Deck 005
Python – Variables

Dr. Hazel “twitch.tv/hazeldotzone” Campbell

What's in a name?

- Names in programming languages are called "identifiers"

What's in an identifier?

1) Starts with a letter: does not start with a digit or symbol
    a) _ underscore is a letter
    b) 名 is a letter
2) Continues with letters and digits
3) Is not a keyword (aka reserved word)
    a) https://docs.python.org/3/reference/lexical_analysis.html#keywords

Identifiers vs Other Stuff

i j k count max int str

username page label

print score info data

battery cookie end

output option color

Keywords

match (sometimes) :
type (sometimes) =

import	try	or
pass	as	yield
None	def	for
break	from	lambda
except	nonlocal	False
in	while	await
raise	assert	else
True	del	
class	global	
finally	not	
is	with	
return	async	
and	elif	
continue	if	

Operators

+=	&=	@=	~	< >	**	.
-=	|=	:=	<<	==	*	@
*=	^=	&	>>	!=	//	
**=	<<=	|	<=	+	/	
/= //=	>>=	^	>=	-	%	
%=						

Delimiters

( ) [ ] { }

, : ! ; = ->

Numbers

10 0b1 0o7 0xf

0 1.414

1_000_000 1e6

6.0221408e23 .1

1e-6 1.1j 2j

1e3j 1e-3j

Avoid reusing builtins!

i j k count max int str

username page label

print score info data

battery cookie end

output option color

Good names should be unique, so when we choose a name, we should avoid names that python is already using.

Use descriptive names!

i j k count max int str

username page label

print score info data

battery cookie end

output option color

Good names should be descriptive, so we should avoid using single-letter names...

Identifiers that start with _

i j k count max int str

username page label

print score info data

battery cookie end

output option color

_private _

Names that start with an _
indicate some kind of privacy.

Except _ being the whole name:
that's an idiom for "unused":
We are use the name to convey
our intention to never use it again

Multi Word Names in Python

- Idiomatic Python uses `snake_case` not `camelCase`
- Words in variable names should be separated by _
- variable names should start with a lower-case letter to indicate that they are variables

- user_name
- word_count
- input_text
- prompt_string
- box_height

Making a Variable

```
1   prompt_string = "Enter a number: "
2
```

In Python, this is a delimiter.

(In other languages it's a binary operator.)

Making a Variable

```
1  prompt_string = "Enter a number: "
2
```

In Python, this is a statement.

Statements begin at the start of the line and end at the end of the line unless we're inside something.

Aside: Statements

```
1   prompt_string = "Enter a number: "
2   print(type(prompt_string))
3   input(prompt_string)
4
```

Statements begin at the start of the line and end at the end of the line unless we're inside something.

Aside: Statements

```
1   prompt_string = "Enter a number: "
2   print(type(prompt_string))
3   input(prompt_string)
4
```

Assignment Statement
Expression Statement
Expression Statement

Statements begin at the start of the line and end at the end of the line unless we're inside something.

Aside: Statements

```
1   prompt_string = "Enter a number: "
2   print(type(prompt_string))
3   input(prompt_string)
4
```

Assignment Statement
Expression Statement
Expression Statement

A statement is an

expression statement

when the whole statement is just a single expression

Aside: Statements

```
1   prompt_string = "Enter a number: "
2   print(type(prompt_string))
3   input(prompt_string)
4
```

Assignment Statement
Expression Statement
Expression Statement

A statement is an

assignment statement

when the statement has two sides delimited by =

Aside: Statements

```
1  prompt_string = "Enter a number: "
2  print(
3      type(
4          prompt_string
5          )
6      )
7  input(prompt_strin
```

Assignment Statement

Expression Statement

This assignment has multiple lines because the line breaks are inside the expressions created by the ( ) expression delimiters

Statements begin at the start of the line and end at the end of the line unless we're inside something.

Aside: Statements

```
1  prompt_string = "Enter a number: "
2  print(
3      type(
4          prompt_string
5          )
6      )
7  input(prompt_string)
```

Statement

Statement

Statement

Statements begin at the start of the line and end at the end of the line unless we're inside something.

Making a Variable

```
1  prompt_string = "Enter a number: "
2
```

In Python, this is an assignment statement.

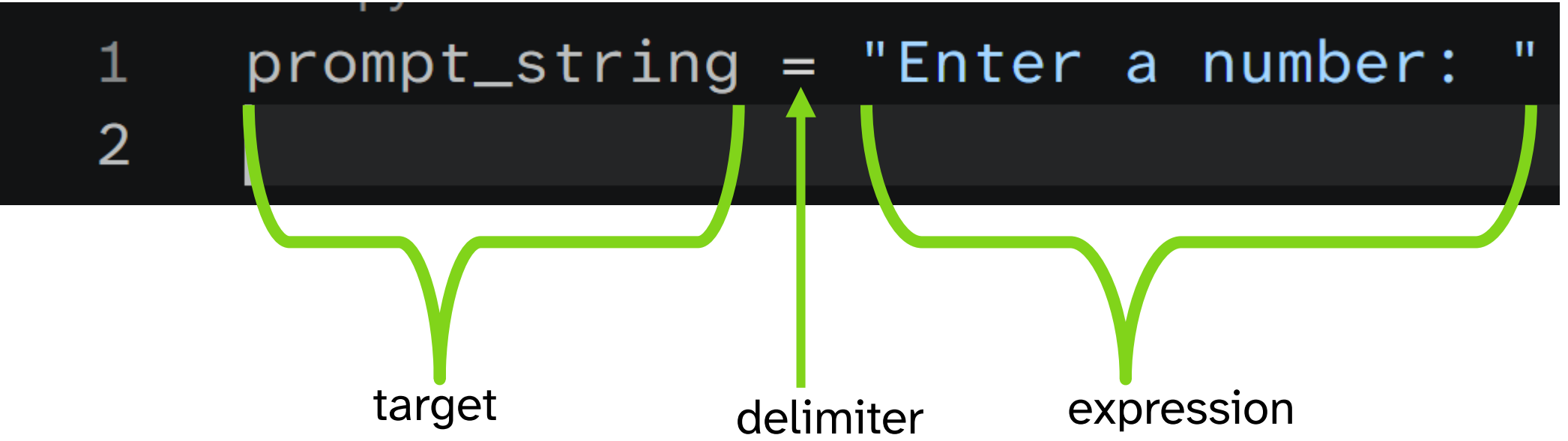
Making a Variable:
The Assignment Statement

```
1   prompt_string = "Enter a number: "
2
```

On the left side of the delimiter is the *target*
or an expression that evaluates to an *lvalue* (short for left value)

On the right side is an expression. It is evaluated to produce an
rvalue
(right value)

Making a Variable:
The Assignment Statement



Making a Variable:
The Assignment Statement

```
1    prompt_string = "Enter a number: "
2
```

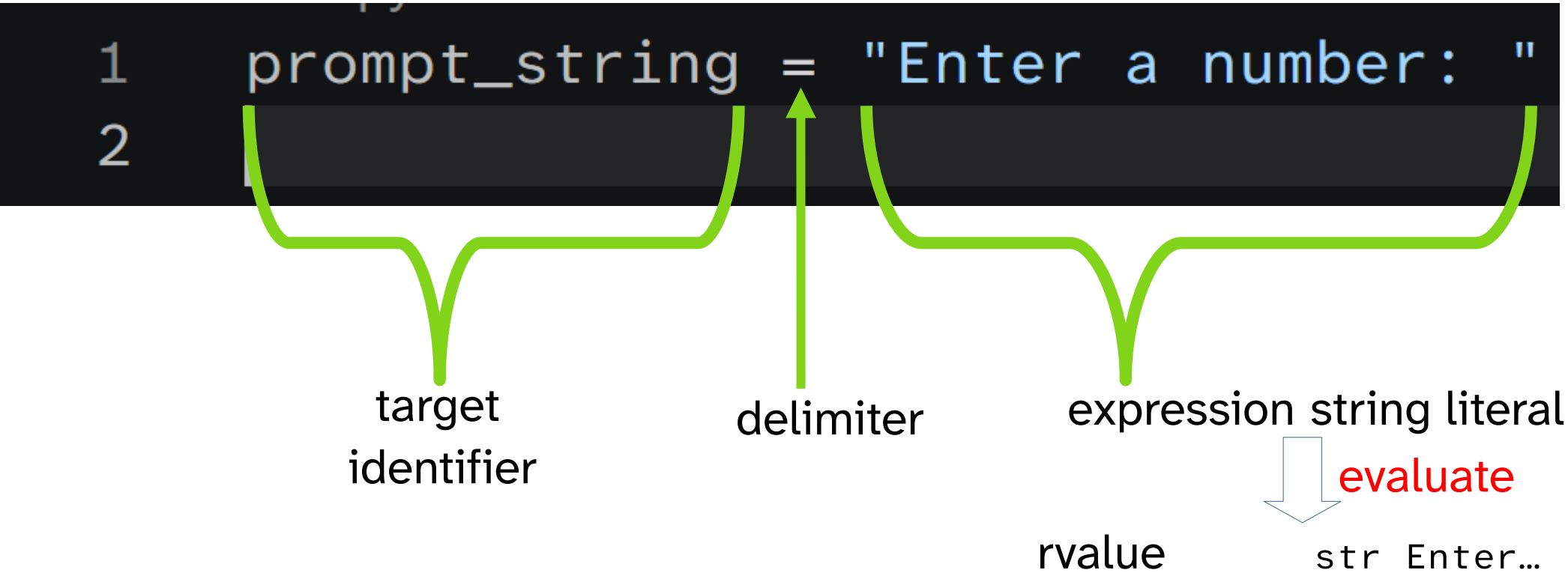
In this example, target is an identifier

