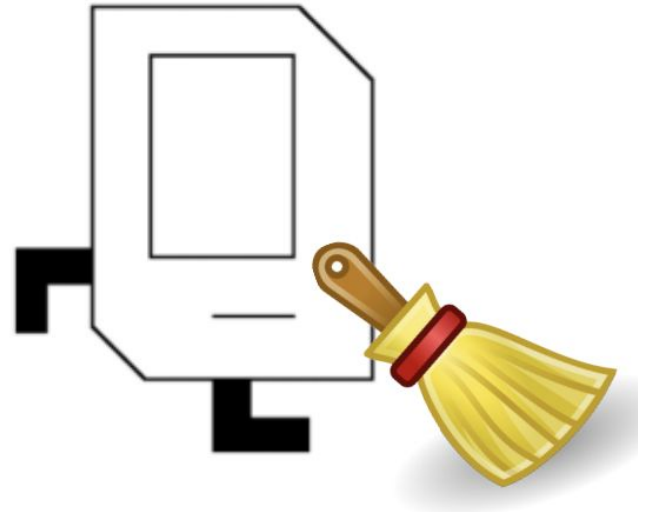
Function Execution

Brahm Capoor

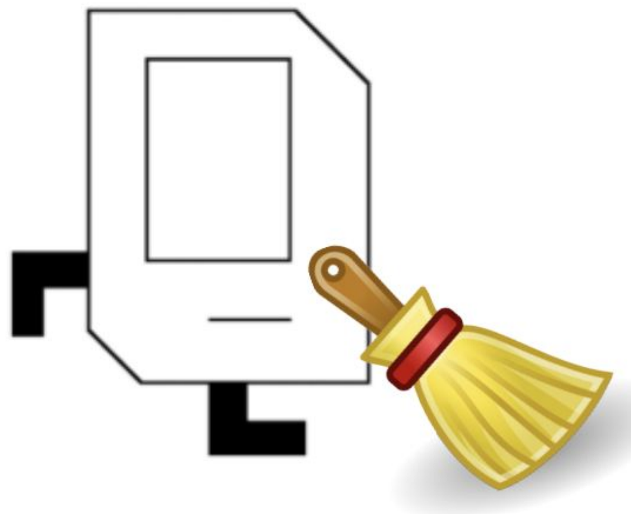
Some Housekeeping

- Assignment 7
  - New announcement: we are going to disregard your lowest assignment grade
  - We released it yesterday, and it's due next Wednesday at midnight
  - You can take *at most* one late day on the assignment
  - Reach out to us if you feel like you need further accommodations and we'd be happy to figure something out



Some Housekeeping

- The Game Plan for the next few lectures:
  - Today is the last lecture with new material
  - On Friday, we'll be discussing what to do next, now that you have CS 106A under your belt
  - Next Monday, we'll be celebrating your accomplishments so far and showing you some of our favorite contest entries

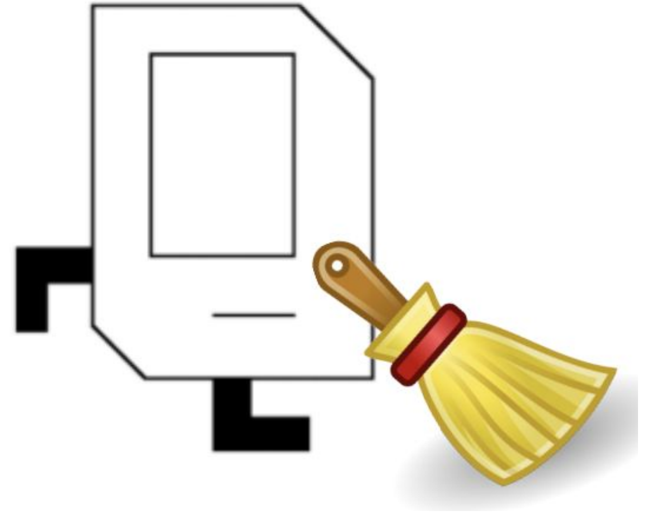


Some Housekeeping

- The Game Plan for the next few lectures:
  - Today is the last lecture with new material
  - On Friday, we'll be discussing what to do next, now that you have CS 106A under your belt
  - Next Monday, we'll be celebrating your accomplishments so far and showing you some of our favorite contest entries
  - There's no class next Wednesday

Some Housekeeping

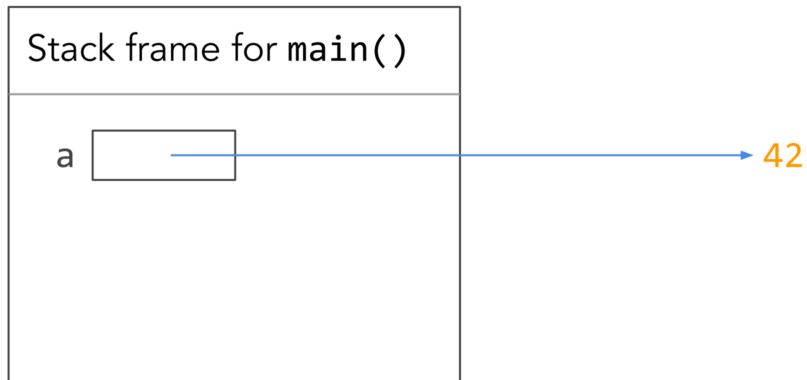
- I’ll be posting a couple of resources over the next few days
  - A video explaining how to package your Python programs as standalone applications
  - A video explaining in greater detail how to add UI components like buttons and text entry fields to graphical programs
- Neither of these are necessary for the contest or the assignment, but we thought you might find them helpful



What’s happening here?

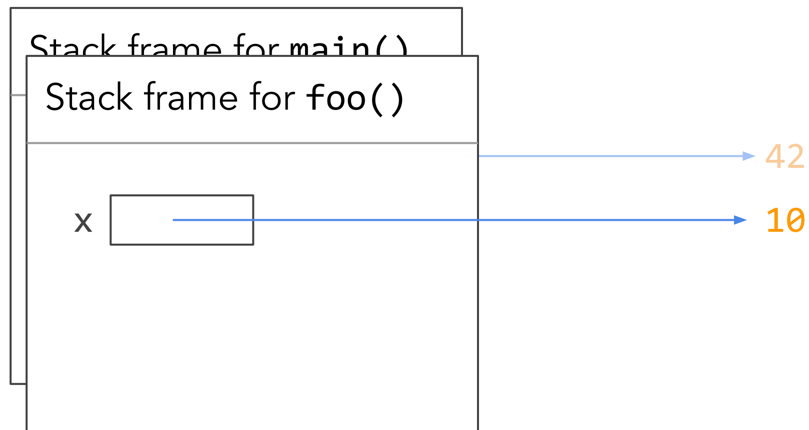
```
def foo():

    x = 10

    y = 12

    print(x  + y)

def main():

    a = 42

    foo()

if __name__ == “__main__”:

    main()
```

What’s happening here?



