiply(4, 5)
multiply(4, 5, 2, 7, 3, 9)
```

Introducing `*args`

`*args` is a parameter that says “Put as many parameters as you’d like!”

- Pronounced like a pirate - “arrrrghhhs!”
- Known as **positional arguments**
- The `*` at the beginning is what specifies the variable number of arguments

```
def multiply(*args):
    product = 1

    # We don't know the number of args, so we need a loop
    for num in args:
        product *= num
    print(product)

multiply(4, 5, 6) # Prints 120!
```

We Do: `*args`

Let's create a local file for this lesson - `args_practice.py`.

- We'll write a function, `sum_everything` that takes any numbers of arguments and adds them together.
- At the end, we'll print out the sum.
- Let's try it with `sum_everything(4, 5, 6)` and `sum_everything(6, 4, 5)`. The order doesn't matter!
- `*args` says "any number" - you can pass in none at all!

Discussion: Often, Order Does Matter.

Let’s switch gears. Back to a set number of arguments!

Check this out:

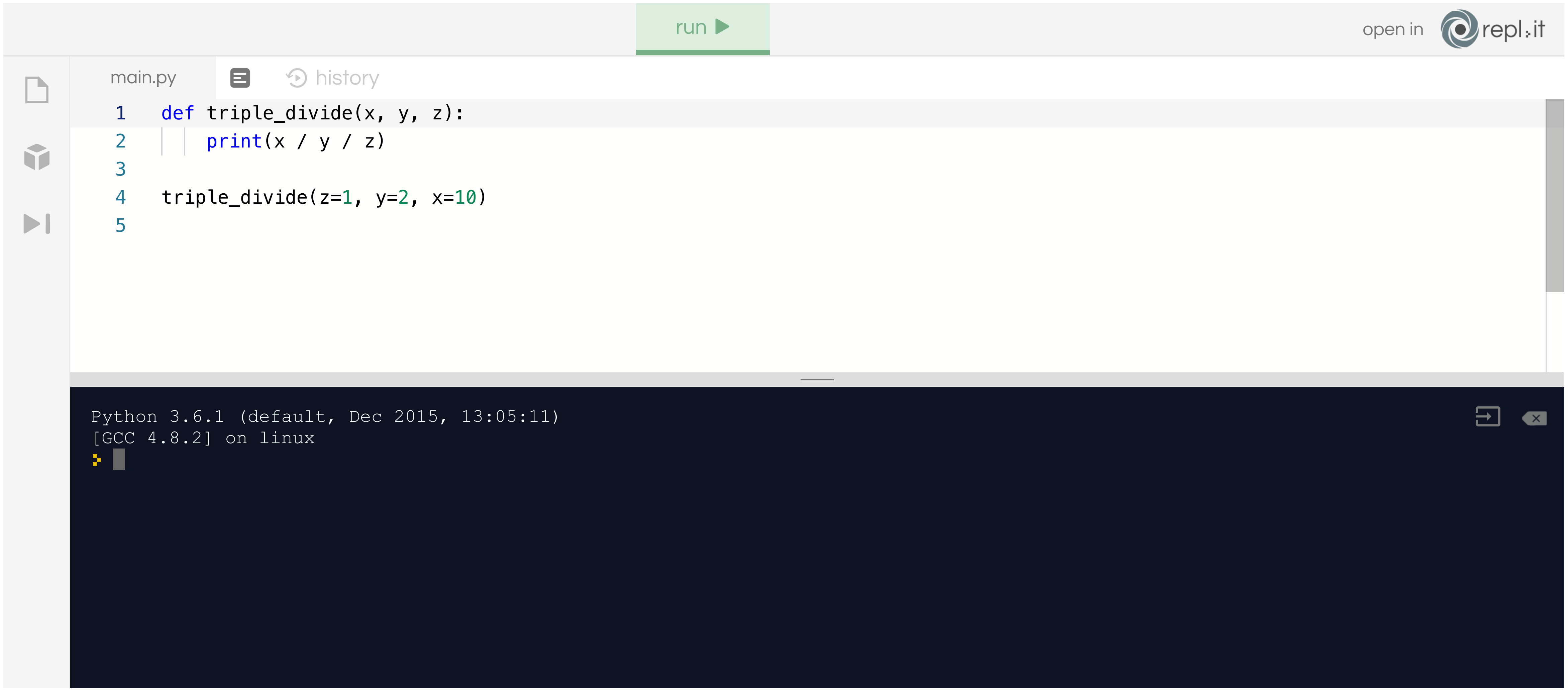
```
def triple_divide(x, y, z):
    print(x / y / z)


triple_divide(1, 2, 10) # Prints 0.05
```

Without otherwise specifying, `x` is `1`, `y` is `2`, and `z` is `10`.

- What if we want `x`, the first parameter to get the value `10`?
- Is there a way to specify which argument goes to which parameter?

Forcing the Order

Here we've forced the order to be reversed from the default. In fact, we can specify any ordering we want by using the names of the parameters (keywords) when providing the argument values.