delimiter

expression string literal

Making a Variable:
The Assignment Statement

```
1   prompt_string = "Enter a number: "
2
```



Making a Variable:
The Assignment Statement

```
1   prompt_string = "Enter a number: "
2
```

```
10   input("Enter a number: ")
11
```

Let's compare these two statements!

Making a Variable:
The Assignment Statement

```
1   prompt_string = "Enter a number: "
2
```

Assignment statement
(left and right side delimited by =)

```
10    input("Enter a number: ")
11
```

Expression statement
(the whole statement is just
a expression)

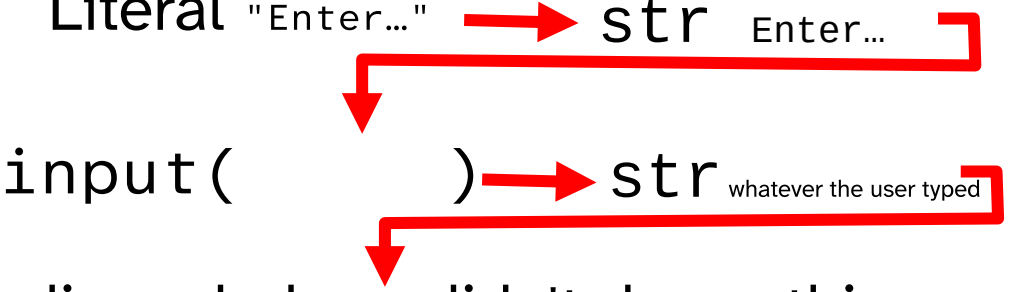
Making a Variable:
The Assignment Statement

```
1   prompt_string = "Enter a number: "
2
```

Assignment statement

```
10   input("Enter a number: ")
11
```

Expression statement



Making a Variable:
The Assignment Statement

```
1   prompt_string = "Enter a number: "
2
```

Assignment statement

Literal `"Enter…"` → `str` Enter…

```
10  input("Enter a number: ")
11
```

Expression statement

Literal `"Enter…"` → `str` Enter…

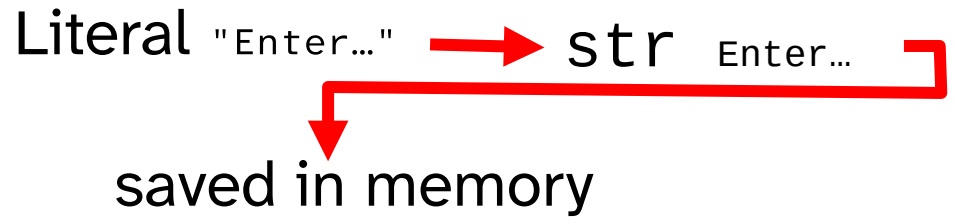
`input(` `)` → `str` whatever the user typed

discarded, we didn't do anything with it

Making a Variable:
The Assignment Statement

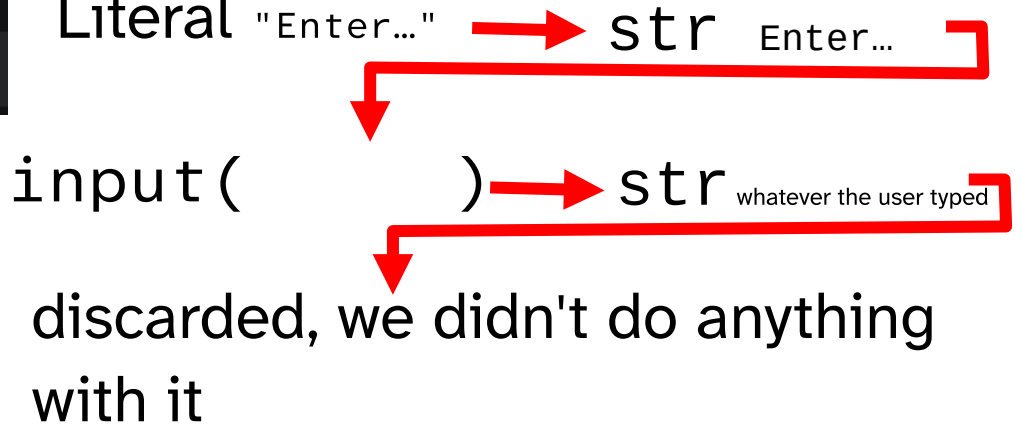
```
1   prompt_string = "Enter a number: "
2
```

Assignment statement




```
10    input("Enter a number: ")
11
```

Expression statement
(the whole statement is just a expression)



Making a Variable:
The Assignment Statement

```
1   prompt_string = "Enter a number: "
2
```

Assignment statement

Literal `"Enter…"` → `str` `Enter…` → saved in memory

1. look if the identifier already exists

2. if it exists rebind it to the rvalue

2. if it doesn't exist, create it and bind it to the rvalue

Making a Variable:
The Assignment Statement

```
1   prompt_string = "Enter a number: "
2
```

Assignment statement

Literal `"Enter…"` → `str` `Enter…`

saved in memory

1. look if the identifier already exists
   `prompt_string`

2. if it exists rebind it to the rvalue

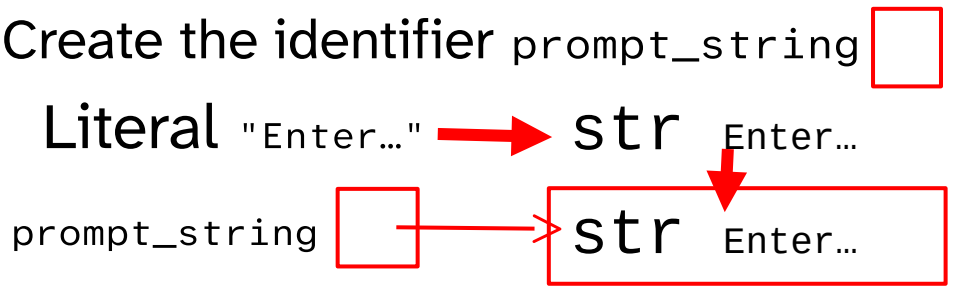
2. if it doesn't exist, create it and bind it to the rvalue

Making a Variable:
The Assignment Statement

```
1   prompt_string = "Enter a number: "
2
```