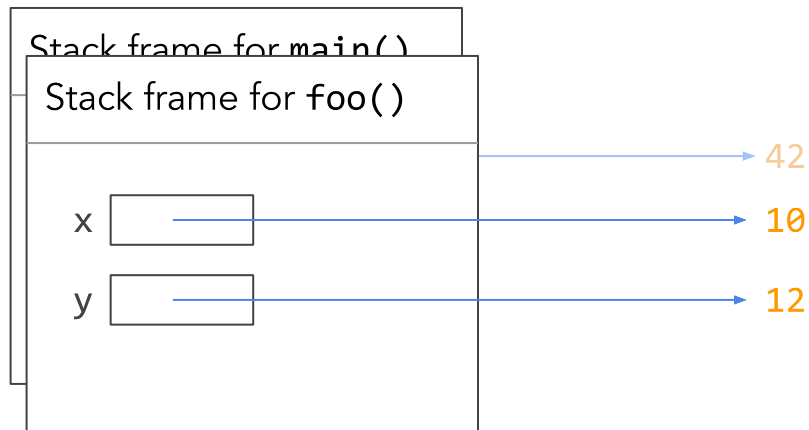

```
def foo():
    x = 10
    y = 12
    print(x  + y)

def main():
    a = 42
    foo()

if __name__ == “__main__”:
    main()
```

What’s happening here?



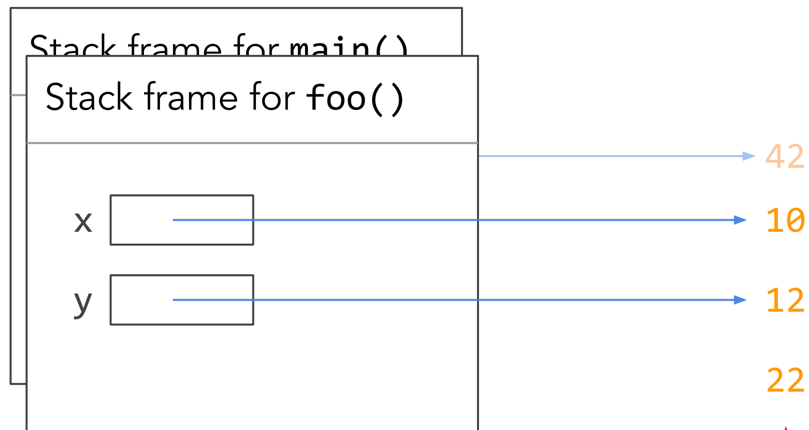

```
def foo():
    x = 10
    y = 12
    print(x  + y)

def main():
    a = 42
    foo()

if __name__ == “__main__”:
    main()
```

What's happening here?




```
def foo():
    x = 10
    y = 12
    print(x  + y)

def main():
    a = 42
    foo()

if __name__ == “__main__”:
    main()
```

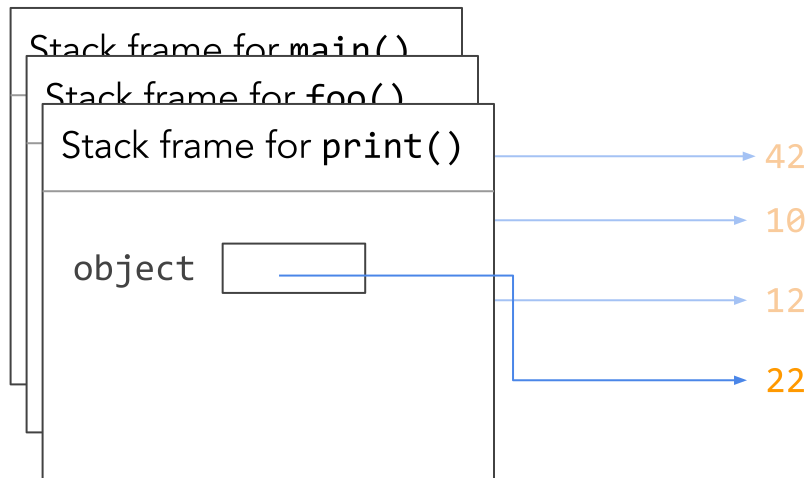
What’s happening here?



We calculate `x + y` in `foo`, but don’t give it a name

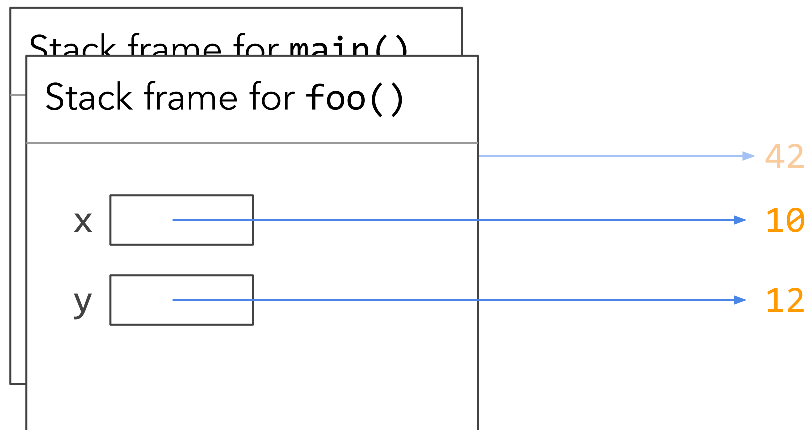
```
def foo():
    x = 10
    y = 12
    print(x  + y)

def main():
    a = 42
    foo()

if __name__ == “__main__”:
    main()
```

What’s happening here?



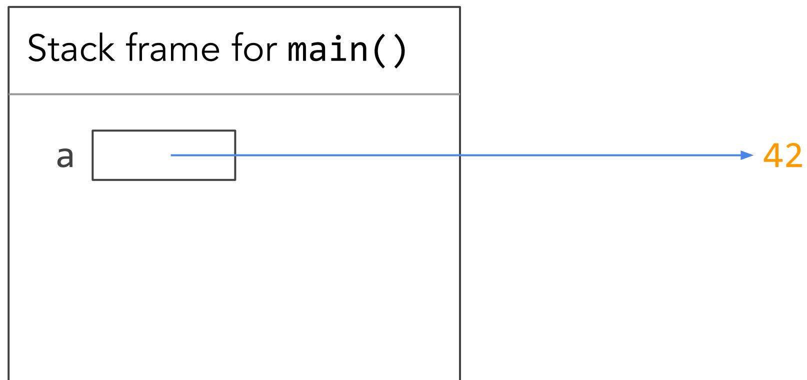

```
def foo():
    x = 10
    y = 12
    print(x  + y)

def main():
    a = 42
    foo()

if __name__ == “__main__”:
    main()
```

What's happening here?




```
def foo():
    x = 10
    y = 12
    print(x  + y)

def main():
    a = 42
    foo()

if __name__ == “__main__”:
    main()
```

What’s happening here?




```
def foo():
    x = 10
    y = 12
    print(x  + y)

def main():
    a = 42
    foo()

if __name__ == “__main__”:
    main()
```

What’s happening here?

```
def foo():
    x = 10
    y = 12
    print(x  + y)

def main():
    a = 42
    foo()

if __name__ == “__main__”:
    main()
```

We’ve been handwaving something

How did the function calls work?

