Where Do We Write Python Code?

sarah — zsh

```
>>> 10 + 20 + 30
60
```

script.py

```
total = 10 + 20 + 30
print(total)
```

The Python Interactive Shell

The Python shell let's you run Python lines of code one at a time

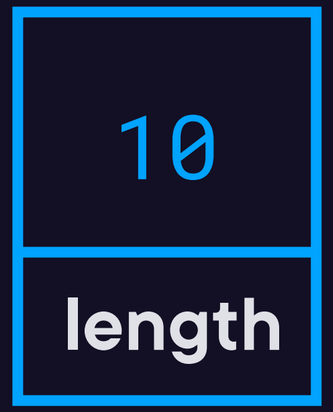
A Python File

A Python script or file is where you create longer Python programs

Saving Numbers to Variables

Assigning the value `10` *to the variable* `length`

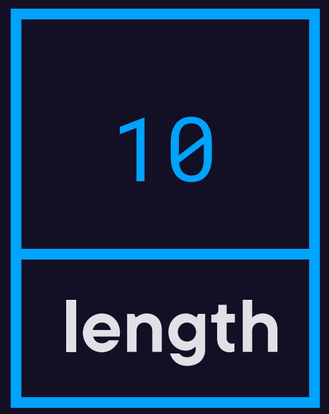
```
>>> length = 10
```



Now on your computer there is a piece of memory labeled `length` *that stores the value* `10`

Saving Numbers to Variables

```
>>> length = 10
>>> length
10
```



From the shell we can enter the name of the variable `length` *to see it's value and see that it's actually* `10`

Saving Numbers to Variables

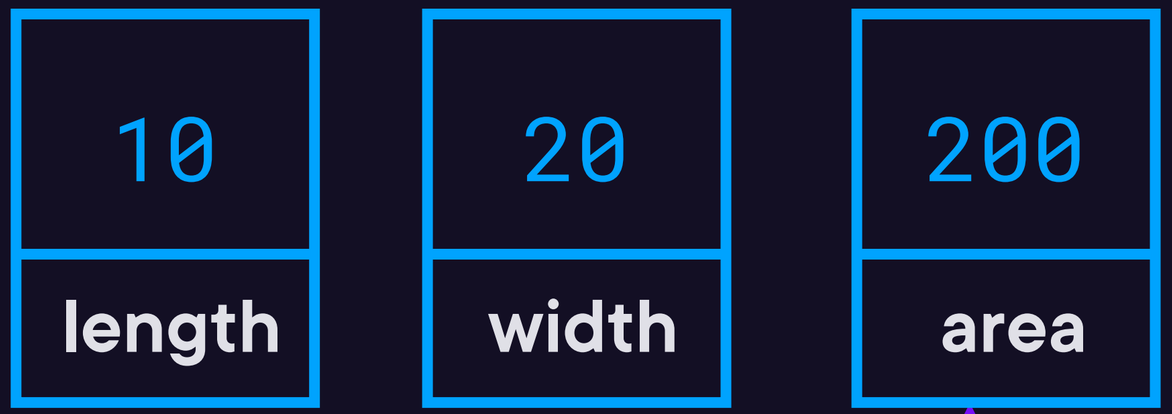
```
>>> length = 10
>>> width = 20
```

Let's also add the `width` *of the rectangle*

Now we have another variable stored in memory

Saving Numbers to Variables

```
>>> length = 10
>>> width = 20
>>> area = length * width
```



Now we can calculate the area with the multiplication operator

And now we have another variable stored in memory

The arithmetic operators in Python are mostly the same ones you know already from a calculator: `+` `-` `*` `/`

Saving Numbers to Variables

```
>>> length = 10
>>> width = 20
>>> area = length * width
>>> area
200
```

10	20	200
length	width	area

The value of `area` *is output to the screen*

Primitive Data Types

Python assumes the type of variable based on the assigned value

```
>>> amount = 10
```

int

Python infers that `amount` **is an** `int` **since it is a whole number**

```
>>> amount = 10.50
```

float

Python infers that `amount` **is a** `float` **since it is a decimal**

A Python Script

sales_tax.py

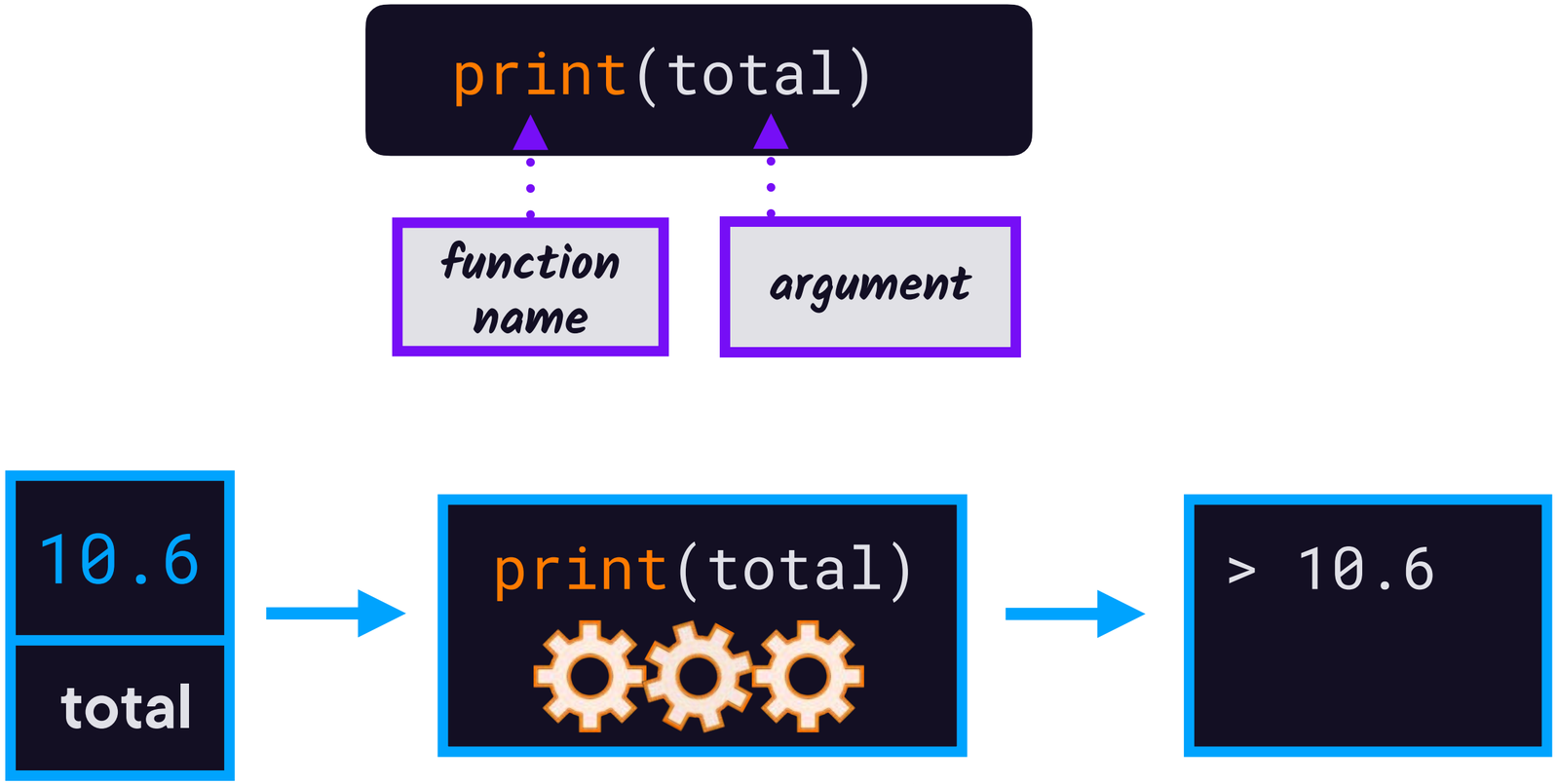
```
amount = 10
tax = .06
total = amount + amount*tax
print(total)
```

We can call the `print()` *function to output* `total`

```
> python3 sales_tax.py
10.6
```

Now the value of `total` *is printed to the screen*

Python `print()` Function



You can think of this function as a black box machine. We don't know how it works...

But we do know it will output the given value to the screen.