Assignment statement

assuming the identifier doesn't already exist

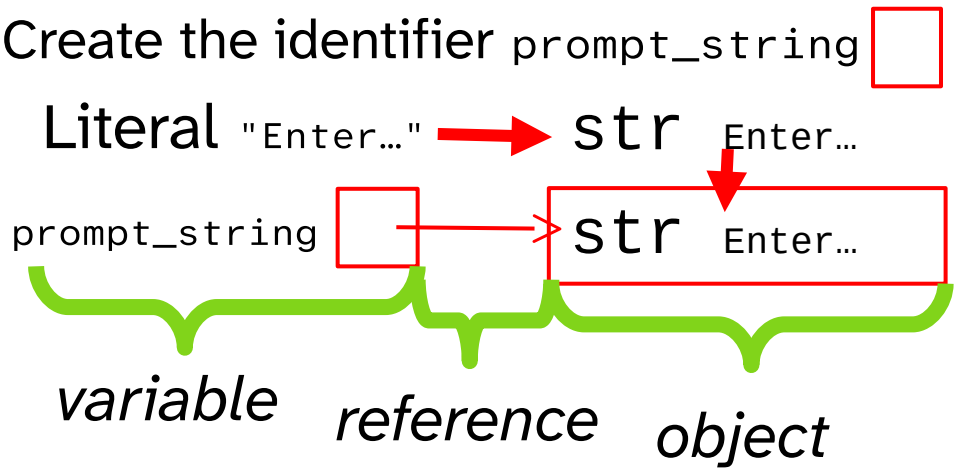


Making a Variable:
The Assignment Statement

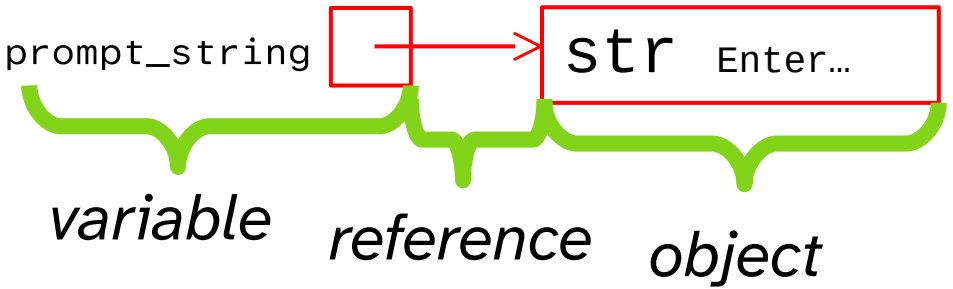
```
1   prompt_string = "Enter a number: "
2
```

Assignment statement

assuming the identifier doesn't already exist



Box Diagram



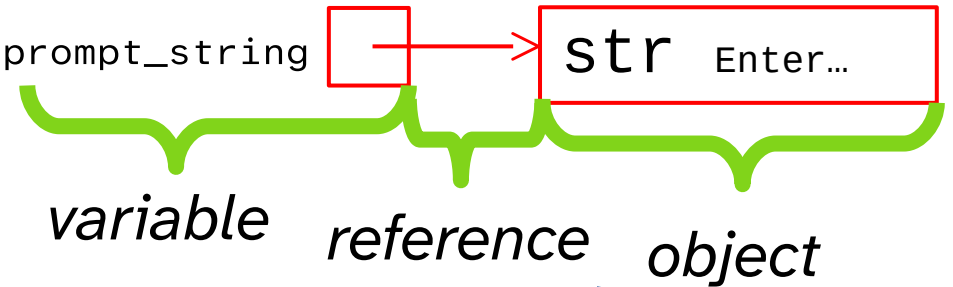
Variables look like this

References look like this

Objects look like this

(note that the arrow comes from inside the box)

Box Diagram



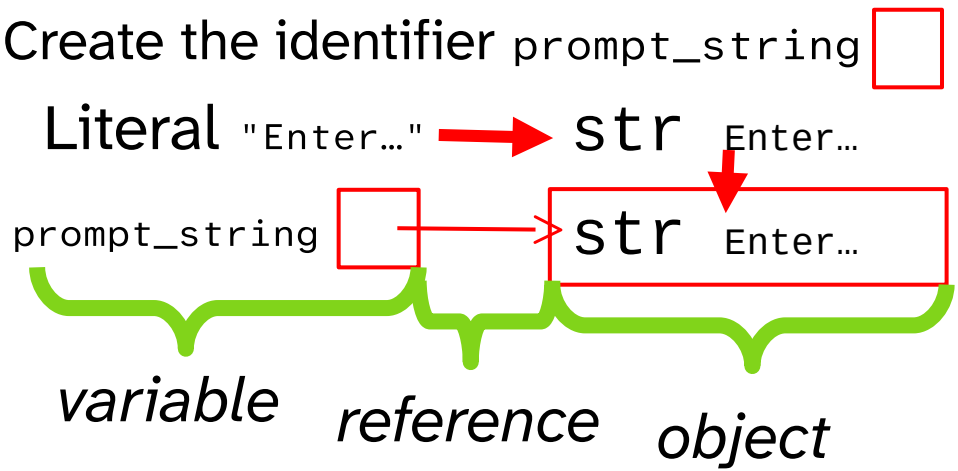
Read "prompt string refers to the string Enter a number"

Making a Variable: The Assignment Statement

```
1   prompt_string = "Enter a number: "
2
```

Read "let prompt string refer to the string Enter a number"

assuming the identifier doesn't already exist



Making a Variable:
The Assignment Statement

```
1   prompt_string = "Enter a number: "
2
```

Read "let prompt string refer to the str Enter a number"

hello.py

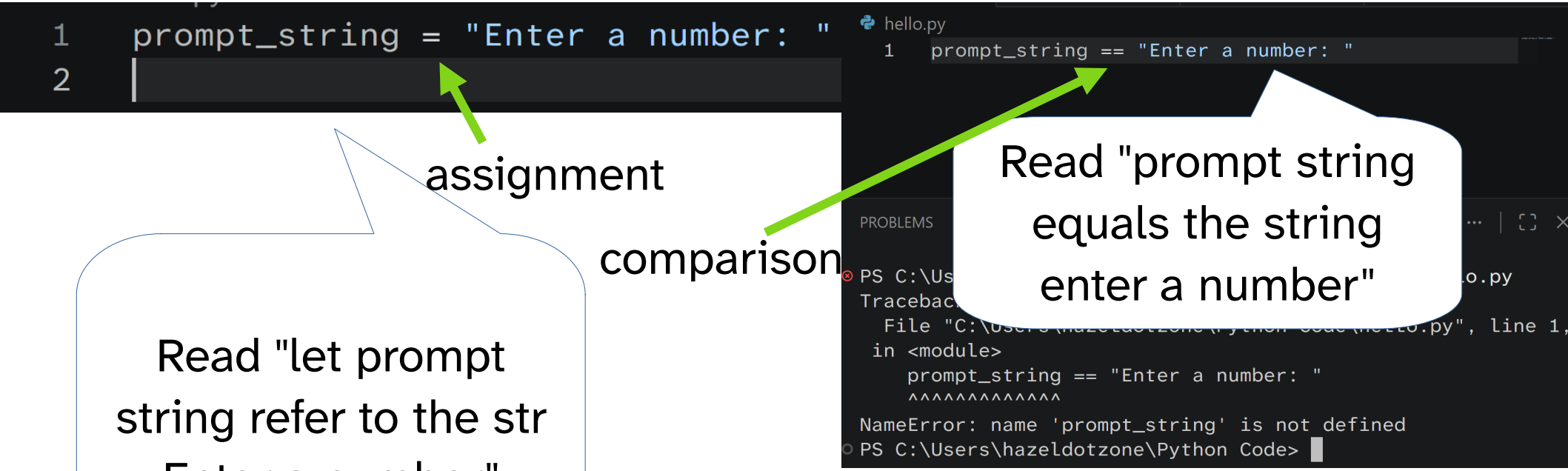
```
1   prompt_string == "Enter a number: "
```

PROBLEMS

```
PS C:\Us
Traceba
  File "C:\
 in <module>
    prompt_string == "Enter a number: "
    ^^^^^^^^^^^^^
NameError: name 'prompt_string' is not defined
PS C:\Users\hazeldotzone\Python Code>
```

Read "prompt string equals the string enter a number"

Making a Variable:
The Assignment Statement



Making a Variable:
The Assignment Statement

```
1   prompt_string = "Enter a number: "
2
```

Making a change to something!

hello.py

```
1   prompt_string == "Enter a number: "
```

```
PROBLEMS  OUTPUT  TERMINAL  powershell
PS C:\Users\hazeldotzone\Python Code> python .\hello.py
Traceback (most recent call last):
  File "C:\Users\hazeldotzone\Python Code\hello.py", line 1,
 in <module>
    prompt_string == "Enter a number: "
    ^^^^^^^^^^^^^
NameError: name 'prompt_string' is not defined
PS C:\Users\hazeldotzone\Python Code>
```

These two are very different!