How did Python know that these were all valid function names?

```
def foo():
    x = 10
    y = 12
    print(x  + y)

def main():
    a = 42
    foo()

if __name__ == “__main__”:
    main()
```

How did the function calls work?

How does Python know what isn't a valid function name?

```
NameError: name 'foob' is not defined
```

```
def foo():
    x = 10
    y = 12
    print(x  + y)

def main():
    a = 42
    foob()

if __name__ == "__main__":
    main()
```

How does Python work?

Python makes use of what it calls execution frames, which contain administrative information about code blocks

How does Python work?

Python makes use of what it calls execution frames, which contain administrative information about code blocks

The most important part of an execution frame is its namespace, which keeps track of the named values (variables) available to the programmer

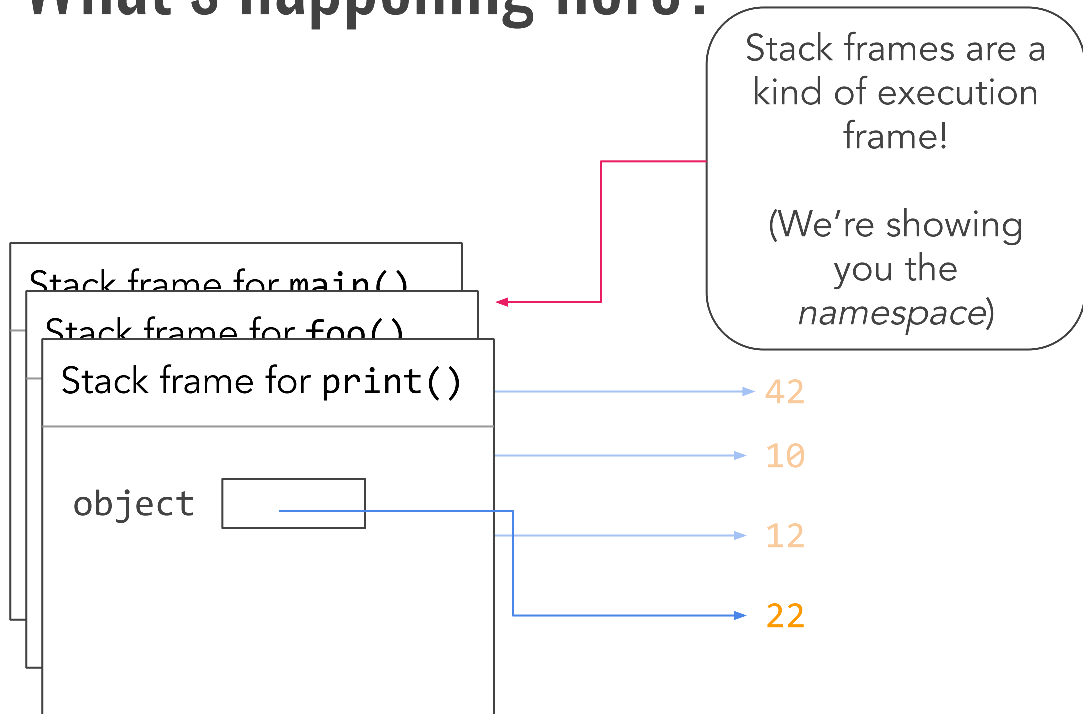
How does Python work?

Python makes use of what it calls execution frames, which contain administrative information about code blocks

The most important part of an execution frame is its namespace, which keeps track of the named values (variables) available to the programmer

You've seen these before!

What's happening here?




```python
def foo():
    x = 10
    y = 12
    print(x  + y)

def main():
    a = 42
    foo()

if __name__ == "__main__":
    main()
```

Stack frames

Stack frames keep track of all the values available in a function, and the names we give them

Every time we make a variable (say, `a = 42`), we are adding a name-value pair to the function's namespace

Stack frames

Stack frames keep track of all the values available in a function, and the names we give them

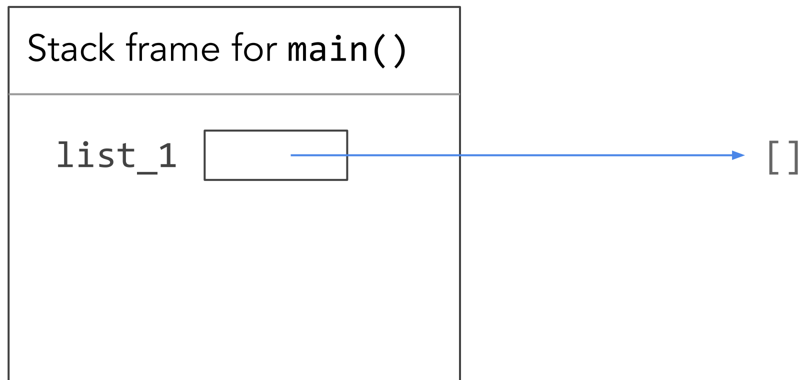
Every time we make a variable (say, `a = 42`), we are adding a name-value pair to the function's namespace

Formally, we're *binding* the name `a` to the object 42

Objects can have multiple names

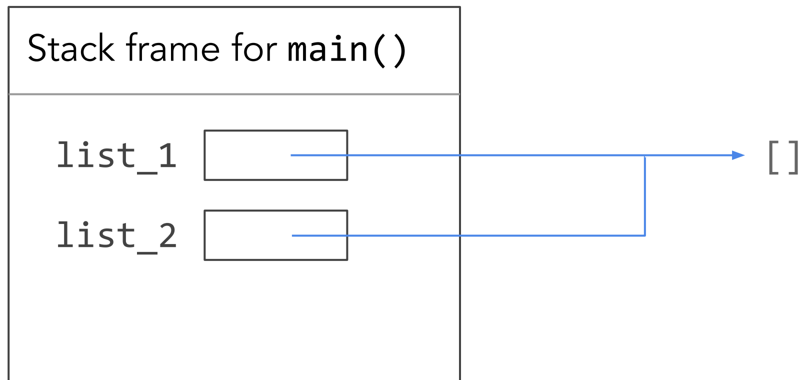
```
def main():
    list_1 = []
    list_2 = list_1
    list_2.append(42)

if __name__ == "__main__":
    main()
```