Data Type Conversion Functions

What if we want to convert a `float` to an `int`?

```
>>> amount = int(10.6)
>>> amount
10
```

`int()`

Use the `int()` conversion function

What if we want to convert an `int` to a `float`?

```
>>> amount = float(10)
>>> amount
10.0
```

`float()`

Use the `float()` conversion function

A `String` Stores Text

greeting.py

```
name = 'Sarah'

print(name)
```

Creating a `String` *with single quotes*

The string `'Sarah'` *is saved to the variable* `name`

```
> python3 greeting.py
Sarah
```

The value of `name` *prints without quotes*

The quotes are only used to tell Python that anything inside them is a `String`.

Create Strings with Single or Double Quotes

greeting.py

```
store_name = "Sarah's Store"
print(store_name)
```

Double quotes are useful if a single quote is literally part of the `String`

```
store_name = 'Sarah's Store'
print(store_name)
```

This would cause an error because the second single quote would end the `String` *and Python doesn't know what to do with the rest.*

String Concatenation

greeting.py

```
hello = "Hello"
name = "Sarah"
greeting = hello + name
print(greeting)
```

Concatenate two Strings *with a* +

```
> python3 greeting.py
HelloSarah
```

Notice how the two strings are smushed together? We need a space between them.

Fixing Our Program

greeting.py

```
hello = "Hello"
name = "Sarah"
greeting = hello + " " + name
print(greeting)
```

Concatenate a space

```
> python3 greeting.py
Hello Sarah
```

Fixed

Fixing Our Program

greeting.py

```
hello = "Hello"
name = "Sarah"
greeting = hello + " " + name
print(greeting)
```

Let's ask the user for their name.

```
> python3 greeting.py
Hello Sarah
```

How can we customize this program for other names?

Python input() Function

```
>>> my_name = input("What's your name?")
```

function name

The argument is a message

The string the user types in is then saved to the variable

```
> What's your name?
Alice
```

The message gets printed to the screen

The program waits for the user to input something and press enter

Console Input

greeting.py

```
hello = "Hello"
name = input(“What's your name?”)
greeting = hello + " " + name
print(greeting)
```

`input()` *prints the statement, then waits for a value from the console*

```
> python3 greeting.py
What’s your name?Bob
Hello Bob
```

Notice how the name Bob is now printed inside of the greeting.

Console Input

greeting.py

```
hello = "Hello"
name = input(“What's your name?”)
greeting = hello + " " + name
print(greeting)
```

```
> python3 greeting.py
What’s your name?Bob
Hello Bob
```

This looks bad. Can we enter the name on the next line?

Console Input

greeting.py

```
hello = "Hello"
name = input(“What's your name?\n”)
greeting = hello + " " + name
print(greeting)
```

\n *is a special character for a new line*

```
> python3 greeting.py
What’s your name?
Bob
Hello Bob
```

Now input is entered on the next line.

Summary of Input and Output

```
>>> name = input(“What's your name?\n”)
What’s your name?
Sarah
```

input

```
>>> print(“Hello ” + name + "!!")
Hello Sarah!!
```

output

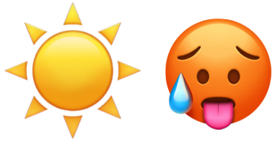
Age Calculator

```
> How old are you?
> 202
> You are 20 decades
  and 2 year(s) old.
```

- Ask the user for input
- Save the input to a variable
- Calculate the decades and years
- Convert these numbers to text
- Print the result to the screen

How Do We Make Decisions in a Program?

A conditional statement, or if statement, lets us make decisions in Python



If it's sunny and 90° and higher

Stay inside!

If it's raining

Stay inside!

Otherwise

Go outdoors!

The 6 Python Comparators

<	<=	==	>=	>	!=
less than	less than equal to	equal	greater than equal to	greater than	NOT equal

Assigning 95 *to the* temp *variable*

```
>>> temp = 95
>>> temp == 95
True
```

Making a comparison is like asking the question:

Is the temp *equal to* 95*?*

Notice: the assignment is 1 = sign

And the equals to comparator is 2 == signs

The 6 Python Comparators

<	<=	==	>=	>	!=
less than	less than equal to	equal	greater than equal to	greater than	NOT equal

Is the temperature less than 90?

```
>>> temp = 95
>>> temp == 95
True

>>> temp < 90
False
```

An `if` statement

Lets us *decide* what to do: if *True*, then *do* this.

weather.py

```
temperature = 95

if temperature > 80:
```

Assign 95 *to the* temperature *variable*

Is the temperature greater than 80*?*

If this is True*, let's add something to do here*

An `if` statement

Lets us *decide* what to do: if *True*, then *do* this.

weather.py

```
temperature = 95

if temperature > 80:
  print("It's too hot!")
  print("Stay inside!")
```

This is True

So these lines are run

```
> python3 weather.py
It's too hot!
Stay inside!
```

`if` Code Block

weather.py

```
temperature = 95

if temperature > 80:
  print("It's too hot!")
  print("Stay inside!")
```

Any indented code that comes after an if statement is called a code block

```
> python3 weather.py
It's too hot!
Stay inside!
```

When the `if` statement is `False`

weather.py

```
temperature = 75

if temperature > 80:
  print("It's too hot!")
  print("Stay inside!")
```

This is `False`

So these lines are NOT run

```
> python3 weather.py
```

And there is no output

The Program Continues After the `if` Code Block

weather.py

```
temperature = 75

if temperature > 80:
  print("It's too hot!")
  print("Stay inside!")
print("Have a good day!")
```

This is False

The program keeps running after the if statement and its code block, so this is printed after.

```
> python3 weather.py
Have a good day!
```

Rules for Whitespace in Python

weather.py

```
temperature = 75

if temperature > 80:
  print("It's too hot!")
    print("Stay inside!")
```

2 space indent

4 space indent

```
> python3 weather.py
File "weather.py", line 6
    print("Stay inside!")
    ^
IndentationError: unexpected indent
```

Whitespace indents in Python need to be consistent, otherwise there will be an IndentationError.

An `if`, `else` statement

weather.py

```
temperature = 75

if temperature > 80:
  print("It's too hot!")
  print("Stay inside!")
```

This is `False`

How do we do something else here if this is `False`*?*

An `if`, `else` statement

weather.py

```
temperature = 75

if temperature > 80:
  print("It's too hot!")
  print("Stay inside!")
else:
  print("Enjoy the outdoors!")
```

If this statement is `False`*, then run the code block below*

Otherwise, then run this code block

```
> python3 weather.py
Enjoy the outdoors!
```

if, elif, and else

weather.py

```
temperature = 50

if temperature > 80:
  print("It's too hot!")
  print("Stay inside!")
elif temperature < 60:
  print("It's too cold!")
  print("Stay inside!")
else:
  print("Enjoy the outdoors!")
```

False

True

So both of these lines are run.

```
> python3 weather.py
It's too cold!
Stay inside!
```

Can We Combine Two `if` Statements?

weather.py

```
temperature = 75

if temperature > 80:
  print("Stay inside!")
elif temperature < 60:
  print("Stay inside!")
else:
  print("Enjoy the outdoors!")
```

Let's shorten our program to only say: "Stay inside!" OR "Enjoy the outdoors!"

We're repeating `print("Stay inside!")`

Can we combine the first 2 `if` *statements?*

Logical Operator - or

weather.py

```
temperature = 75


if temperature > 80 or temperature < 60:
  print("Stay inside!")
else:
  print("Enjoy the outdoors!")
```

The keyword or *lets you combine multiple comparisons.*

At least one needs to be True *for the whole if statement to be* True

Logical Operator - `or`

Only *one* comparison needs to be *True* for the if statement to be *True*

weather.py

```
temperature = 75

if temperature > 80 or temperature < 60:
  print("Stay inside!")
else:
  print("Enjoy the outdoors!")
```

False or False → False

This is run

```
> python3 weather.py
Enjoy the outdoors!
```

Logical Operator - `or`

Only *one* comparison needs to be *True* for the if statement to be *True*

weather.py

```
temperature = 50

if temperature > 80 or temperature < 60:
  print("Stay inside!")
else:
  print("Enjoy the outdoors!")
```

False or True → True

This is run

```
> python3 weather.py
Stay inside!
```

Store the Forecast as a `String`

weather.py

```
temperature = 75
forecast = "rainy"
```

Let's add another variable with the forecast as "rainy", "cloudy", or "sunny".