Asking a question!

Making a Variable:
The Assignment Statement

```
1   prompt_string = "Enter a number: "
2
```

Read "let prompt string refer to the str Enter a number"

hello.py

```
1   prompt_string == "Enter a number: "
```

PROBLEMS

```
PS C:\Us
Traceba
  File "C:\
 in <module>
    prompt_string == "Enter a number: "
    ^^^^^^^^^^^^^
NameError: name 'prompt_string' is not defined
PS C:\Users\hazeldotzone\Python Code>
```

Read "prompt string equals the string enter a number"

Making a Variable:
The Assignment Statement

```
1   prompt_string = "Enter a number: "
2
```

or Read "assign prompt string to refer to the str Enter a number"

hello.py

```
1   prompt_string == "Enter a number: "
```

Read "prompt string equals the string enter a number"

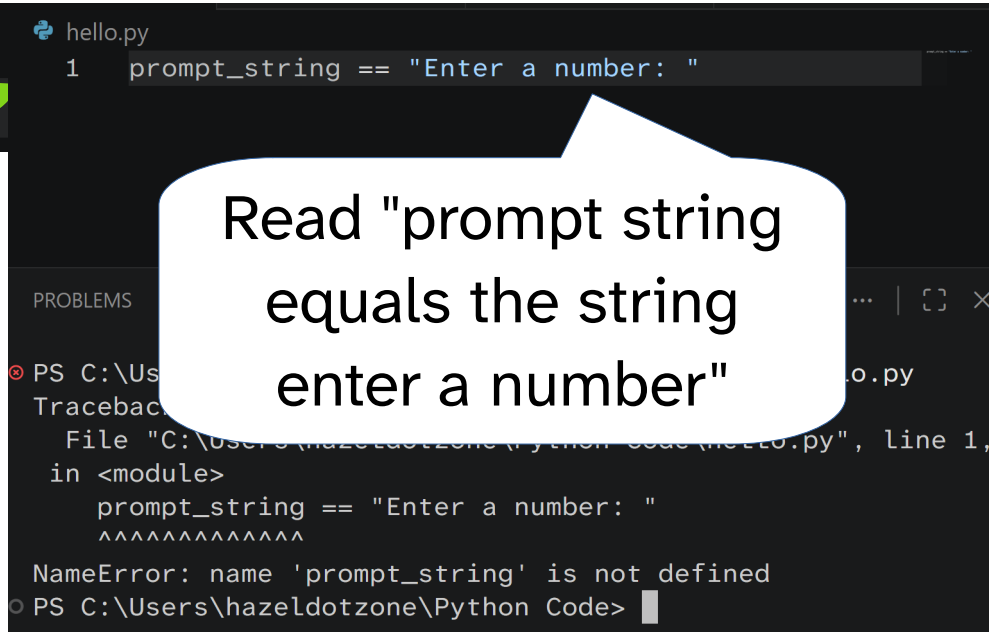
```
PROBLEMS

PS C:\Us                                    o.py
Traceba
  File "C:\                                 .py", line 1,
 in <module>
    prompt_string == "Enter a number: "
    ^^^^^^^^^^^^^
NameError: name 'prompt_string' is not defined
PS C:\Users\hazeldotzone\Python Code>
```

Making a Variable:
The Assignment Statement

```
1   prompt_string = "Enter a number: "
2
```

or Read "bind prompt string to the str Enter a number"

- ✔ bind
- ✔ assign
- ✔ let refer
- ✔ set refer
- ~~equals~~

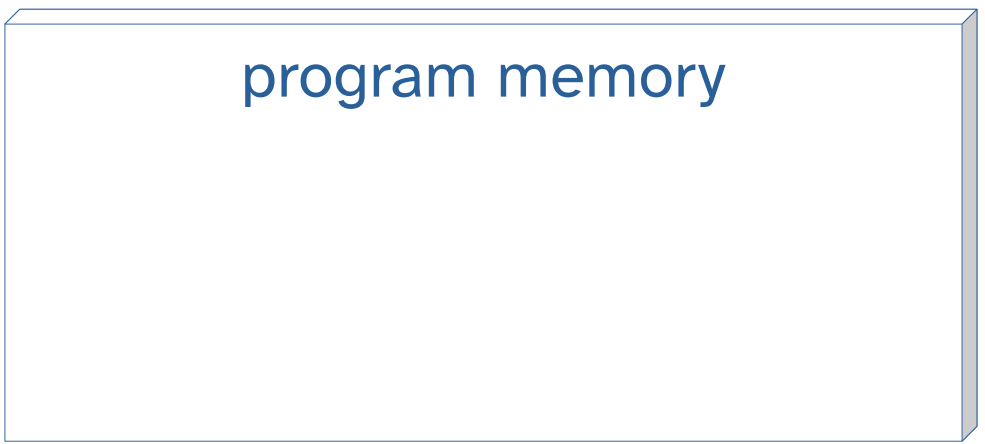


Read "prompt string equals the string enter a number"

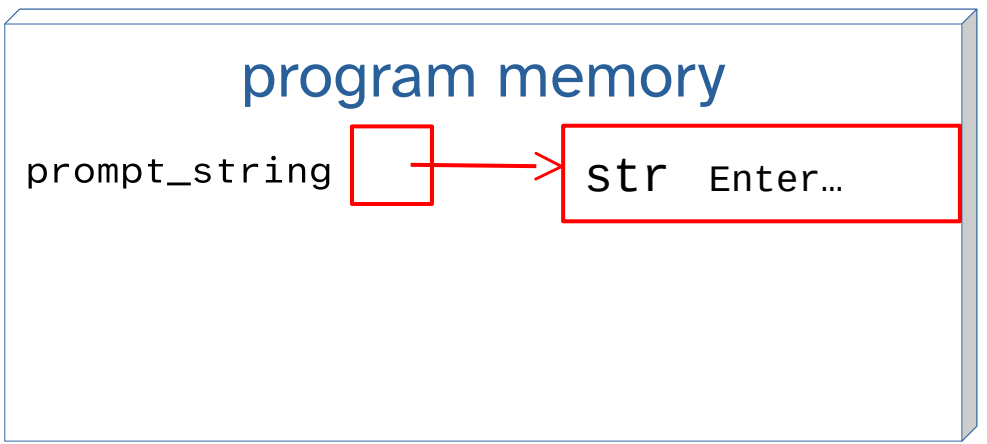
Making a Variable:
The Assignment Statement

```
1   prompt_string = "Enter a number: "
2
```