Objects can have multiple names



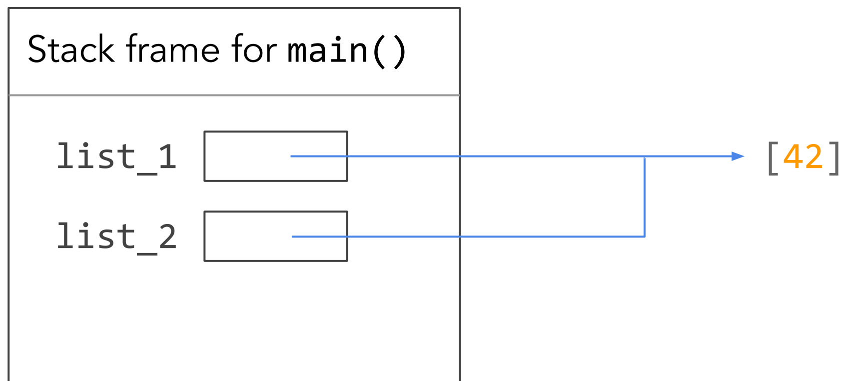

```
def main():
    list_1 = []
    list_2 = list_1
    list_2.append(42)

if __name__ == “__main__”:
    main()
```

Objects can have multiple names



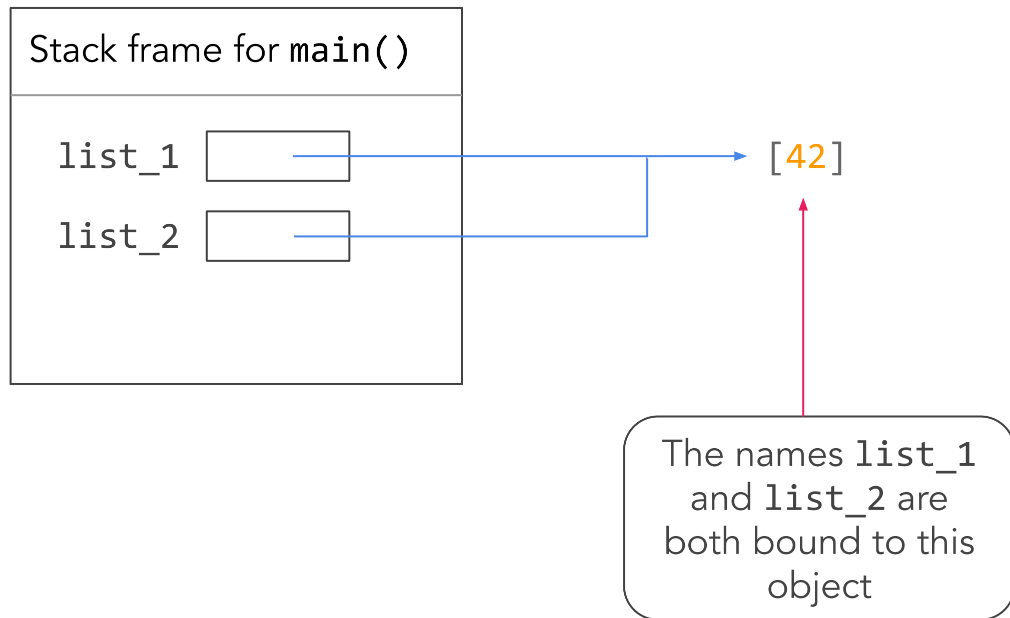

```
def main():
    list_1 = []
    list_2 = list_1
    list_2.append(42)


if __name__ == “__main__”:
    main()
```

Objects can have multiple names




```
def main():
    list_1 = []
    list_2 = list_1
    list_2.append(42)


if __name__ == “__main__”:
    main()
```

Objects can have multiple names




```
def main():
    list_1 = []
    list_2 = list_1
    list_2.append(42)

if __name__ == “__main__”:
    main()
```

Every program also has an execution frame

This execution frame is set up as soon as the program begins

Every program also has an execution frame

This execution frame is set up as soon as the program begins

- The program's namespace is set up as part of this to set up the constants and functions

What happens when a program runs?

```
CONSTANT_NUM = 42

def foo():
    pass

def bar():
    pass

def main():
    pass

if __name__ == “__main__”:
    main()
```

We make an execution frame for the program, whose namespace is initially empty

Execution frame for program

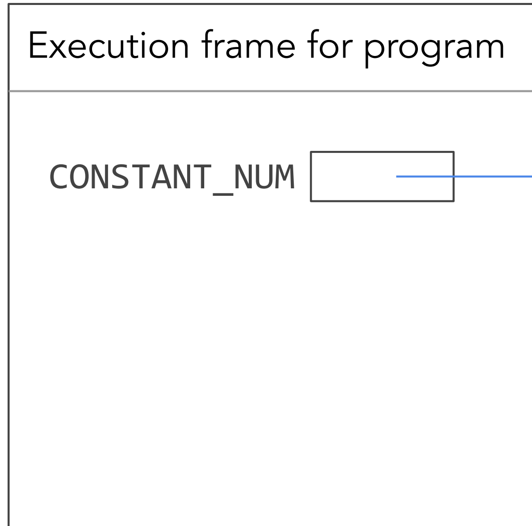
```
CONSTANT_NUM = 42

def foo():
    pass

def bar():
    pass

def main():
    pass

if __name__ == "__main__":
    main()
```

We read the file from *top to bottom*

Execution frame for program

```
CONSTANT_NUM = 42

def foo():
    pass

def bar():
    pass

def main():
    pass

if __name__ == "__main__":
    main()
```

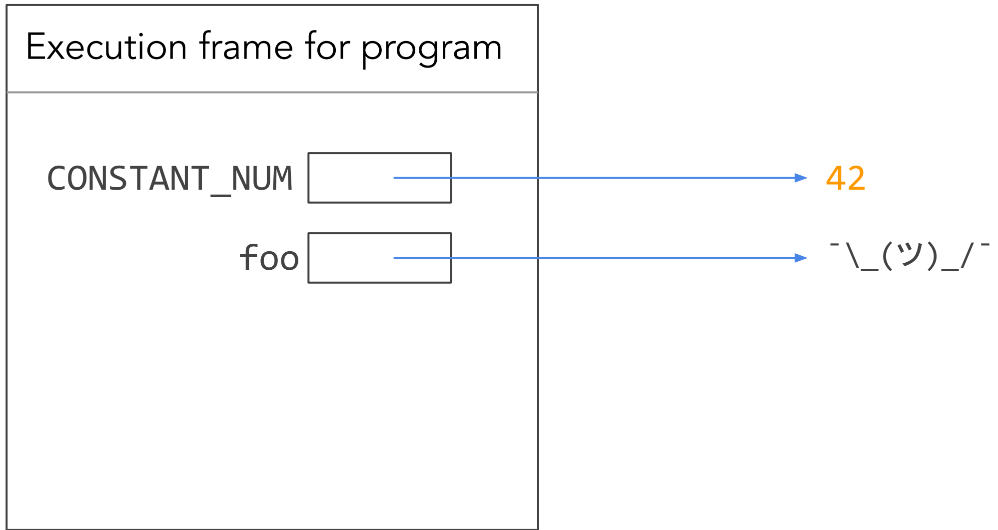
Whenever we see a variable declaration like for `CONSTANT_NUM`, we add it to the namespace