Logical Operator - and

Both comparisons need to be `True` for the if statement to be `True`

weather.py

```
temperature = 75
forecast = "rainy"


if temperature < 80 and forecast != "rain":
    print("Go outside!")
else:
    print("Stay inside!")
```

Logical Operator - and

Both comparisons need to be `True` for the if statement to be `True`

weather.py

```
temperature = 75
forecast = "rainy"

True   and   False → False
if temperature < 80 and forecast != "rain":
    print("Go outside!")
else:
    print("Stay inside!")  ◄·· This is run
```

Logical Operator - and

Both comparisons need to be `True` for the if statement to be `True`

weather.py

```
temperature = 75
forecast = "sunny"

if temperature < 80 and forecast != "rain":
    print("Go outside!")
else:
    print("Stay inside!")
```

True and True → True

This is run

Logical Operator - `not`

The keyword `not` lets you negate a comparison. And can help make the statement more readable.

weather.py

```
forecast = "rainy"


if not forecast == "rainy"
    print("Go outside!")
else:
    print("Stay inside!")
```

Logical Operator - not

weather.py

```
forecast = "rainy"

if not forecast == "rainy"
    print("Go outside!")
else:
    print("Stay inside!")
```

not True → False

Negate means make the opposite:
not True → False,
not False → True

This is run

The 3 Python Logical Operators

and

or

not

The keywords and and or let you combine multiple comparisons

The keyword not lets you negate a comparison

Evaluating Boolean Variables

weather.py

```
raining = True

if raining:
    print("Stay inside!")
```

You can set boolean variables to either True *or* False

This reads more like English

```
> python3 weather.py
Stay inside!
```

Evaluating Boolean Variables

weather.py

```
raining = True

if not raining:
    print("Go outside!")
else:
    print("Stay inside!")
```

not True → False

This is run

```
> python3 weather.py
Stay inside!
```

A List is a Container of Things

```
empty = []
```

empty list

```
words = ['LOL', 'IDK', 'TBH']
```

list of strings

```
nums = [5, 10, 15]
```

list of numbers

```
mixed = [5, 'SDK', 1.5]
```

list of mixed items

```
lists = [ ['A', 'B', 'C'], ['D', 'E', 'F'] ]
```

list of lists

Creating a List of Internet Slang Acronyms

```
acronyms = ['LOL', 'IDK', 'SMH', 'TBH']
```

We're compiling a list of acronyms that we'll define later

An Item's Index is its Position

```
acronyms = ['LOL', 'IDK', 'SMH', 'TBH']
```

0 1 2 3

```
print(acronyms[0])
```

1st item

```
> LOL
```

An Item's Index is its Position

```
acronyms = ['LOL', 'IDK', 'SMH', 'TBH']
```

0 1 2 3

```
print(acronyms[3])
```

4th item

```
> TBH
```

Note: if you want the nth *item then use index* [n-1]

Creating a List and Adding Items

```
acronyms = []

acronyms.append('LOL')
acronyms.append('IDK')

print(acronyms)
```

We can create an empty list

And then add each item individually

```
> ['LOL', 'IDK']
```

We can see the 2 items in the list

Creating a List with Items and Then Adding Items

```
acronyms = ['LOL', 'IDK', 'SMH']

acronyms.append('BFN')
acronyms.append('IMHO')

print(acronyms)
```

We haven't called a method like this before.
Let's take a closer look...

```
> ['LOL', 'IDK','SMH', 'BFN','IMHO']
```

Calling a Method

```
acronyms.append('BFN')
```

The list you want to change

dot

method name

What you want to add to the list in parentheses ( )

Removing Items

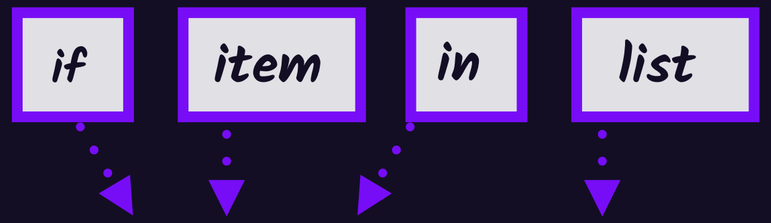
```
acronyms = ['LOL', 'IDK', 'SMH', 'TBH', 'BFN']

acronyms.remove('BFN')  OR  del acronyms[4]

print(acronyms)
```

You can use either `remove` *or* `del` *depending on whether you know the value or the index.*

```
> ['LOL', 'IDK', 'SMH', 'TBH']
```

You can see `'BFN'` *was removed*

Check if Exists in List




```
if 1 in [1, 2, 3, 4, 5]:
    print('True')
```

```
> True
```

Check if Exists in List

```
acronyms = ['LOL', 'IDK', 'SMH', 'TBH']
word = ‘BFN'

if word in acronyms:
    print(word + ' is in the list')
else:
    print(word + ' is NOT in the list')
```

False

So this line is run

```
> BFN is NOT in the list
```

Printing a List

```
acronyms = ['LOL', 'IDK', 'SMH', 'TBH']

print(acronyms)
```

```
> ['LOL', 'IDK', 'SMH', 'TBH']
```

What if we want to print each acronym on a separate line?

We need a loop

The Syntax of a for loop

for

Loop variable

in

list

```
for acronym in acronyms:
    print(acronym)
```

acronym *is a temporary variable that holds one of the* acronyms *in the list for each run*

Like saying "do this" for each string acronym *in our* acronyms *list*

For Loop: Looping Over Each Item in a List

```
acronyms = ['LOL', 'IDK', 'SMH', 'TBH']

for acronym in acronyms:
    print(acronym)
```

```
> LOL
  IDK
  SMH
  TBH
```

1st loop
2nd loop
3rd loop
4th loop

For Loop: Looping Over Each Item in a List

```
acronyms = ['LOL', 'IDK', 'SMH', 'TBH']

for acronym in acronyms:
    print(acronym)
```

Notice how the code block you want repeated inside the loop is indented, just like in an if statement.

```
> LOL
  IDK
  SMH
  TBH
```

1st loop
2nd loop
3rd loop
4th loop

Adding Input to the Expenses Calculator

expenses.py

We want the user to be able to enter their own expenses

```
expenses = [10.50, 8.50, 5.30, 15.05, 20.00, 5.00, 3.00]
total = sum(expenses)

print("You spent $", total, " on lunch this week.", sep='')
```

Adding Input to the Expenses Calculator

expenses.py

```
expenses = []
expenses.append(float(input("Enter an expense:\n")))
expenses.append(float(input("Enter an expense:\n")))
expenses.append(float(input("Enter an expense:\n")))
expenses.append(float(input("Enter an expense:\n")))
expenses.append(float(input("Enter an expense:\n")))
expenses.append(float(input("Enter an expense:\n")))
expenses.append(float(input("Enter an expense:\n")))
...
```

With our current set of tools, we would type input 7 times.

Is there a way we can loop 7 times instead and ask for input inside the loop?

The `range()` Function

```
>>> range(7)
```

```
(0, 1, 2, 3, 4, 5, 6)
```

```
>>> range(0, 7, 1)
```

```
(0, 1, 2, 3, 4, 5, 6)
```

Start

Stop

Step

Note: Start and Step are optional. Start is 0 by default, Step is 1.

Note: The sequence starts at 0 and ends at 6, but there are 7 numbers.

The `range()` Function

```
>>> range(7)
```

```
(0, 1, 2, 3, 4, 5, 6)
```

```
>>> range(0, 7, 1)
```

```
(0, 1, 2, 3, 4, 5, 6)
```

```
>>> range(2, 14, 2)
```

```
(2, 4, 6, 8, 10, 12)
```

Start

Stop

Step

We get even numbers starting at 2 and stopping before 14

The Syntax of a for loop

```
for i in range(7):
    print(i)
```

```
> 0
  1
  2
  3
  4
  5
  6
```

We can then use a for loop like we've seen with the sequence generated by `range()`.

This let's us loop a certain number of times, which is what needed to enter expenses...