before the assignment statement



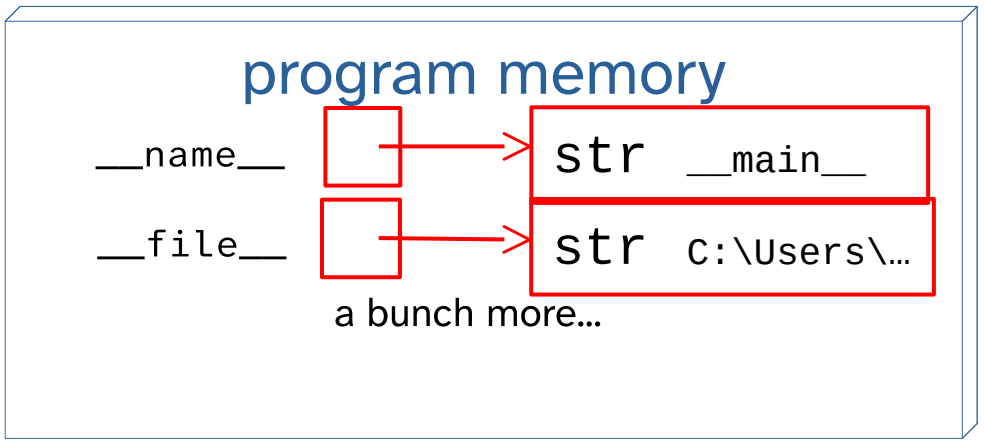
after the assignment statement



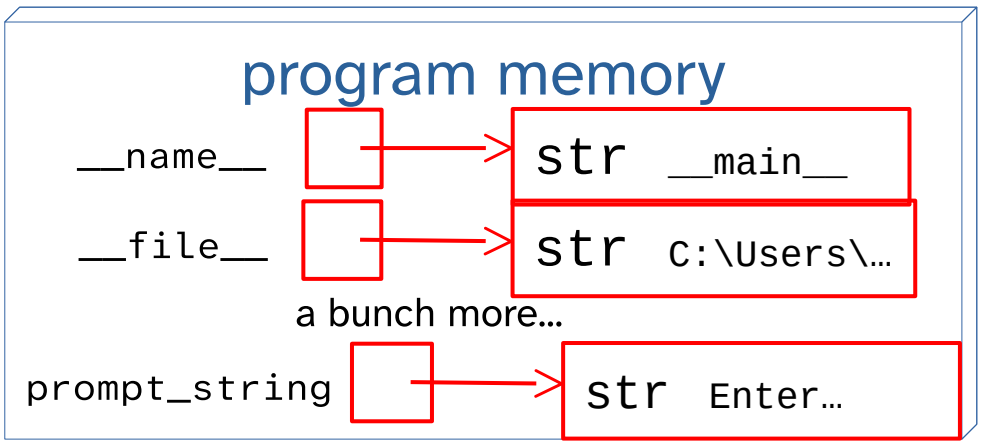
Making a Variable:
The Assignment Statement

```
1   prompt_string = "Enter a number: "
2
```

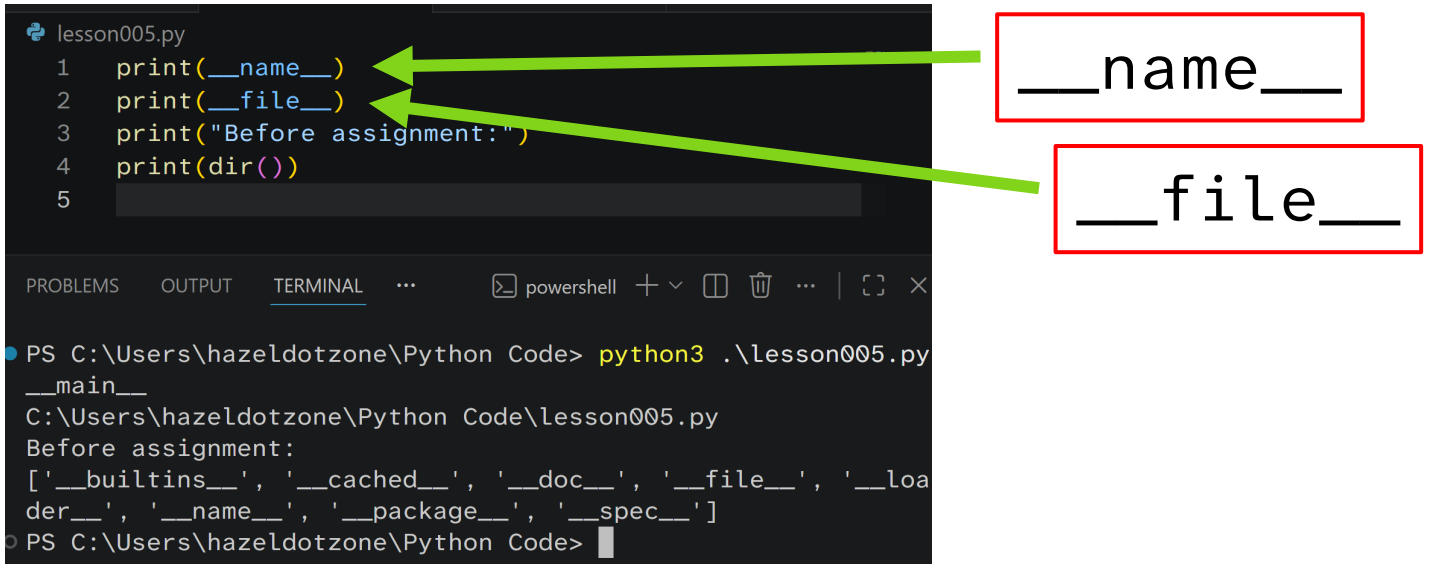
before the assignment statement



after the assignment statement



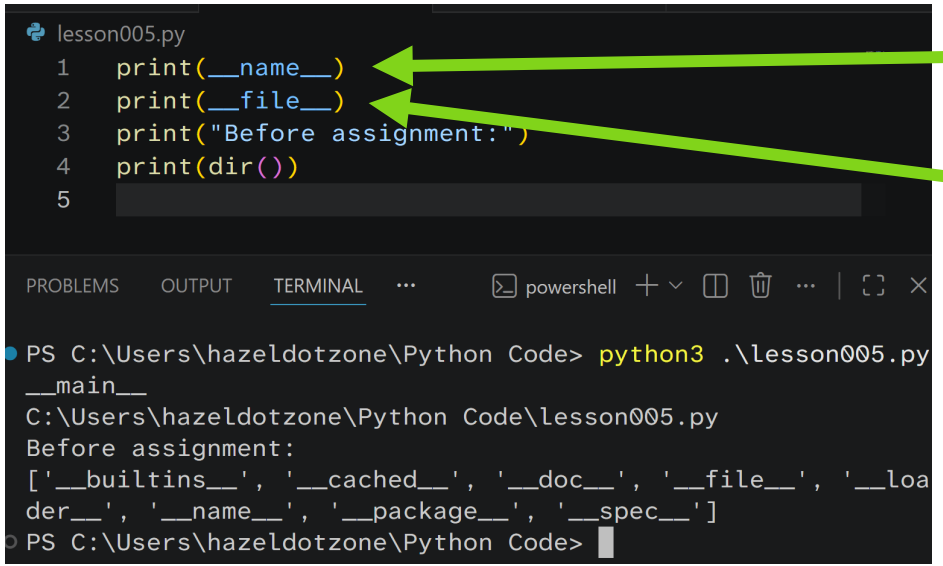
What variables do we have?



__name__

__file__

What variables do we have?

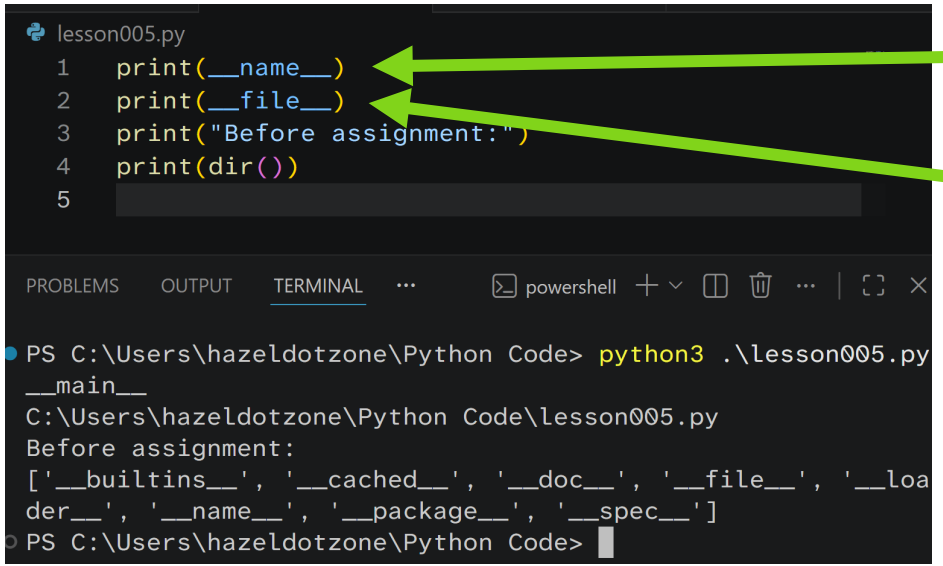


`__name__`

`__file__`

The `dir()` built-in function will tell us about
all the names we have around when we call it.

What variables do we have?