42

```
CONSTANT_NUM = 42

def foo():
    pass

def bar():
    pass

def main():
    pass

if __name__ == “__main__”:
    main()
```

Whenever we see a function with a def statement, we add it to the namespace as well




```
CONSTANT_NUM = 42

def foo():
    pass

def bar():
    pass

def main():
    pass

if __name__ == “__main__”:
    main()
```

(Don't worry about what the object is for now)

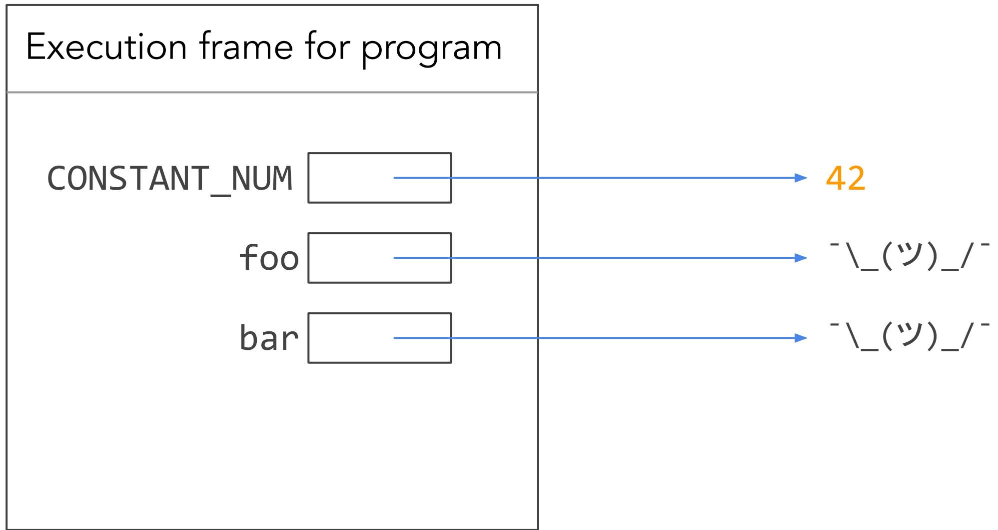
Execution frame for program

CONSTANT_NUM → 42

foo → ¯_(ツ)_/¯

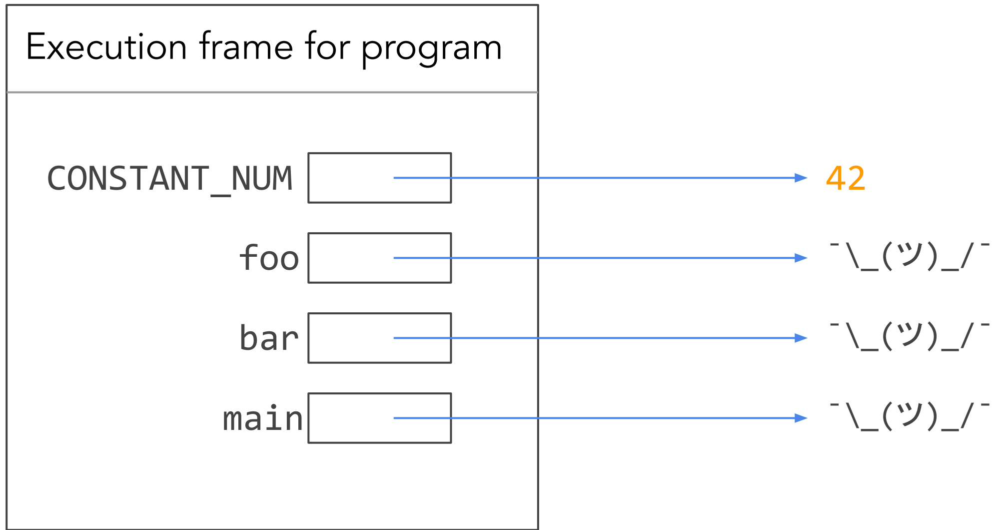
```
CONSTANT_NUM = 42

def foo():
    pass

def bar():
    pass

def main():
    pass

if __name__ == "__main__":
    main()
```

Whenever we see a function with a def statement, we add it to the namespace as well



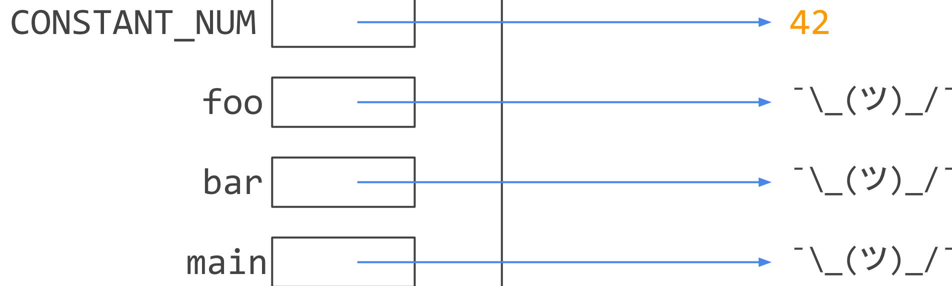

```
CONSTANT_NUM = 42

def foo():
    pass

def bar():
    pass

def main():
    pass

if __name__ == “__main__”:
    main()
```

Whenever we see a function with a def statement, we add it to the namespace as well




```
CONSTANT_NUM = 42

def foo():
    pass

def bar():
    pass

def main():
    pass

if __name__ == “__main__”:
    main()
```

This if statement checks whether we're directly running the program from the terminal (rather than importing it)

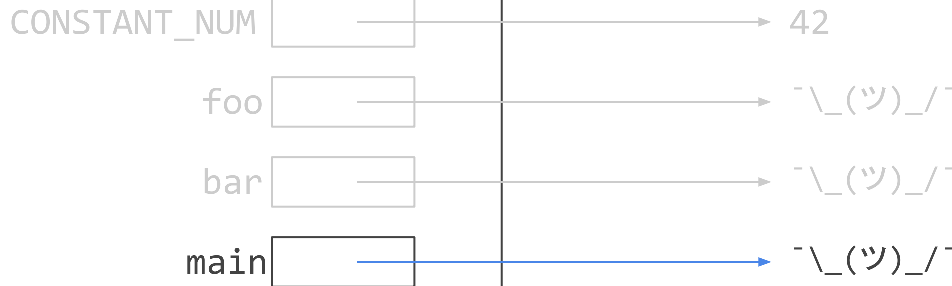



```
CONSTANT_NUM = 42

def foo():
    pass

def bar():
    pass

def main():
    pass

if __name__ == "__main__":
    main()
```

We are, so we call the `main` function

```
CONSTANT_NUM = 42

def foo():
    pass

def bar():
    pass

def main():
    pass

if __name__ == “__main__”:
    main()
```

This `if` statement needs to be at the bottom of the program to make sure that `main` has already been added to the namespace