Adding Input to the Expenses Calculator

expenses.py

```
total = 0
expenses = []
for i in range(7):
    expenses.append(float(input("Enter an expense:")))

total = sum(expenses)

print("You spent $", total, sep='')
```

Adding Input to the Expenses Calculator

expenses.py

```
total = 0
expenses = []
for i in range(7):
    expenses.append(float(input("Enter an expense:")))

total = sum(expenses)

print("You spent $", total, sep='')
```

```
> python3 expenses.py
Enter an expense:10
Enter an expense:5
Enter an expense:20
Enter an expense:12
Enter an expense:13
Enter an expense:8
Enter an expense:4
You spent $72
```

Adding Input to the Expenses Calculator

expenses.py

```
total = 0
expenses = []
for i in range(7):
    expenses.append(float(input("Enter an expense:")))

total = sum(expenses)

print("You spent $", total, sep='')
```

What if we want the user to enter the number of expenses?

```
> python3 expenses.py
Enter an expense:10
Enter an expense:5
Enter an expense:20
Enter an expense:12
Enter an expense:13
Enter an expense:8
Enter an expense:4
You spent $72
```

Adding Input to the Expenses Calculator

expenses.py

```
total = 0
expenses = []
num_expenses = int(input("Enter # of expenses:"))
```

Adding Input to the Expenses Calculator

expenses.py

```python
total = 0
expenses = []
num_expenses = int(input("Enter # of expenses:"))
for i in range(num_expenses):
    expenses.append(float(input("Enter an expense:")))

total = sum(expenses)

print("You spent $", total, sep='')
```

```
> python3 expenses.py
Enter # of expenses:5
Enter an expense:5
Enter an expense:20
Enter an expense:12
Enter an expense:13
Enter an expense:8
You spent $58
```

Maintaining Two Lists

```
acronyms =     ['LOL',                'IDK',              'TBH']
translations = ['laugh out loud', "I don't know", 'to be honest’]

del acronyms[0]
del translations[0]

print(acronyms)
print(translations)
```

If we add or delete from one list ...
We have to do the same thing in the other list.

```
>  ['IDK', 'TBH']
   ["I don't know", 'to be honest']
```

A Dictionary Maps Keys to Values

```
acronyms = {'LOL': 'laugh out loud',
            'IDK': "I don't know",
            'TBH': 'to be honest'}
```

key | assign to | value

Key	Value
'LOL'	'laugh out loud'
'IDK'	"I don't know"
'TBH'	to be honest’

These would be the keys and values stored in the dictionary.

Each item is known as a "key-value pair"

A Dictionary Maps Keys to Values

```
acronyms = {'LOL': 'laugh out loud',
            'IDK': "I don't know",
            'TBH': 'to be honest'}

print(acronyms['LOL'])
```

To look up a value in a dictionary, we send in a key. (Instead of an index in a list.)

```
> laugh out loud
```

Dictionaries Can Hold Anything

Dictionary of strings to strings

```
acronyms = {'LOL': 'laugh out loud', 'IDK': "I don't know"}
```

Dictionary of strings to numbers

```
menu = {'Soup': 5, 'Salad': 6}
```

A menu item's name is they key, and its price is the value.

Dictionary of anything

```
my_dict = {10: 'hello', 2: 6.5}
```

Creating a Dictionary and Adding Values

```
acronyms = {}

acronyms['LOL'] = 'laugh out loud'
acronyms['IDK'] = "I don't know"
acronyms['TBH'] = 'to be honest'

print(acronyms)
```

Create an empty dictionary

Adding new dictionary items

```
>  {'IDK': "I don't know", 'LOL': 'laugh out loud',
    'TBH': 'to be honest'}
```

Notice our 3 key-value pairs are there, but order is random in a dictionary.

Updating Values in Our Dictionary

```
acronyms = {'LOL': 'laugh out loud',
            'IDK': "I don't know",
            'TBH': 'to be honest'}

acronyms['TBH'] = 'honestly'

print(acronyms['TBH'])
```

We can also create a dictionary with initial values

A value is update the same way a value is added.

Looking up the same value

```
>  honestly
```

The value is printed

Removing Dictionary Items

```
acronyms = {'LOL': 'laugh out loud',
            'IDK': "I don't know",
            'TBH': 'to be honest'}

del acronyms['LOL']

print(acronyms)
```

To delete a value in a dictionary, we send in a key just like when we look up a value.

```
>  {'IDK': "I don't know", 'TBH': 'to be honest'}
```

Getting an Item That's NOT in the Dictionary

```
acronyms = {'LOL': 'laugh out loud',
            'IDK': "I don't know",
            'TBH': 'to be honest'}

definition = acronyms['BTW']
```

Trying to access a key that doesn't exist will cause an error

```
> KeyError: 'BTW'
```

Getting an Item That's NOT in the Dictionary

```
acronyms = {'LOL': 'laugh out loud',
            'IDK': "I don't know",
            'TBH': 'to be honest'}

definition = acronyms.get('BTW')

print(acronyms)
```

Using `get()` *won't crash your program with an error.*

Instead, `get()` *will return* None *if the key doesn't exist*

```
> None
```

None *is a type that represents the absence of a value.*

None **Type**

None **means the absence of a value, and values to** `False` **in a conditional**

```
acronyms = {'LOL': 'laugh out loud', 'IDK': "I don't know"}
definition = acronyms.get('BTW')
```

Definition equals None ***because the key doesn't exist***

```
if definition:
    print(definition)
else:
    print("Key doesn't exist")
```

False ***because definition is*** None

So this is run

```
> Key doesn't exist
```

Using a Dictionary to Translate a Sentence

```
acronyms = {'LOL': 'laugh out loud',
            'IDK': "I don't know",
            'TBH': 'to be honest’}

sentence = 'IDK' + ' what happened ' + 'TBH'
translation = acronyms.get('IDK') +' what happened '+ acronyms.get('TBH')
```

Look up the value of each acronym in the acronyms *dictionary*

Using a Dictionary to Translate a Sentence

```
acronyms = {'LOL': 'laugh out loud',
            'IDK': "I don't know",
            'TBH': 'to be honest’}

sentence = 'IDK' + ' what happened ' + 'TBH'
translation = acronyms.get('IDK') +' what happened '+ acronyms.get('TBH')

print('sentence:', sentence)
print('translation:', translation)
```

```
> sentence: IDK what happened TBH
  translation: I don't know what happened to be honest
```

A Dictionary of Lists

We could also use a dictionary for our menus with keys for Breakfast, Lunch, and Dinner

```
menus = { 'Breakfast' : ['Egg Sandwich', 'Bagel', 'Coffee'],
          'Lunch'     : ['BLT', 'PB&J', 'Turkey Sandwich'],
          'Dinner'    : ['Soup', 'Salad', 'Spaghetti', 'Taco'] }

print('Breakfast Menu:\t', menus['Breakfast'])
print('Lunch Menu:\t', menus['Lunch'])
print('Dinner Menu:\t', menus['Dinner'])
```

Using the keys to access each list

```
> Breakfast Menu:  ['Egg Sandwich', 'Bagel', 'Coffee']
  Lunch Menu:      ['BLT', 'PB&J', 'Turkey Sandwich']
  Dinner Menu:     ['Soup', 'Salad', 'Spaghetti', 'Taco']
```

A Dictionary of Lists

We could also use a dictionary for our menus with keys for Breakfast, Lunch, and Dinner

```
menus = { 'Breakfast' : ['Egg Sandwich', 'Bagel', 'Coffee'],
          'Lunch'     : ['BLT', 'PB&J', 'Turkey Sandwich'],
          'Dinner'    : ['Soup', 'Salad', 'Spaghetti', 'Taco'] }

print('Breakfast Menu:\t', menus['Breakfast'])
print('Lunch Menu:\t', menus['Lunch'])
print('Dinner Menu:\t', menus['Dinner'])
```

What if we had a lot more menus, is there a better way to print each one?

```
> Breakfast Menu:  ['Egg Sandwich', 'Bagel', 'Coffee']
  Lunch Menu:      ['BLT', 'PB&J', 'Turkey Sandwich']
  Dinner Menu:     ['Soup', 'Salad', 'Spaghetti', 'Taco']
```

Printing the Dictionary Menu Items

```
menus = { 'Breakfast' : ['Egg Sandwich', 'Bagel', 'Coffee'],
          'Lunch'     : ['BLT', 'PB&J', 'Turkey Sandwich'],
          'Dinner'    : ['Soup', 'Salad', 'Spaghetti', 'Taco'] }

for item in menus:
    print(item)
```

This defaults to just returning the keys in a dictionary... Which is <u>not</u> what we want.

```
> Breakfast
  Lunch
  Dinner
```