```python
def triple_divide(x, y, z):
    print(x / y / z)

triple_divide(z=1, y=2, x=10)
```

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
› 
```

Take a moment to play around with the values until you really believe it!

Keyword Arguments (kwargs)

Using kwargs, odrer deons’t mtater:

- Arguments are named according to their corresponding parameters.
- Order doesn’t matter - Python will check the names and match them!
- Values are assigned because the *keyword argument* and the *parameter name* match.

```
def triple_divide(x, y, z):
    print(x / y / z)


triple_divide(x=10, y=2, z=1)
# This runs 10 / 2 / 1, and prints 5
triple_divide(y=2, z=1, x=10)
# This ALSO runs 10 /  2 / 1, and prints 5.
```

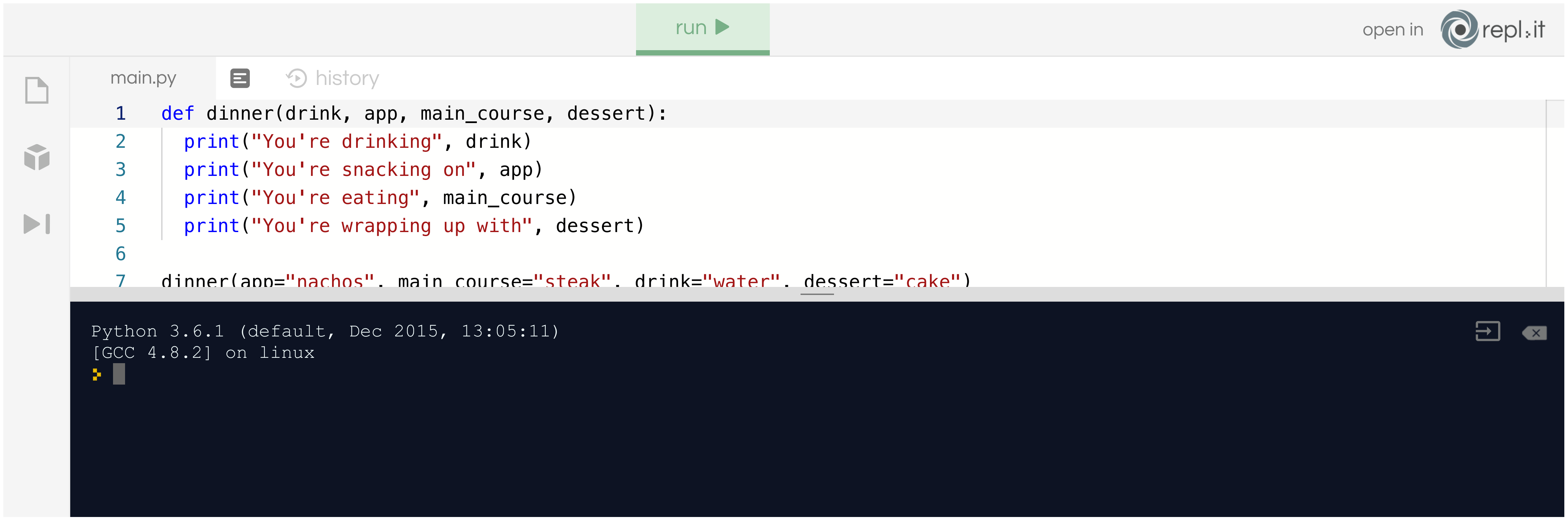
Protip*: Keep your parameter names simple and concise to prevent typos and misspellings!*

Mix It Up.... but Not With Every Argument?

Fun fact: You can provide some args in order - **positional** - and some with keywords.

- Undefined are assigned in sequential order.
- Keywords have to come last! - then, in any order.

```
dinner(app="chicken wings", main_course="medium rare steak", drink="water",
dinner("chicken wings", "water", dessert="milkshake", main_course="medium ra
```



```
def dinner(drink, app, main_course, dessert):
  print("You're drinking", drink)
  print("You're snacking on", app)
  print("You're eating", main_course)
  print("You're wrapping up with", dessert)

dinner(app="nachos", main_course="steak", drink="water", dessert="cake")
```

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
›
```

Quick Review

`*args`: Any number of arguments - even 0! - can be passed in.