__name__

__builtins__

__file__

__cached__

__doc__

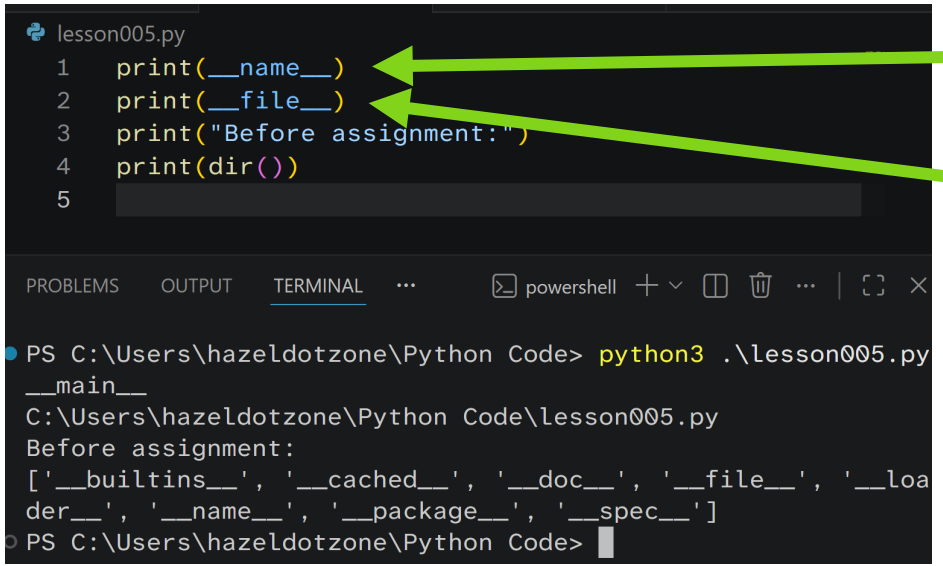
__loader__

__package__

__spec__

The `dir()` built-in function will tell us about all the names we have around when we call it.

What variables do we have?



__name__

__builtins__

__file__

__cached__

__doc__

__loader__

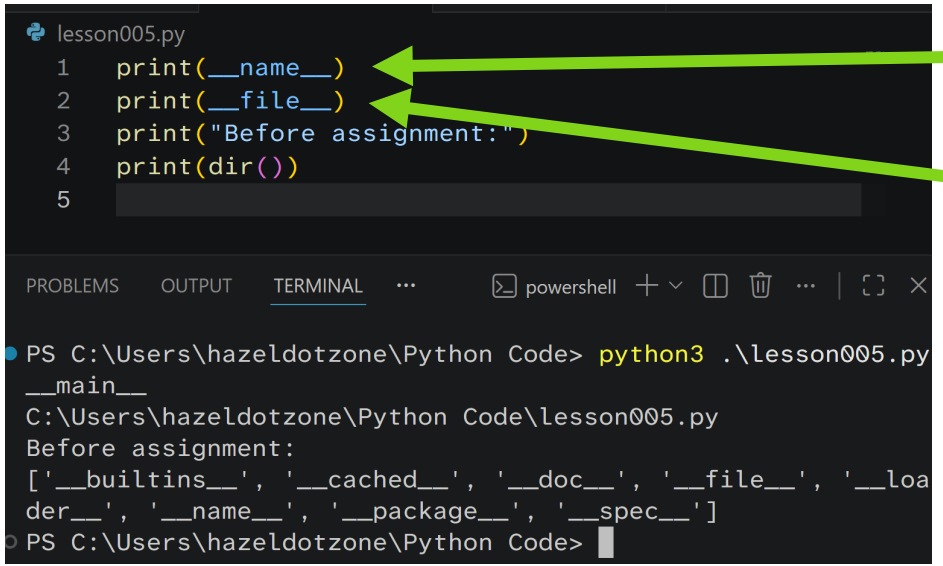
__package__

__spec__

These names all start with an _

What does that tell us about them?

What variables do we have?



__name__

__builtins__

__file__

__cached__

__doc__

__loader__

__package__

__spec__

These names all start with an _

What does that tell us about them?

Before and After

lesson005.py

```
1  print(__name__)
2  print(__file__)
3  print("Before assignment:")
4  print(dir())
5  prompt_string = "Enter a number: "
6  print("After assignment:")
7  print(dir())
8
```

PROBLEMS OUTPUT TERMINAL ··· powershell

```
PS C:\Users\hazeldotzone\Python Code> python .\lesson005.py
__main__
C:\Users\hazeldotzone\Python Code\lesson005.py
Before assignment:
['__annotations__', '__builtins__', '__cached__', '__doc__',
 '__file__', '__loader__', '__name__', '__package__', '__spe
c__']
After assignment:
['__annotations__', '__builtins__', '__cached__', '__doc__',
 '__file__', '__loader__', '__name__', '__package__', '__spe
c__', 'prompt_string']
PS C:\Users\hazeldotzone\Python Code>
```

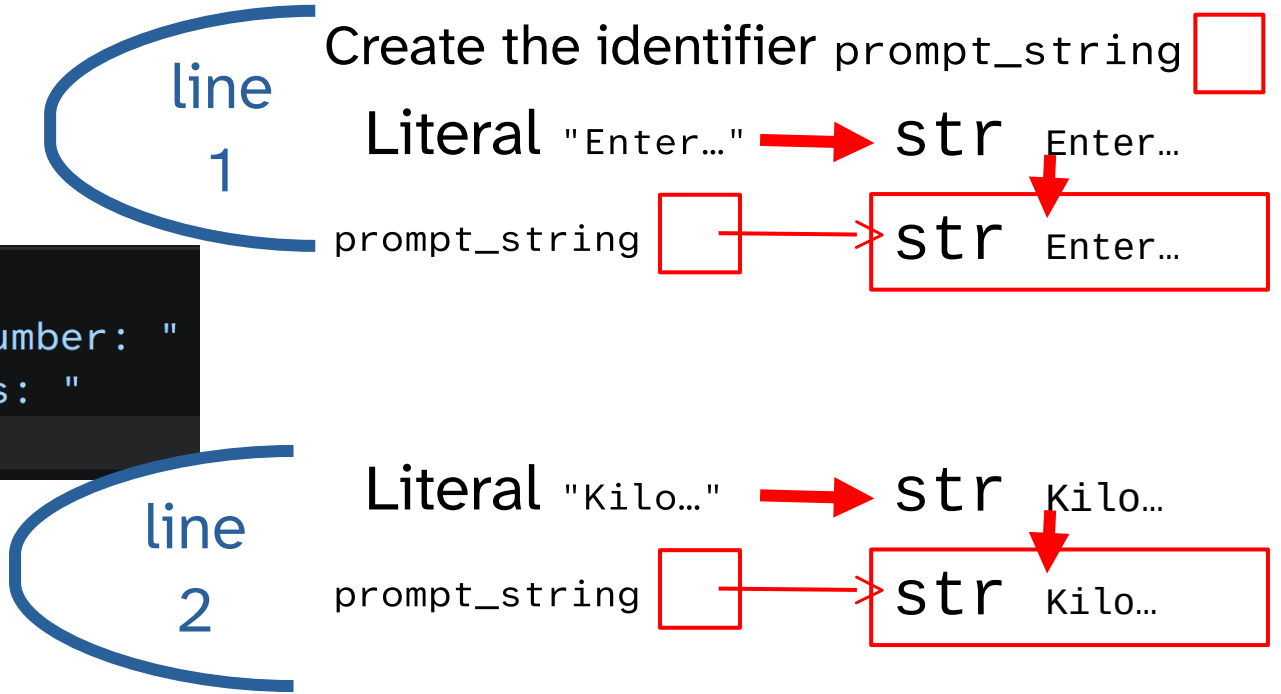
no prompt_string yet

it's in our variables

Rebinding

lesson005.py

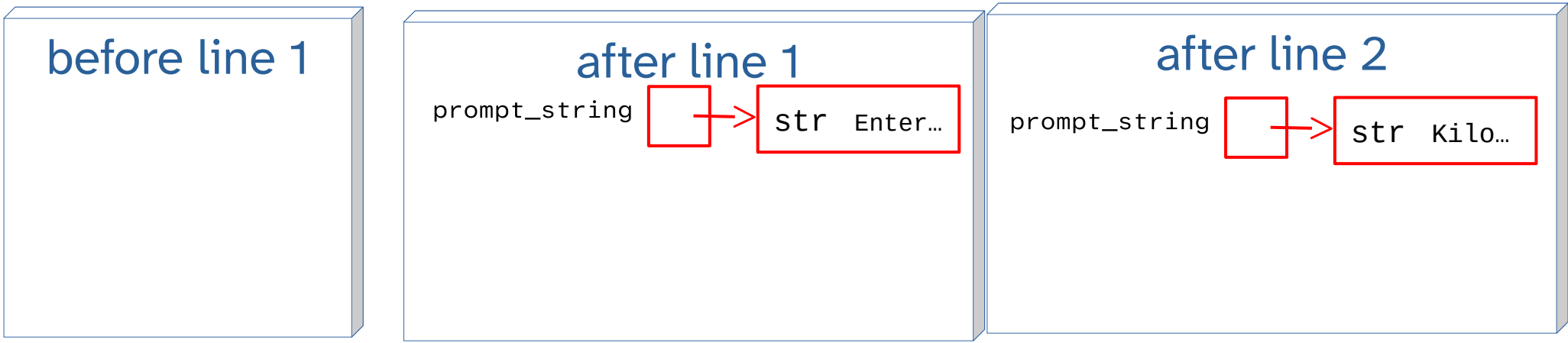
```
1   prompt_string = "Enter a number: "
2   prompt_string = "Kilometers: "
3
```