Using a Dictionary's Key and Value in a `for` Loop

```
menus = { 'Breakfast' : ['Egg Sandwich', 'Bagel', 'Coffee'],
          'Lunch'     : ['BLT', 'PB&J', 'Turkey Sandwich'],
          'Dinner'    : ['Soup', 'Salad', 'Spaghetti', 'Taco'] }

for name, menu in menus.items():
    print(name, ':', menu)
```

Now the loop has access to both the key and the value here.

```
> Breakfast:  ['Egg Sandwich', 'Bagel', 'Coffee']
  Lunch:      ['BLT', 'PB&J', 'Turkey Sandwich']
  Dinner:     ['Soup', 'Salad', 'Spaghetti', 'Taco']
```

Using Dictionaries to Represent Objects

Let's say we have a person and we want to represent their attributes, such as their name, age, and city they're from.

```
person = {'name': 'Sarah Smith',
          'city': "Orlando",
          'age': '100'}

print(person.get('name'), 'is', person.get('age'), 'years old.')
```

We could use a dictionary where the attributes are saved as key, value pairs.

```
> Sarah Smith is 100 years old.
```

Functions We've Used

Functions are like mini-programs that complete a specific task.

```
print('Hello World')
```

`print()` *takes in one string (or multiple strings) and prints them to the console*

Functions We've Used

Functions are like mini-programs that complete a specific task.

```
print('Hello World')

name = input('Enter your name:\n')
```

`input()` *prompts the user for input and returns the string they entered.*

Functions We've Used

Functions are like mini-programs that complete a specific task.

```
print('Hello World')

name = input('Enter your name:\n')

amount = int(10.6)
```

`int()` converts the given number to an integer.

Functions We've Used

Functions are like mini-programs that complete a specific task.

```
print('Hello World')

name = input('Enter your name:\n')

amount = int(10.6)

roll = random.randint(1,6)
```

`randint()` *takes in a low and high bound and returns a random integer within that range.*

Functions We've Used

Functions are like mini-programs that complete a specific task.

```
print('Hello World')

name = input('Enter your name:\n')

amount = int(10.6)

roll = random.randint(1,6)
```

We can define a function to do anything we want and once we do we can use it over and over again.

Defining a Function

We want a simple function that defines a greeting for a given name.

def keyword

Function name

0 to many parameters

```
def greeting(name):
    print('Hello', name)
```

The function body is indented below the definition.

Defining a Function

greetings.py

```
def greeting(name):
    print('Hello', name)
```

The function definition.

```
# Main program
input_name = input('Enter your name:\n')

greeting(input_name)
```

The program starts running here. This is called the main body of the program.

Order Matters

greetings.py

```
def greeting(name):
    print('Hello', name)
```

The functions need to be defined first...

```
# Main program
input_name = input('Enter your name:\n')

greeting(input_name)
```

Before they are called.

Flow Through a Program

greetings.py

```
def greeting(name):
    print('Hello', name)


# Main program
input_name = input('Enter your name:\n')

greeting(input_name)
```

```
> python3 greetings.py
Enter your name:
Sarah
```

The 1st line of code that isn't in a function definition is where the program starts.

Flow Through a Program

greetings.py

```
def greeting(name):
    print('Hello', name)


# Main program
input_name = input('Enter your name:\n')

greeting(input_name)
```

Call the greeting() function

```
> python3 greetings.py
Enter your name:
Sarah
```

Flow Through a Program

greetings.py

```
def greeting(name):
    print('Hello', name)


# Main program
input_name = input('Enter your name:\n')

greeting(input_name)
```

Enter the function, name has the value of input_name which is "Sarah".

```
> python3 greetings.py
Enter your name:
Sarah
```

Flow Through a Program

greetings.py

```
def greeting(name):
    print('Hello', name)

# Main program
input_name = input('Enter your name:\n')

greeting(input_name)
```

Prints "Hello Sarah"

```
> python3 greetings.py
Enter your name:
Sarah
Hello Sarah
```

Flow Through a Program

greetings.py

```
def greeting(name):
    print('Hello', name)


# Main program
input_name = input('Enter your name:\n')

greeting(input_name)
```

End of the program

```
> python3 greetings.py
Enter your name:
Sarah
Hello Sarah
```

Scope

Local scope: variable created inside a function can only be used inside that function

greetings.py

```
def greeting(name):
    print('Hello', name)


# Main program
input_name = input('Enter your name:\n')

greeting(input_name)
print(name)
```

The variable name only exists inside this function where it was defined.

The variable name doesn't exist here, outside of the function, so this would give us an error.

Global Scope

A variable created in main body of the program is a ***global*** variable and has ***global*** scope. That means it can be used anywhere.

greetings.py

```
def greeting():
    print('Hello', name)
```

The variable name is global so we can reference it inside this function.

```
# Main program
name = input('Enter your name:\n')

greeting()
```

The variable name is global.

We don't need a parameter for greeting() since it can reference the global variable name

Global Scope

greetings.py

```
def greeting():
    print('Hello', name)

# Main program
name = input('Enter your name:\n')

greeting()
```

```
> python3 greetings.py
Enter your name:
Sarah
Hello Sarah
```

The program using the global `name` *variable works the same as before.*

Global Scope

Using global variables can become messy

greetings.py

```
def greeting():
    print('Hello', name)

# Main program
name = input('Enter your name:\n')
greeting()
name2 = input('Enter your name:\n')
name = name2
greeting()
```

The variable name is global.

Now how do we use the greeting() function with name2?

We could save name2 to the name variable. But then the value for name is gone... Let's try local scope again.

Local Scope

greetings.py

```
def greeting(name):
    print('Hello', name)
```

Now we can use the greeting() with any passed in value for name.

```
# Main program
name1 = input('Enter your name:\n')
greeting(name1)
name2 = input('Enter your name:\n')
greeting(name2)
```

We have two different values and we can use the greeting() function for both of them.

Local Scope

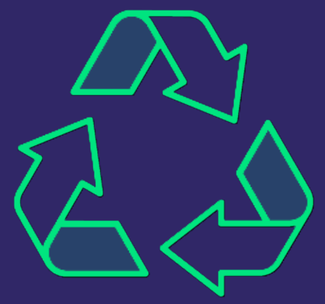
greetings.py

```
def greeting(name):
    print('Hello', name)

# Main program
name1 = input('Enter your name:\n')
greeting(name1)
name2 = input('Enter your name:\n')
greeting(name2)
```

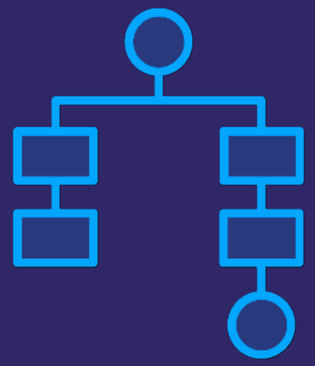
```
> python3 greetings.py
Enter your name:
Sarah
Hello Sarah
Enter another name:
Bob
Hello Bob
```

Local scope allows us to reuse the `greeting()` *function with different values.*

Reasons to Create a Function



You want to reuse a chunk of code over and over.



You want to organize your code by logical units.

Another Example Function

We want a simple function that adds two numbers and returns the result.

def keyword

Function name

2 parameters

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

return keyword

Some functions return a value, or even a sequence of values.

Defining Our Function

addition.py

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

The function definition

```
# Main program
num1 = float(input('Enter your 1st number:\n'))
num2 = float(input('Enter your 2nd number:\n'))

# Calling our function
result = addition(num1, num2)
print('The result is', result)
```

The main program starts running here.