```
def sum_everything(*args):
    sum = 0

    for num in args:
        sum += num
    print(sum)

sum_everything(4, 5, 6) # Prints 15
```

Keyword arguments (kwargs): Arguments can be passed in out of order.

```
def divide(first, second, third):
    print(first / second / third)

divide(first=10, second=2, third=1)
divide(second=2, third=1, first=10)
```

Discussion: Variable Numbers of Kwargs?

What if I go to Froyo? I need:

- One argument `spoon`, to pick a spoon size.
- A variable number of arguments for all the flavors of frozen yogurt I might eat!

`def yogurt_land(*args)`?

- No! `*args` won’t work - we need to know which arg is the spoon.

`def yogurt_land(spoon, froyo)`?

- No! We don’t know the number of froyo arguments.

Any ideas?

Introducing: `**kwargs`

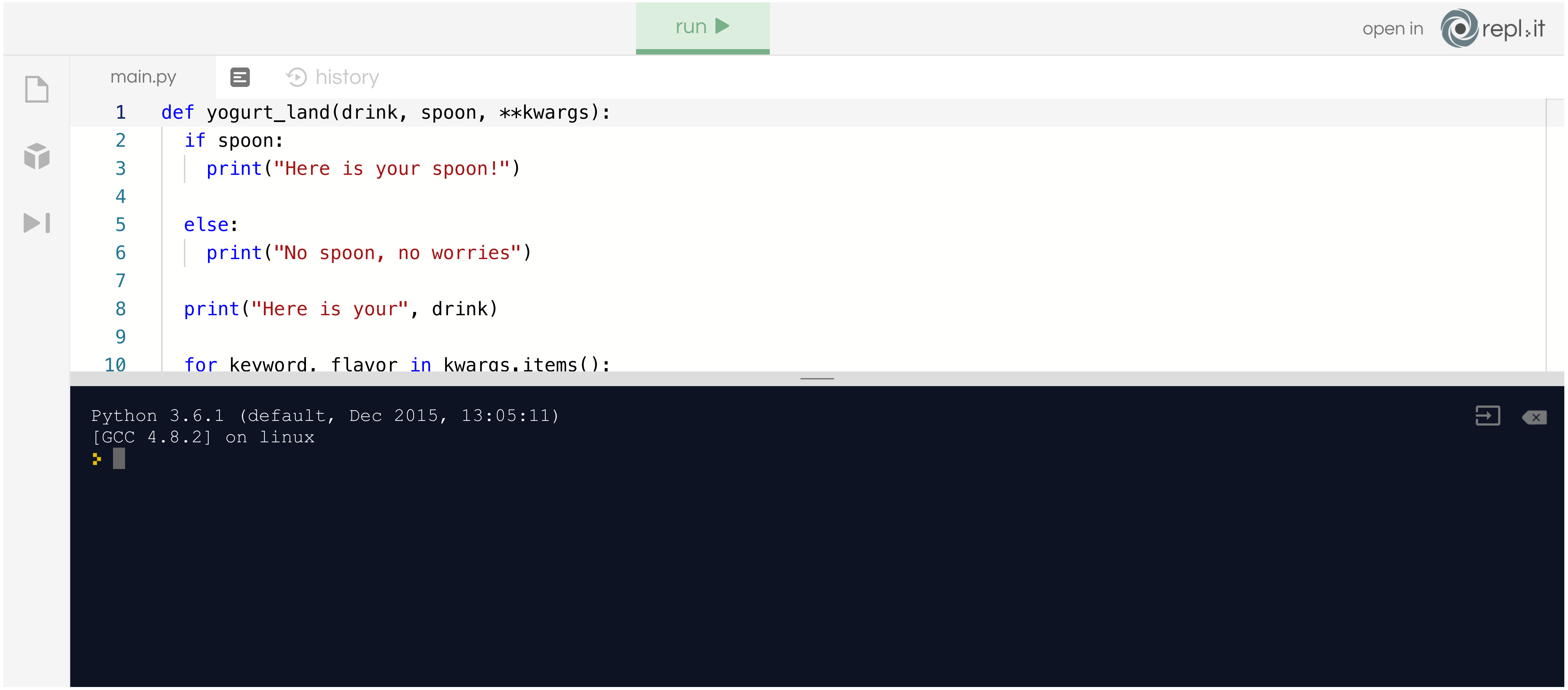
The `*` in `*args` means: Any number of arguments.

Let’s add `**` to our kwargs: `**kwargs` can take a variable number of arguments. Note the double `**`!