Rebinding

lesson005.py

```
1   prompt_string = "Enter a number: "
2   prompt_string = "Kilometers: "
3
```



Using Variables

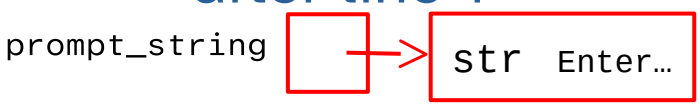
lesson005.py

```
1  prompt_string = "Enter a number: "
2  input(prompt_string)
3  prompt_string = "Kilometers: "
4  input(prompt_string)
5
```

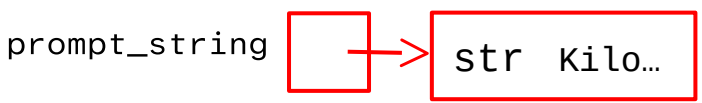
```
PROBLEMS  OUTPUT  TERMINAL  ...  powershell
PS C:\Users\hazeldotzone\Python Code> python .\lesson005.py
Enter a number:
Kilometers:
PS C:\Users\hazeldotzone\Python Code>
```

before line 1

after line 1



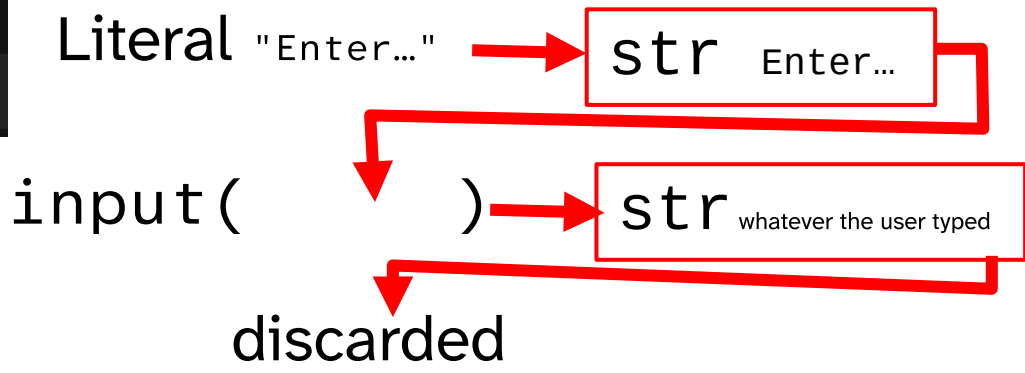
after line 3

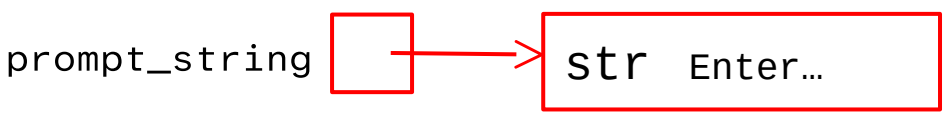


Using Variables

```
10  input("Enter a number: ")
11
```

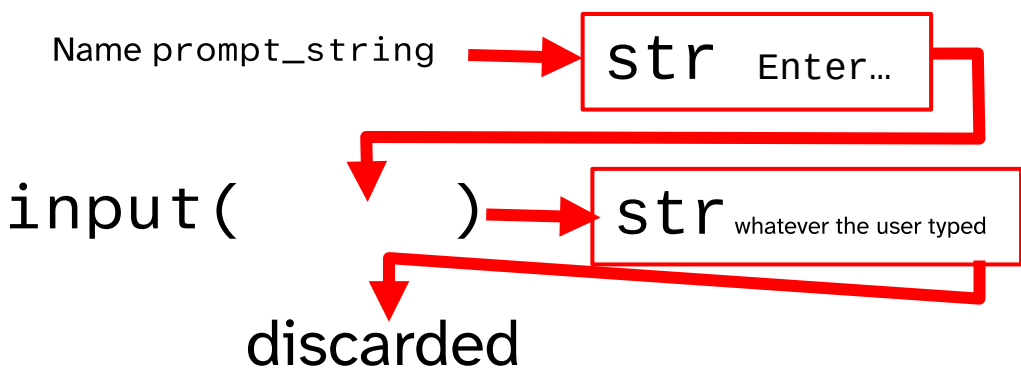
Expression statement






```
4  input(prompt_string)
5
```