Flow Through the Program

addition.py

```
4  def addition(a, b):
5      return a + b

   # Main program
1  num1 = float(input('Enter your 1st number:\n'))
2  num2 = float(input('Enter your 2nd number:\n'))

   # Calling our function
3  result = addition(num1, num2)
6  print('The result is', result)
7
```

```
> python3 addition.py
Enter your 1st number:
25
Enter your 2nd number:
37
The result is 62
```

Organizing Our Main Code into a Function

addition.py

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


# Main program
num1 = float(input('Enter your 1st number:\n'))
num2 = float(input('Enter your 2nd number:\n’))

# Calling our function
result = addition(num1, num2)
print('The result is', result)
```

Let's move the whole main body of code to its own function.

Organizing Our Main Code into a Function

addition.py

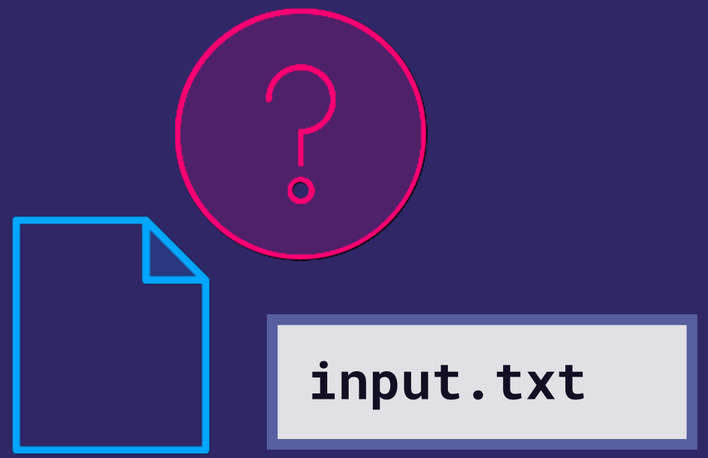
```
def addition(a, b):
    return a + b

def main():
    num1 = float(input('Enter your 1st number:\n'))
    num2 = float(input('Enter your 2nd number:\n'))