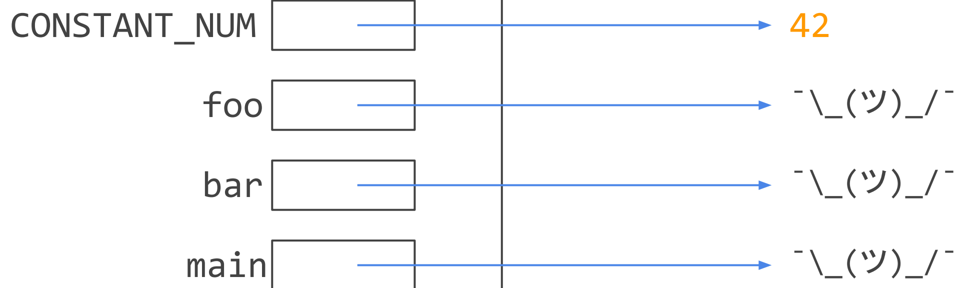



```
CONSTANT_NUM = 42

def foo():
    pass

def bar():
    pass

def main():
    pass

if __name__ == “__main__”:
    main()
```

The program’s namespace is available everywhere

Every function you define in your program has access to the names in the program namespace, in addition to its own local names

Any function in this program can call `foo`, `bar` or `main`, or refer to `CONSTANT_NUM`




```
CONSTANT_NUM = 42

def foo():
    pass

def bar():
    pass

def main():
    pass

if __name__ == “__main__”:
    main()
```

The program’s namespace is available everywhere

When you use a name (for example, by calling a function or using a variable), you follow these steps:

The program’s namespace is available everywhere

When you use a name (for example, by calling a function or using a variable), you follow these steps:

1. Search in the current stack frame’s namespace

The program’s namespace is available everywhere

When you use a name (for example, by calling a function or using a variable), you follow these steps:

1. Search in the current stack frame’s namespace
2. If it’s not there, search in the program namespace

The program's namespace is available everywhere

When you use a name (for example, by calling a function or using a variable), you follow these steps:

1. Search in the current stack frame's namespace
2. If it's not there, search in the program namespace
3.

The program's namespace is available everywhere

When you use a name (for example, by calling a function or using a variable), you follow these steps:

1. Search in the current stack frame's namespace
2. If it's not there, search in the program namespace
3. `NameError`

This is useful, but not actionable

...so why am I talking about it?

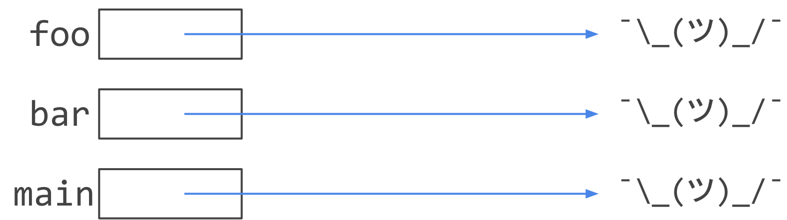
What am I handwaving?

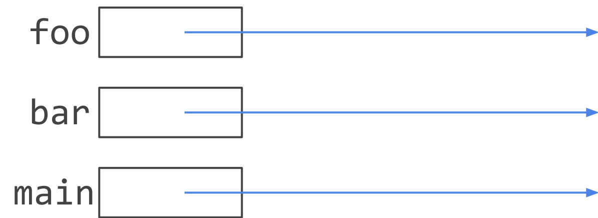
Execution frame for program

```python
CONSTANT_NUM = 42

def foo():
    pass

def bar():
    pass

def main():
    pass

if __name__ == “__main__”:
    main()
```

What these function names are pointing at?

```
CONSTANT_NUM = 42

def foo():
    pass

def bar():
    pass

def main():
    pass

if __name__ == "__main__":
    main()
```


foo
bar
main
¯_(ツ)_/¯
¯_(ツ)_/¯
¯_(ツ)_/¯



`foo`, `bar` and `main` are all function names which point at function objects

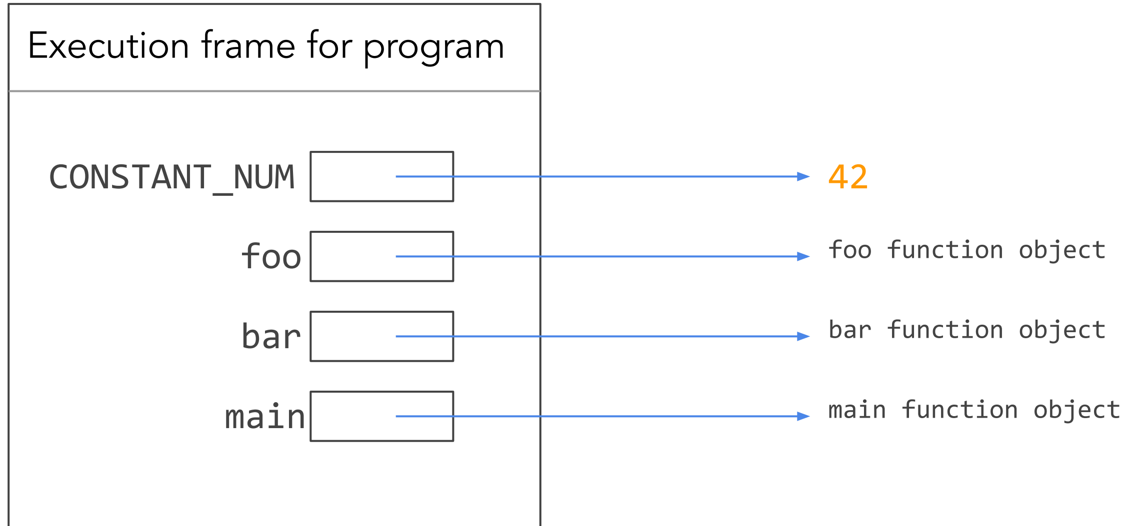
Function Objects

Each time you've made a function using `def`, you've made a function object to which you've bound a function name

Function objects are a very special kind of object

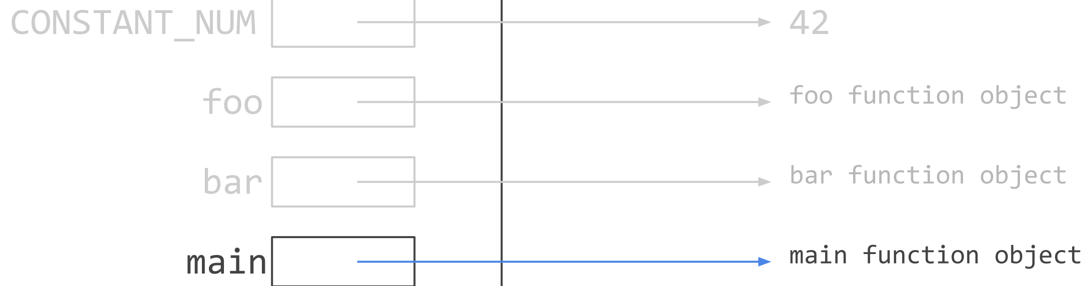
- We almost never use their instance variables
- By default, they have only one behaviour, and that is to execute a certain set of instructions on an input and return the output