```
def yogurt_land(spoon, **kwargs):
  print(spoon)
  # We need a loop, because we don't know how many kwargs there are.
  for keyword, flavor in kwargs.items():
    # kwargs.items has the keyword and the value, which we're calling "flavo
    print("My", keyword, "is a", flavor)


# Like before, the unnamed arg has to come first!
yogurt_land("large!", first_froyo="vanilla", second_froyo="chocolate", third
```

We Do: 4 Froyos

- Can we subtract one of the froyos?
- Where is my 4th froyo?
- What if I drop all my froyos on the ground? (No kwargs)
- Can I skip the drink or spoon positional arguments?



```python
def yogurt_land(drink, spoon, **kwargs):
    if spoon:
        print("Here is your spoon!")

    else:
        print("No spoon, no worries")

    print("Here is your", drink)

    for keyword, flavor in kwargs.items():
```

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
>
```

Quick Review of Useful Argument Types:

At this point, we have `*args`, `kwargs` and `**kwargs`:

```python
# Args: Any number of arguments:

def multiply(*args):

    product = 1

    for num in args:

        product *= num


multiply(4, 5, 6)


# Kwargs: Named (keyword) arguments

def triple_divide(x, y, z):

    print(x / y / z)
```

Discussion: Printing

`print` is a function! That's why it has parentheses! - It's built into Python, so you don't have to define it. You can just use it.

When printing, commas automatically add spaces:

```
print("Hi!", "Vanilla,", "please.")
```

But since `print` is a function, too - do you think there's anything we can do to change those spaces to something else?

```
# Hi!-Vanilla,-please,-but-chocolate-is-cool.

# Hi!-and-Vanilla,-and-please.
```

Print is AWESOME: Optional Parameters

Turns out…

- `print` accepts an optional keyword argument: `sep`.

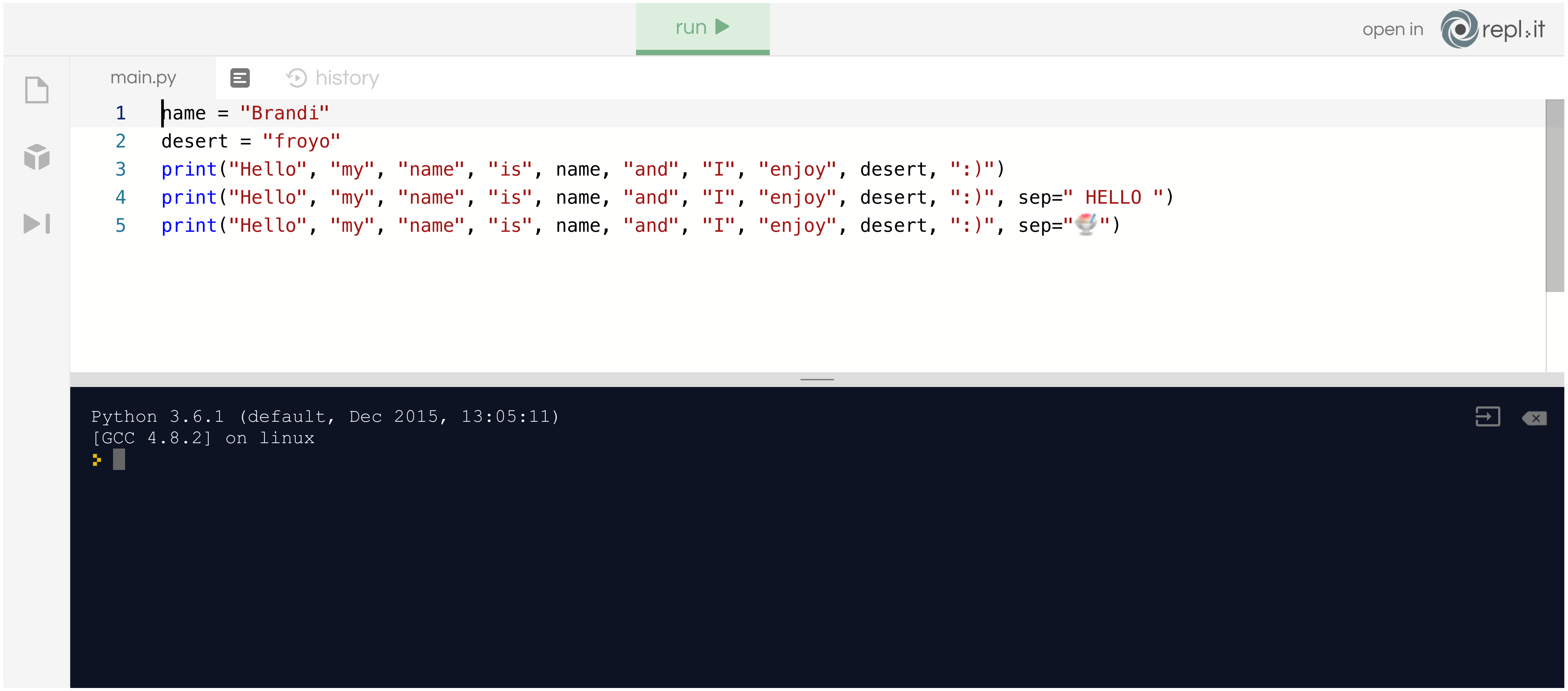
The `sep` value given will be used as a **separator.**