    # Calling our function
    result = addition(num1, num2)
    print('The result is', result)
```

Now all of the program is contained inside this main() function.

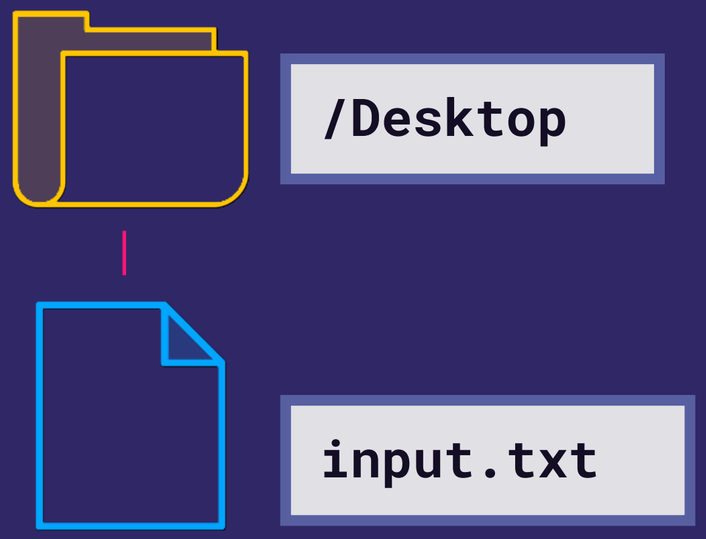
```
main()
```

We need to call main() after the functions are declared.

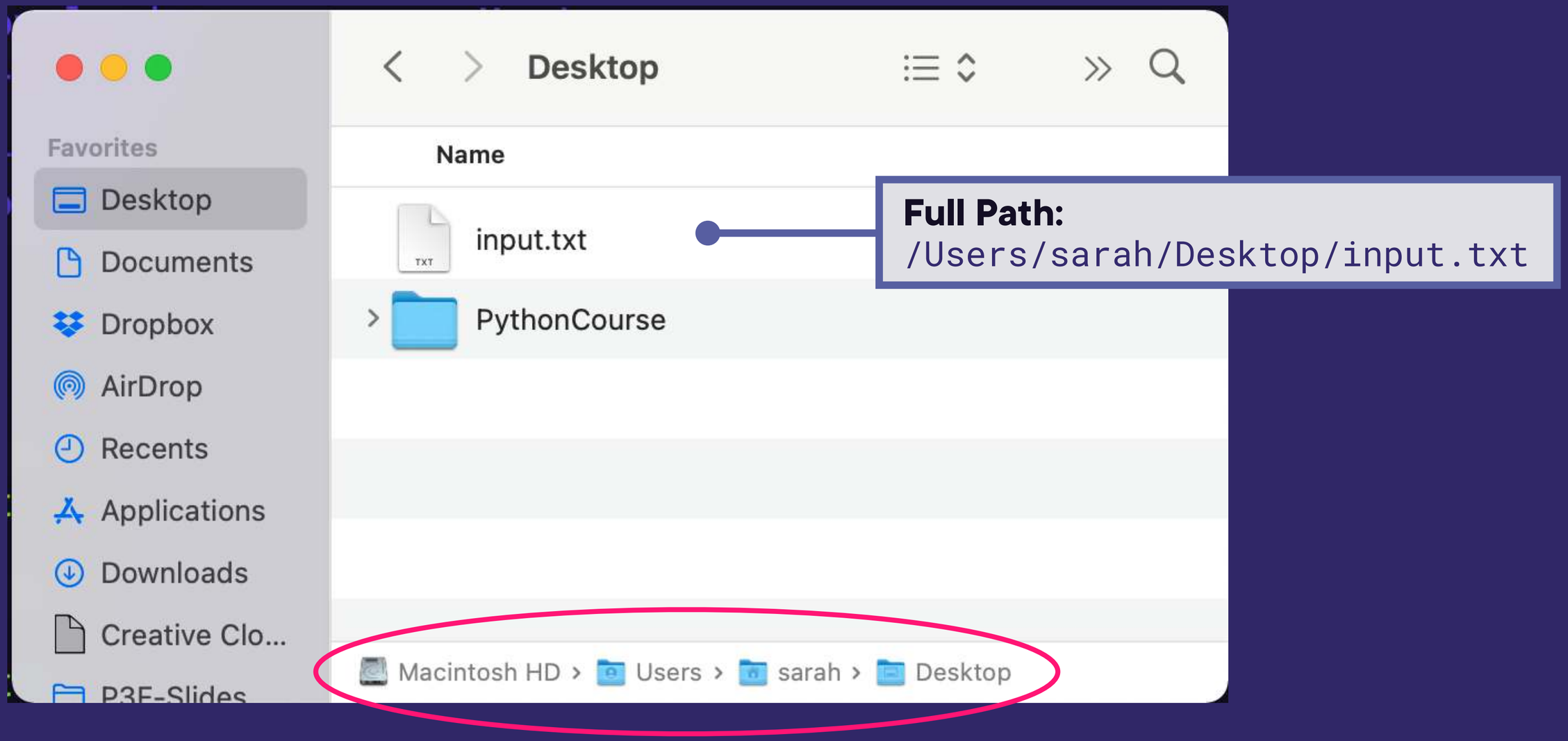
Where is our File? How to Navigate File Paths



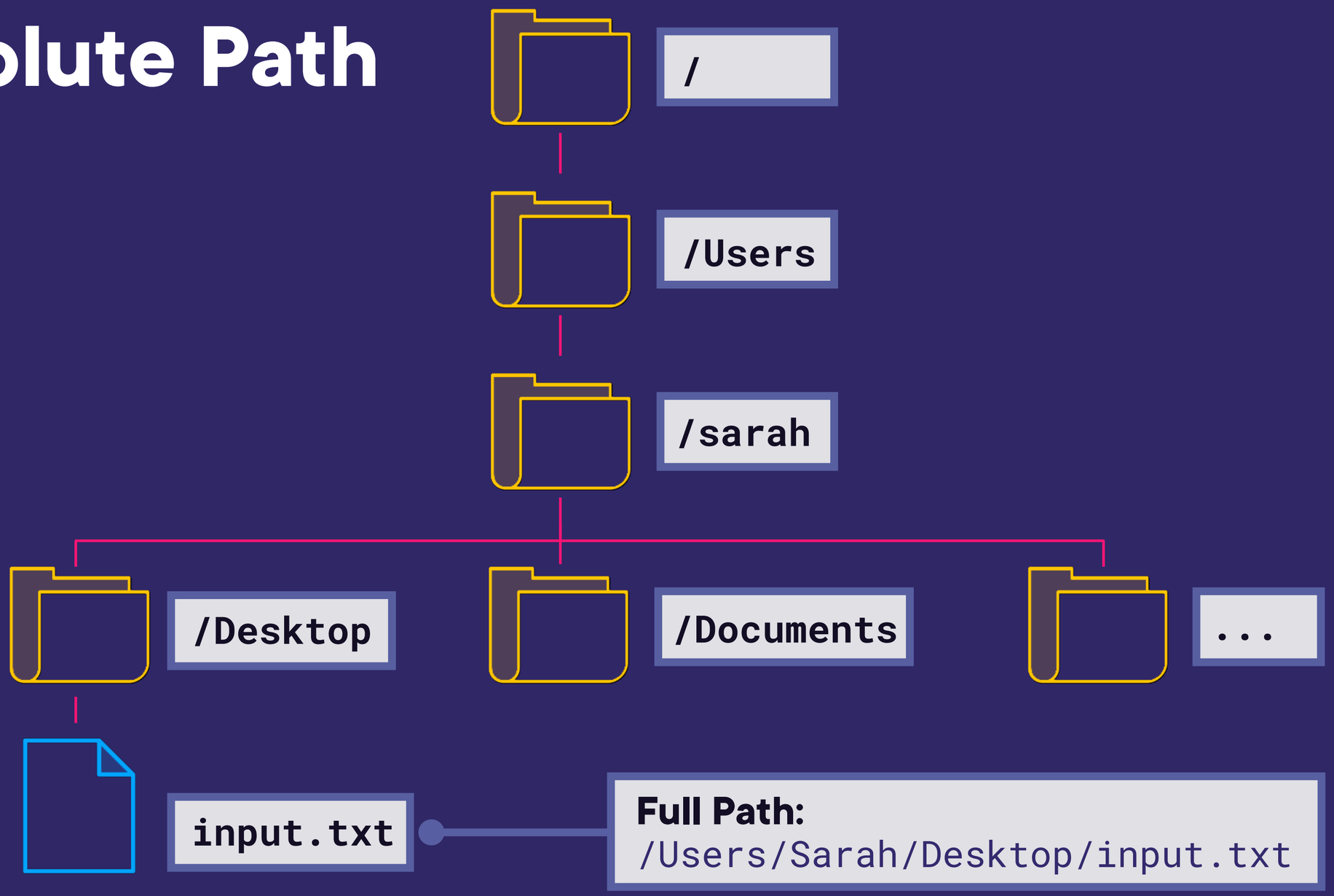
Where is our File? How to Navigate File Paths



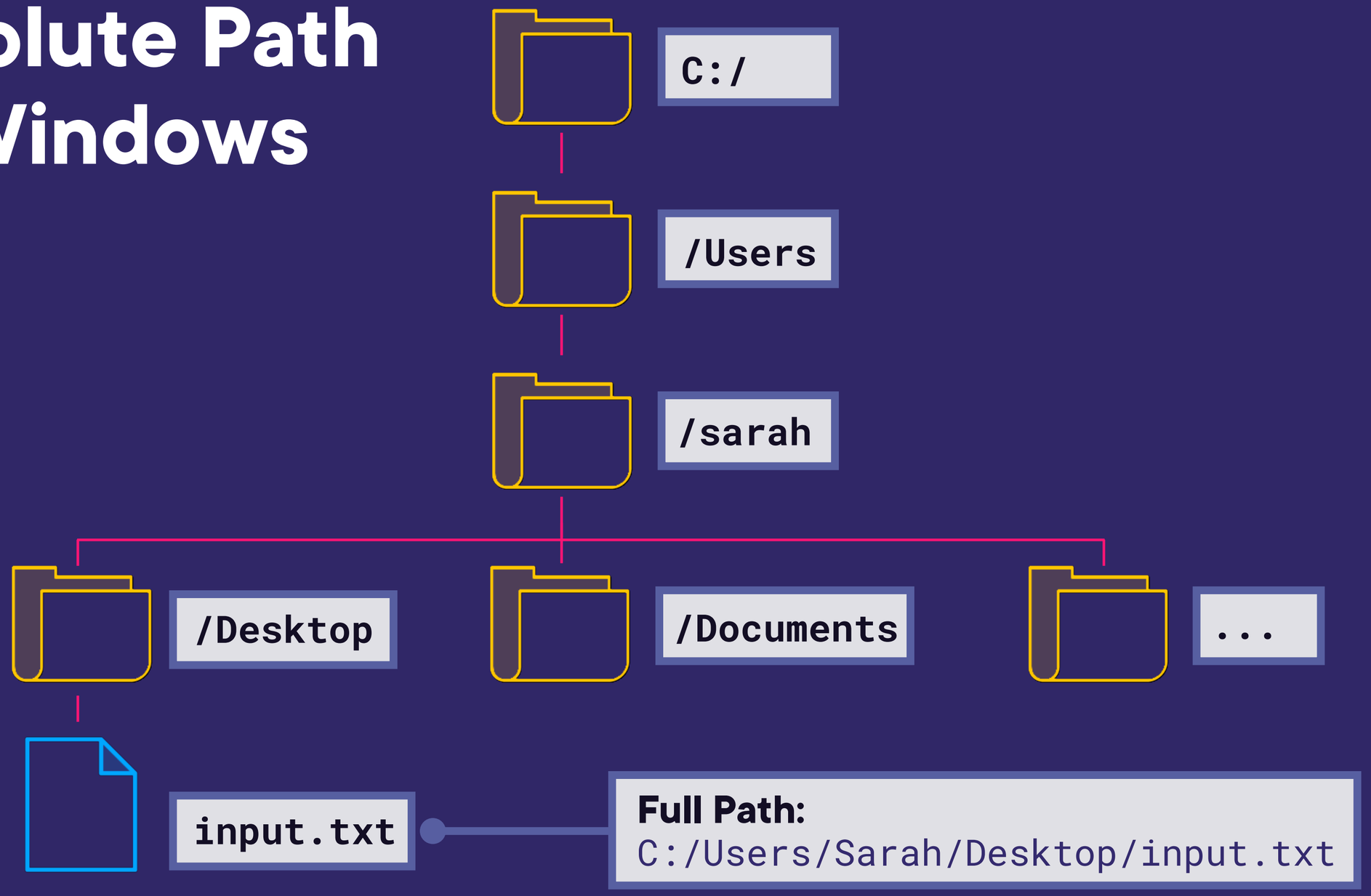
Where am I? How to Navigate File Paths



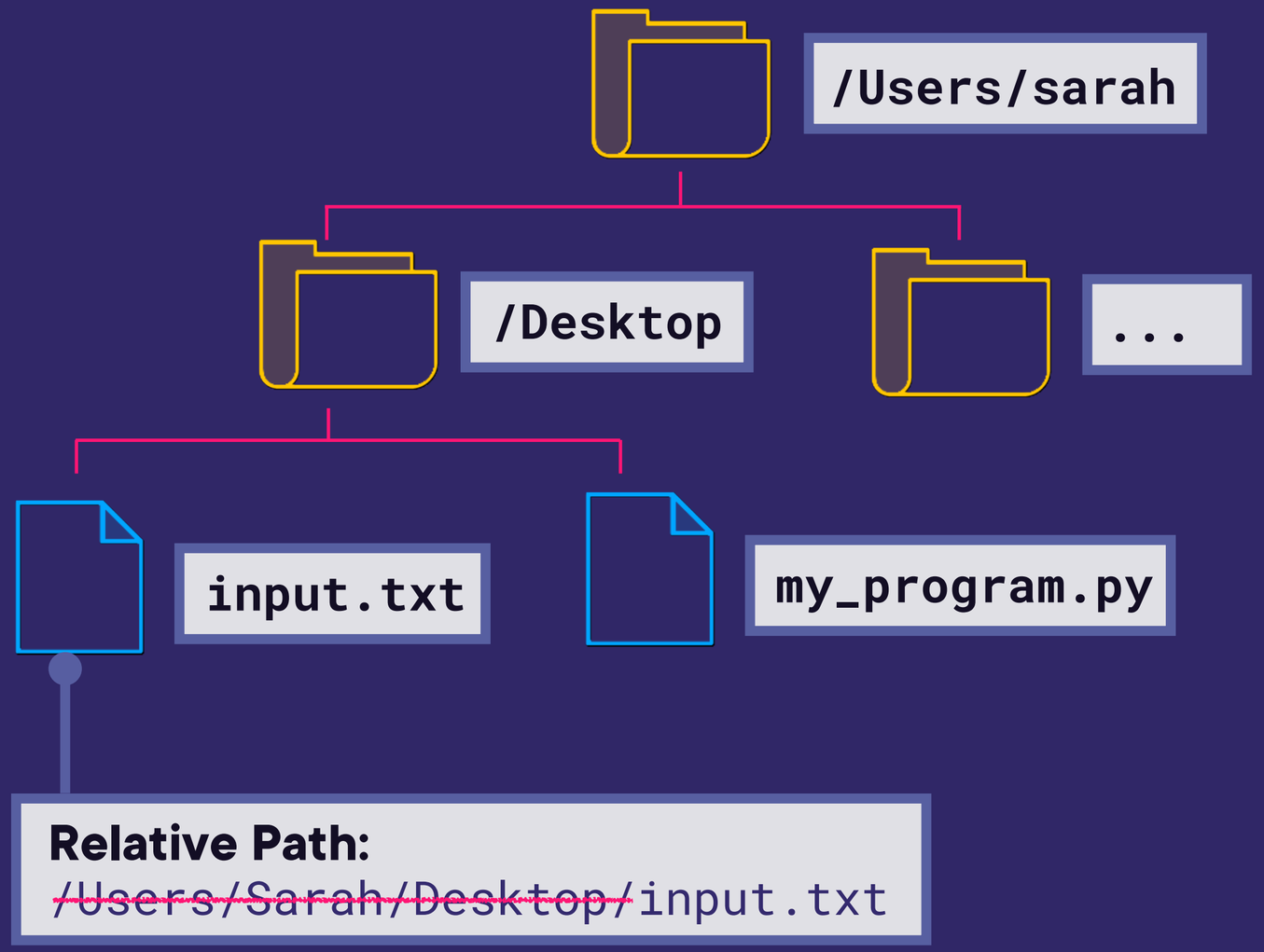
Absolute Path



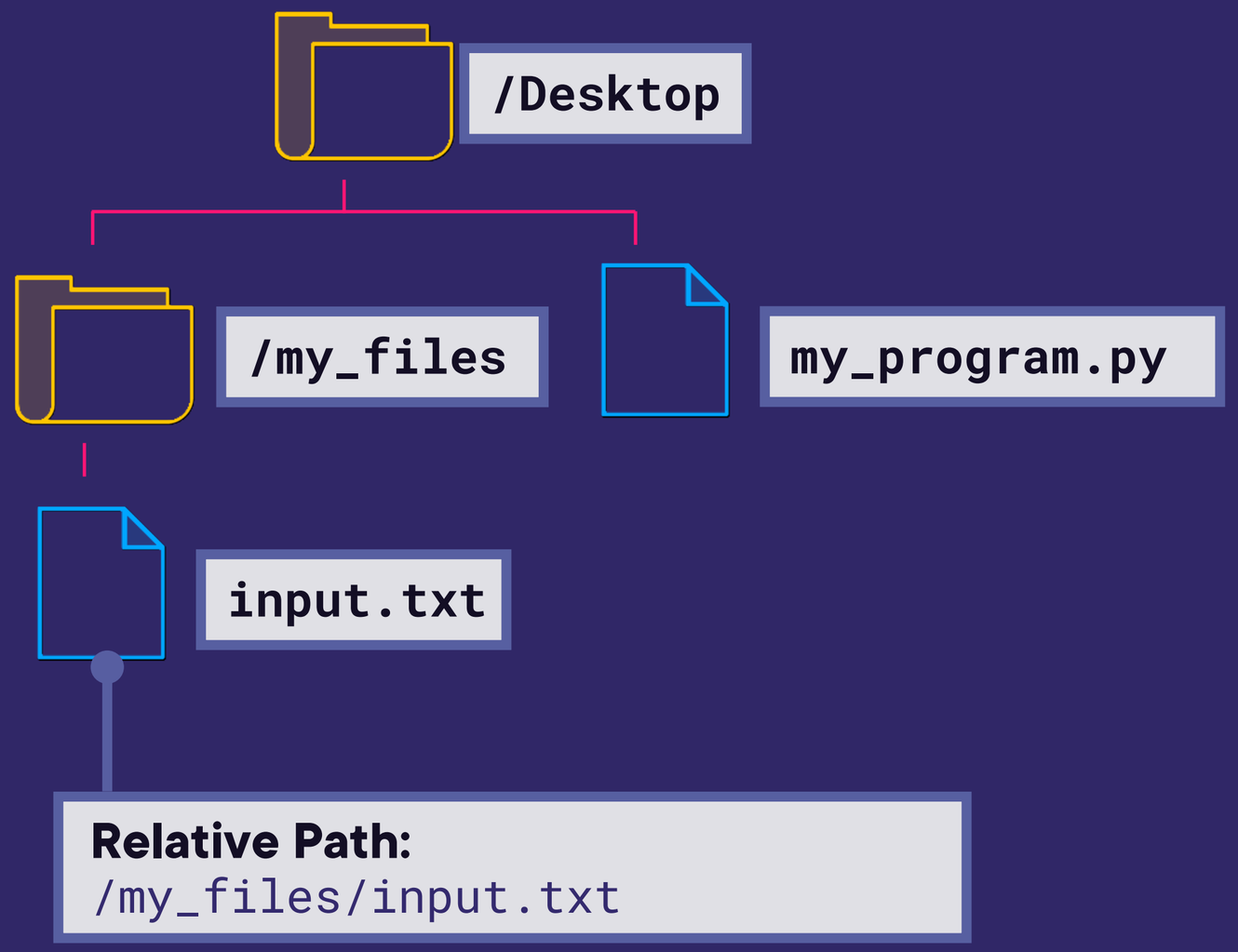
Absolute Path on Windows



Relative Path



Relative Path



Opening a File in Python

acronyms.py