How does this work?




```python
CONSTANT_NUM = 42

def foo():
    pass

def bar():
    pass

def main():
    pass

if __name__ == “__main__”:
    main()
```

Every time we call a function, we automatically invoke the calling behaviour of the function object




```
CONSTANT_NUM = 42

def foo():
    pass

def bar():
    pass

def main():
    pass

if __name__ == “__main__”:
    main()
```

The consequences

Since functions are objects, we can treat them (mostly) like we can other objects

The consequences

Since functions are objects, we can treat them (mostly) like we can other objects

- We can give them additional names

The consequences

Since functions are objects, we can treat them (mostly) like we can other objects

- We can give them additional names
- We can pass them as parameters

The consequences

Since functions are objects, we can treat them (mostly) like we can other objects

- We can give them additional names
- We can pass them as parameters
- We can return them from functions

The consequences

Since functions are objects, we can treat them (mostly) like we can other objects

- We can give them additional names
- We can pass them as parameters
- We can return them from functions

There are a few things we (usually) don't do

The consequences

Since functions are objects, we can treat them (mostly) like we can other objects

- We can give them additional names
- We can pass them as parameters
- We can return them from functions

There are a few things we (usually) don't do

- We don't modify their properties (they always do the same thing)

The consequences

Since functions are objects, we can treat them (mostly) like we can other objects

- We can give them additional names
- We can pass them as parameters
- We can return them from functions

There are a few things we (usually) don't do

- We don't modify their properties (they always do the same thing)
- We don't explicitly create them as we do with, say, lists, `SimpleImages` or `Students`

```
>>> def foo():
        print("Hello!")


>>> foo()
Hello!


>>> copy_of_foo = foo

>>> copy_of_foo()
Hello!
```

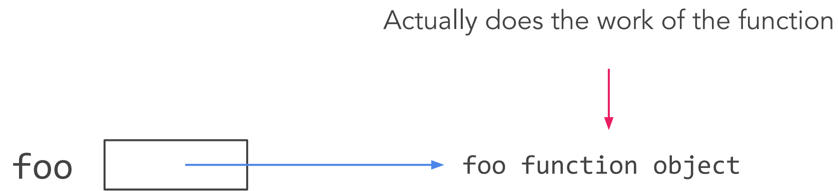
```
>>> def foo():
        print("Hello!")


>>> foo()
Hello!


>>> copy_of_foo = foo

>>> copy_of_foo()
Hello!
```

First, we make a function object named `foo`. This object contains the instructions corresponding to the function body.

```
>>> def foo():
        print("Hello!")


>>> foo()
Hello!


>>> copy_of_foo = foo

>>> copy_of_foo()
Hello!
```

First, we make a function object named `foo`. This object contains the instructions corresponding to the function body.

Next, we call the `foo` function by referring to it by name. As you'd expect, it prints `Hello!` . Under the hood, we're following the arrow from `foo` (the function name) to the function object, and executing the corresponding instructions

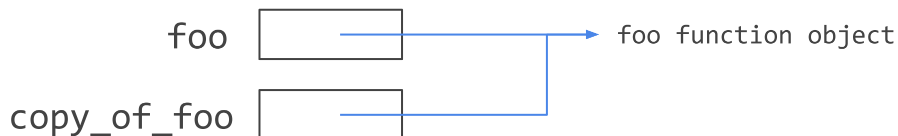


```
>>> def foo():
        print("Hello!")


>>> foo()
Hello!


>>> copy_of_foo = foo

>>> copy_of_foo()
Hello!
```

First, we make a function object named foo. This object contains the instructions corresponding to the function body.

Next, we call the foo function by referring to it by name. As you'd expect, it prints Hello! . Under the hood, we're following the arrow from foo (the function name) to the function object, and executing the corresponding instructions

Next, we make a copy of foo by binding another name to it, specifically copy_of_foo



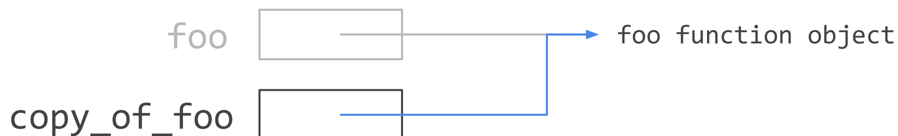
```
>>> def foo():
        print("Hello!")

>>> foo()
Hello!

>>> copy_of_foo = foo

>>> copy_of_foo()
Hello!
```

First, we make a function object named `foo`. This object contains the instructions corresponding to the function body.

Next, we call the `foo` function by referring to it by name. As you'd expect, it prints `Hello!` . Under the hood, we're following the arrow from `foo` (the function name) to the function object, and executing the corresponding instructions

Next, we make a copy of foo by binding another name to it, specifically `copy_of_foo`

Because `copy_of_foo` points at the same instructions as `foo`, it behaves the same way when we call it



Passing functions as parameters

```
def get_len(s):
    return len(s)

def main():
    strs = ['a', 'bbbb', 'cc', 'zzzz']
    strs = sorted(strs, key=get_len)
    print(strs)
```

Passing functions as parameters

```
def get_len(s):
    return len(s)

def main():
    strs = ['a', 'bbbb', 'cc', 'zzzz']
    strs = sorted(strs, key=get_len)
    print(strs)
```

After reading through all the functions, the execution frame for the program looks like this:

Execution frame for program

main → main function object

get_len → get_len function object

```python
def get_len(s):
    return len(s)

def main():
    strs = ['a', 'bbbb', 'cc', 'zzzz']
    strs = sorted(strs, key=get_len)
    print(strs)
```

The key parameter in **sorted** points towards the **get_len** function object, and so **sorted** can call that function object with the name **key**

Execution frame for program

Stack frame for sorted()

key

main function object

get_len function object

```
def get_len(s):
    return len(s)

def main():
    strs = ['a', 'bbbb', 'cc', 'zzzz']
    strs = sorted(strs, key=get_len)
    print(strs)
```

This is actually how `sorted` works!

```c
static PyObject *
list_sort_impl(PyListObject *self, PyObject *keyfunc, int reverse)
/*[clinic end generated code: output=57b9f9c5e23fbe42 input=cb56cd179a713060]*/
{
    MergeState ms;
    Py_ssize_t nremaining;
    Py_ssize_t minrun;
    sortslice lo;
    Py_ssize_t saved_ob_size, saved_allocated;
    PyObject **saved_ob_item;
    PyObject **final_ob_item;
    PyObject *result = NULL;            /* guilty until proved innocent */
```

This is actually how `sorted` works!

```
r (i = 0; i < saved_ob_size ; i++) {
  keys[i] = PyObject_CallOneArg(keyfunc, saved_ob_item[i]);
  if (keys[i] == NULL) {
```

A crucial detail

By passing in `get_len` as parameter to `sorted`, you allow `sorted` to control when it's called

A crucial detail

By passing in `get_len` as parameter to `sorted`, you allow `sorted` to control when it's called

You just need to make sure that `get_len` accepts the correct parameters, but sorted will take care of actually calling the function

Event-Based Programming

Most applications you use today are event-based

Event-Based Programming

Most applications you use today are event-based

Rather than proceeding through a predetermined series of actions, they wait for user input (the event) and react accordingly

Event-Based Programming

Most applications you use today are event-based

Rather than proceeding through a predetermined series of actions, they wait for user input (the event) and react accordingly

> A program might need to react to a mouse click, or a keyboard shortcut, sound or camera input, or even an internet request.