- It’s optional! Without it, `print` by default uses a space, which is why you haven’t seen it.
- **This only applies when using commas.**

```
print("Hi!", "Vanilla", "please,",  "but", "chocolate", "is", "cool.", sep="
# => Hi!--Vanilla,--please,--but--chocolate--is--cool.
```

Delicious Printing

- We can replace `name` and `dessert` with your own name and favorite dessert. It's a regular print!
- `sep` can be any string, or even an icon (they're made of strings - we'll see later!) - but not an int.



```python
name = "Brandi"
desert = "froyo"
print("Hello", "my", "name", "is", name, "and", "I", "enjoy", desert, ":)")
print("Hello", "my", "name", "is", name, "and", "I", "enjoy", desert, ":)", sep=" HELLO ")
print("Hello", "my", "name", "is", name, "and", "I", "enjoy", desert, ":)", sep="🍧")
```

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
›
```

Quick Review

So far, we've learned:

- `*args`:
  - A variable number of function arguments.
- kwargs:
  - A set number of function arguments.
  - Can be defined out of order
- `**kwargs`:
  - Any number of positional arguments.
- `sep` in print.

There's one more: Optional parameters.

Optional Parameters with Default Values

This idea exists in programming - you've already seen it!

The default value for `sep` in `print` is `" "`. You don't **need** to include it.

This makes it optional! **Optional parameters** have default values, so you don't need to include them.

- Only include them if you want to change them!

```
# Here, `sep` is optional to include. It defaults to a space " ".
print("Hello", "my", "name", "is", name, "and", "I", "enjoy", dessert, ":)")

# But we can include it, if we want, and `sep` will use our value instead of
print("Hello", "my", "name", "is", name, "and", "I", "enjoy", dessert, ":)",
```

Default parameters are in the *function declaration*.

They're there if you don't include a value.

Any Functions: Optional Parameters with Default Values

These can be added to any functions.

Here, `c` has a default of `20`. We don’t need to include it!

```
# Optional parameters: Default values are only used if needed.
def my_func(a, b, c=20):
    print(a + b + c)
my_func(1, 2)
# Uses the default! Prints 23.
my_func(1, 2, 4)
# Overrides the default! Prints 7.
```

Partner Exercise: Poke At It!

Pair up! Choose a driver and a navigator.

- In your local file, write a function, `print_food` that has four optional parameters (all with defaults of your choice): `favorite_food`, `lunch_today`, `lunch_yesterday`, and `breakfast`.

`print_food` should print out each of these.

Call this with a couple different arguments:

- No arguments.
- All arguments - a regular function call.
- 2 keyword arguments. Give all four arguments, but use a keyword for `lunch_yesterday` and `breakfast`.
- All keyword arguments - out of order.

Partner Exercise: Keep Poking!

Change roles!

Underneath `print_food`, rewrite it, twice.

First, write `print_food_args`, using `*args` as the parameter. Start the function by printing `args`, so you can see what's going on. Then, print the values you pass in.

Then, write `print_food_kwargs`, using `**kwargs` as the parameter. Start the function by printing `kwargs`, so you can see what's going on. Then, as above, print the values you pass in.

Summary + Q&A

- `*args`:
    - A variable number of function arguments.
    - Taken in any order.
    - `def multiply(*args):`
- kwargs:
    - A set number of function arguments.
    - Can be defined out of order
    - `my_func(a=1, b=2, c=3)`
- `**kwargs`: - Any number of positional arguments.
    - `def froyo(*kwargs)`
- sep in print.
- Optional parameters:
    - Default values in the function declaration
    - `def my_func(a=10, b=15, c=20)`

Additional Resources

- Optional Parameter Repl.it
- Keyword Args
- Args and Kwargs
- Defining Functions