```
file = open('acronyms.txt')
```

`open()` ***returns a*** `File` ***object that has methods like*** `read()` ***and*** `write()`

Opening a File in Python

acronyms.py

```
file = open('acronyms.txt')
```

It's very important to `close()` *a* `File` *object that has been opened...*

Opening a File in Python Using the Keyword `with`

acronyms.py

```
with open('acronyms.txt') as file:
    # Do file operations here
```

The `with` *keyword makes sure the* `File` *is properly closed when the file operations are done even if an exception is raised.*

A Longer Way to Close a File Without the Keyword `with`

acronyms.py

```
file = open('acronyms.txt')
try:
    # Do file operations here
    pass
finally:
    file.close()
```

The `finally` *block makes sure the* `File` *is properly closed when the file operations are done even if an exception is raised.*

Methods for Reading from a File Object — `read()`

acronyms.py

```
with open('acronyms.txt') as file:
    result = file.read()
    print(result)
```

The `read()` *method returns the whole file as a* `String` *by default. Or it will return the specified number of bytes.*

```
> python3 greeting.py

IDE - Integrated Development
Environment
OOP - Object Oriented Programming
UX - User Experience
JSON - JavaScript Object Notation
FIFO - First In First Out
LIFO - Last In First Out
TDD - Test Driven Development
SaaS - Software as a Service
PaaS - Platform as a Service
IaaS - Infastructure as a Service
```

Methods for Reading from a File Object — `readline()`

acronyms.py

```
with open('acronyms.txt') as file:
    result = file.readline()
    print(result)

    result = file.readline()
    print(result)
```

The `readline()` *method returns the next line of the file as a* `String`.

```
> python3 acronyms.py

IDE - Integrated Development
Environment

OOP - Object Oriented
Programming
```

Methods for Reading from a File Object — `readlines()`

acronyms.py

```python
with open('acronyms.txt') as file:
    result = file.readlines()
    for line in result:
        print(line)
```

The `readlines()` *method returns a list of* `Strings` *of all of the lines in the file.*

We can loop over this list and print each line.

```
> python3 acronyms.py

IDE - Integrated Development
Environment

OOP - Object Oriented Programming

UX - User Experience

JSON - JavaScript Object Notation

FIFO - First In First Out
...
```

Using a Loop to Read from a File Object

acronyms.py

```
with open('acronyms.txt') as file:
    result = file.readlines()
    for line in result: file:
        print(line)
```

Since this type of loop is used so often there is a shortcut, we can just loop over the File Object.

Using a Loop to Read from a File Object

acronyms.py

```
with open('acronyms.txt') as file:

    for line in file:
        print(line)
```

Since this type of loop is used so often there is a shortcut, we can just loop over the File Object.

```
> python3 acronyms.py

IDE - Integrated Development
Environment

OOP - Object Oriented Programming

UX - User Experience

JSON - JavaScript Object Notation

FIFO - First In First Out
...
```