Event-Based Programming

Most applications you use today are event-based

Rather than proceeding through a predetermined series of actions, they wait for user input (the event) and react accordingly

> A program might need to react to a mouse click, or a keyboard shortcut, sound or camera input, or even an internet request.

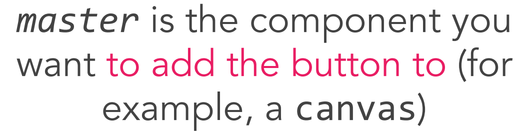
Let's look at how this works

How do you add a button to your program?

```
button = tkinter.Button(
    master,
    text="Text on button",
    command=function
)
button.pack()
```

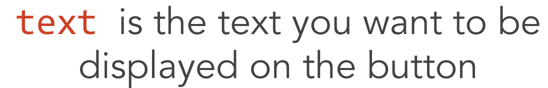
How do you add a button to your program?

```
button = tkinter.Button(

    master,

    text="Text on button",

    command=function

)

button.pack()
```



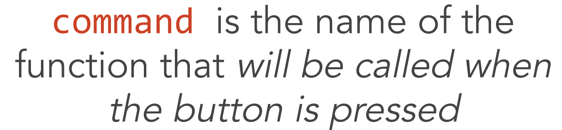
How do you add a button to your program?

```
button = tkinter.Button(
    master,
    text=“Text on button”,
    command=function
)
button.pack()
```



How do you add a button to your program?

```
button = tkinter.Button(
    master,
    text="Text on button",
    command=function
)
button.pack()
```



How do you add a button to your program?

```
button = tkinter.Button(
    master,
    text="Text on button",
    command=function
)
button.pack()
```

Demo: My First Button

What's the limitation here?

The `click_handler` function doesn't have a memory of your program's current state, and we don't control which parameters are passed into it

What's the limitation here?

The `click_handler` function doesn't have a memory of your program's current state, and we don't control which parameters are passed into it

This means it will do the same thing every time it is called

What's the limitation here?

The `click_handler` function doesn't have a memory of your program's current state, and we don't control which parameters are passed into it

This means it will do the same thing every time it is called

This makes for some pretty boring buttons

What’s the limitation here?

The `click_handler` function doesn’t have a memory of your program’s current state, and we don’t control which parameters are passed into it

This means it will do the same thing every time it is called

This makes for some pretty boring buttons

What ways do we have of encoding state into a program?

What’s the limitation here?

The `click_handler` function doesn’t have a memory of your program’s current state, and we don’t control which parameters are passed into it

This means it will do the same thing every time it is called

This makes for some pretty boring buttons

What ways do we have of encoding state into a program?

- Global variables

What’s the limitation here?

The `click_handler` function doesn’t have a memory of your program’s current state, and we don’t control which parameters are passed into it

This means it will do the same thing every time it is called

This makes for some pretty boring buttons

What ways do we have of encoding state into a program?

Global variables 😱

What’s the limitation here?

The `click_handler` function doesn’t have a memory of your program’s current state, and we don’t control which parameters are passed into it

This means it will do the same thing every time it is called

This makes for some pretty boring buttons

What ways do we have of encoding state into a program?

- ~~Global variables~~
- Writing a class

What's the limitation here?

The `click_handler` function doesn't have a memory of your program's current state, and we don't control which parameters are passed into it

This means it will do the same thing every time it is called

This makes for some pretty boring buttons

What ways do we have of encoding state into a program?

- ~~Global variables~~
- Writing a class ✔️

Class Methods work the same way!

Thus far, we've been talking only about functions defined outside of a class

Methods inside classes work exactly the same way

In the `Student` class from last week's lecture, the student class has methods whose names are `self.get_name`, `self.get_ID`, `self.get_units`, `self.set_units`, `self.increment_units` and `self.can_graduate`, each of which is bound to a function object

This means that within a class definition, you can pass its methods as parameters

Let’s take this for a spin

Some final thoughts

Growth happens when you question the abstract.

Overflow slides: adding other UI components

Introduction

I’ll post a video reviewing these concepts in greater detail, but the following slides should serve as a useful reference if you’re looking to include UI elements such as text entry fields or clickable objects in your program.

Note that all of these concepts will need to be used in tandem with classes, and won’t be very helpful otherwise. These slides don’t include the code for integrating them with classes, but that will look similar to today’s lecture example.

Feel free to post on Ed if you have questions about any of this!

Adding a text entry field

```
entry = tkinter.Entry(master)

entry.pack()



# elsewhere

text_in_entry = entry.get()
```

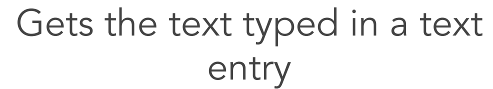
master is the component you want to add the entry to (for example, a `canvas`)

Adding a text entry field

```
entry = tkinter.Entry(master)

entry.pack()


# elsewhere

text_in_entry = entry.get()
```

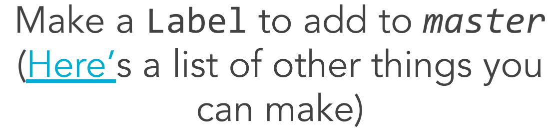
Adding a text entry field

```
entry = tkinter.Entry(master)

entry.pack()



# elsewhere

text_in_entry = entry.get()
```



Reacting to a mouse click

```
label = tkinter.Label(
    master,
    text="Label text"
)
label.bind("<1>", label_click_handler)
label.pack()
```



Reacting to a mouse click

```
label = tkinter.Label(
    master,
    text="Label text"
)
label.bind("<1>", label_click_handler)
label.pack()
```

"<1>" represents a left click (Here's a list of the other events you can react to)

Reacting to a mouse click

```
label = tkinter.Label(
    master,
    text="Label text"
)
label.bind("<1>", label_click_handler)
label.pack()
```

label_click_handler is called when label is left-clicked

Reacting to a mouse click

```
label = tkinter.Label(
    master,
    text=“Label text”
)
label.bind(“<1>”, label_click_handler)
label.pack()
```

This line of code binds `label_click_handler` to the `left-click event`

Click Handlers

```
def label_click_handler(event):

    # click handling code here
```

An event handler passed into the .bind function must accept an **event** object as a parameter, which contains information about the event that took place

Click Handlers

```
def label_click_handler(event):

    # click handling code here
```

See this page for more details about events and how to work with them