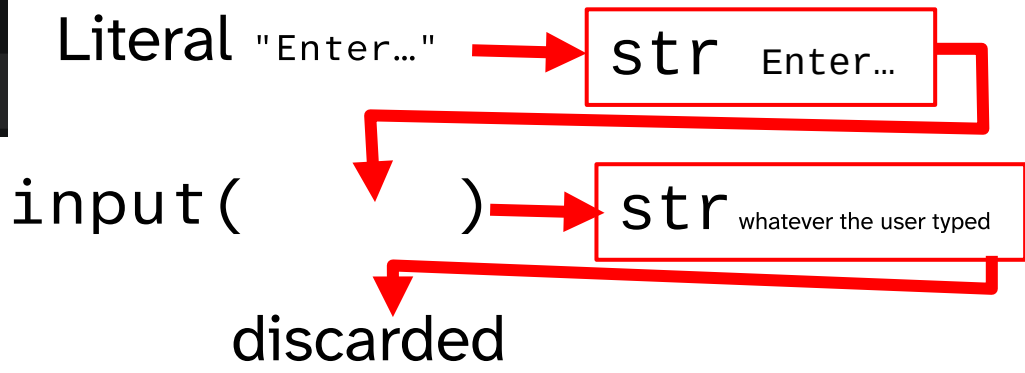
Expression statement



Using Variables

```
10  input("Enter a number: ")
11
```

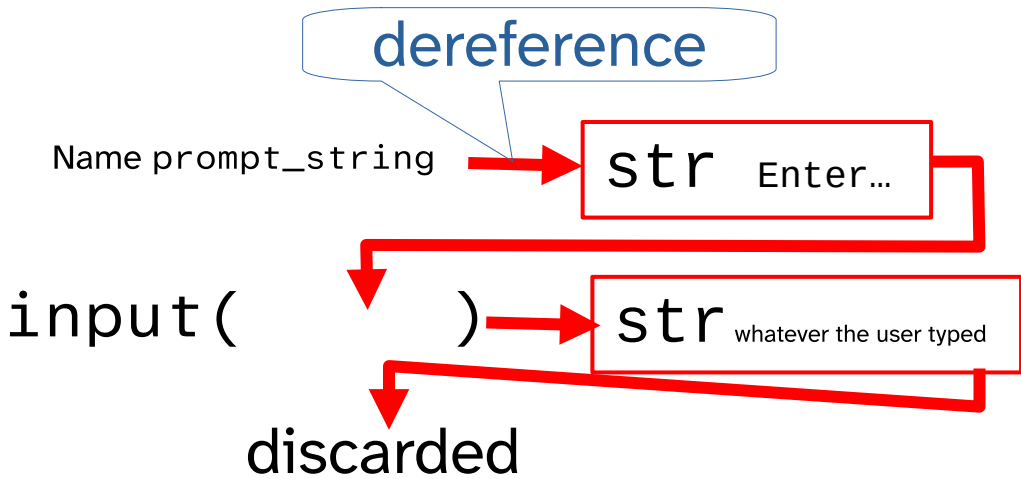
Expression statement



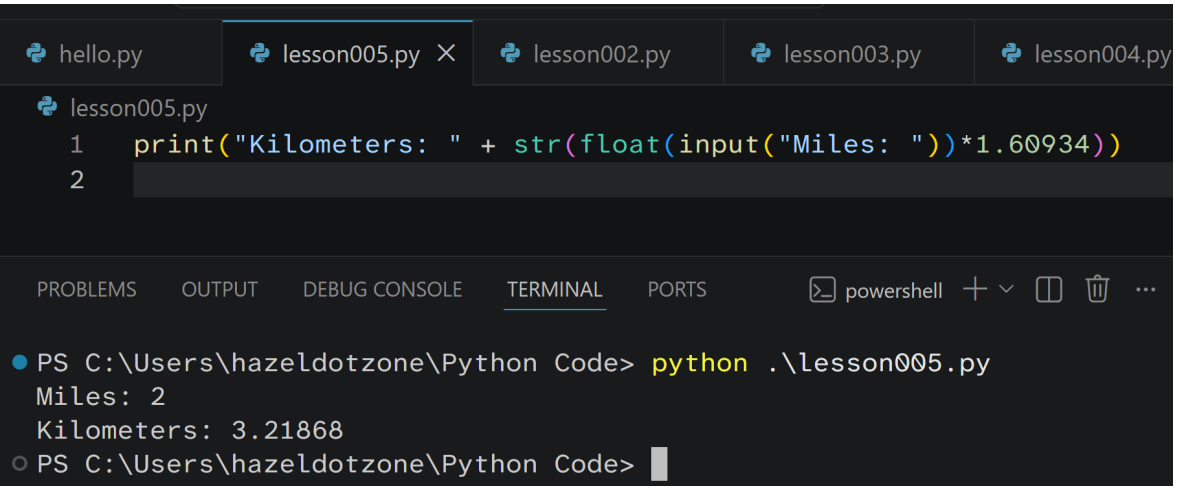
prompt_string


```
4  input(prompt_string)
5
```

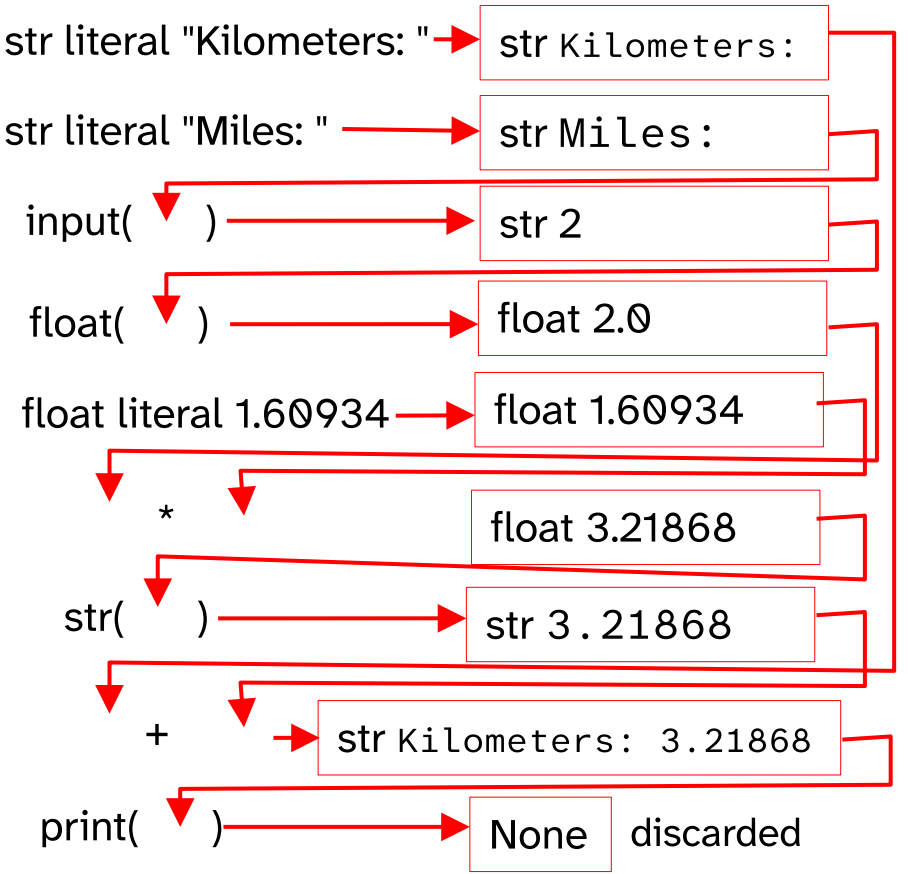
Expression statement



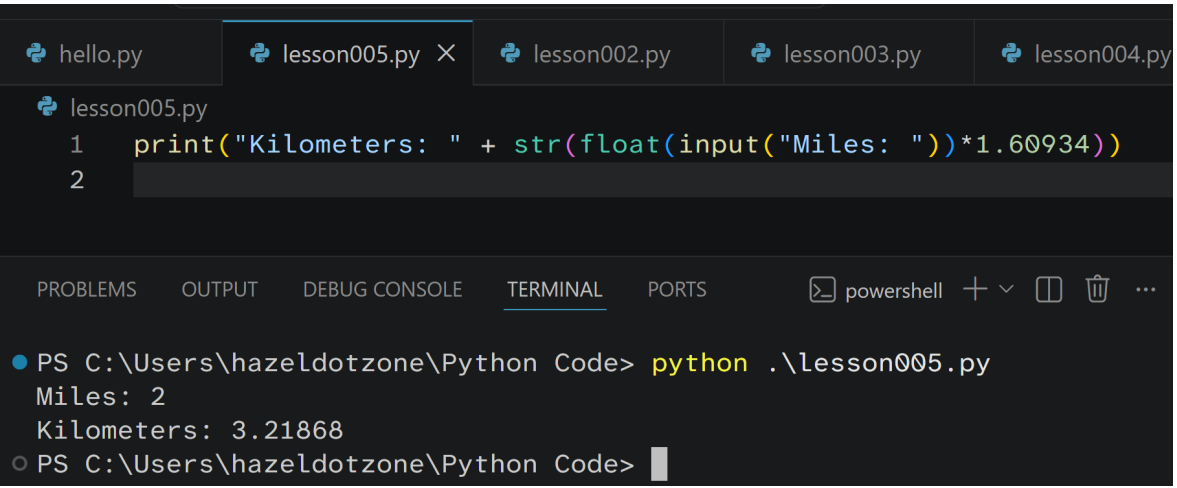
Programming The Computer



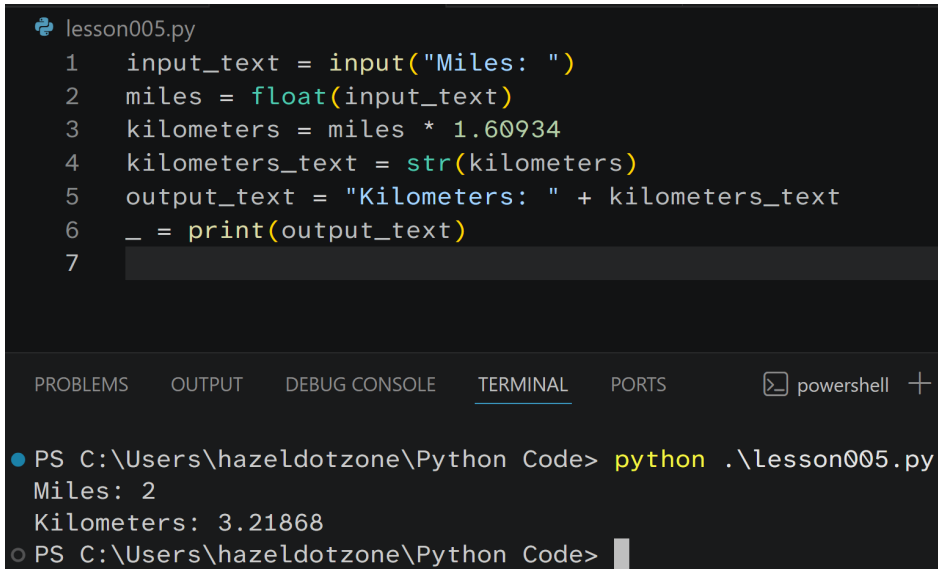
"Functional Style"



Programming The Computer



"Functional Style"



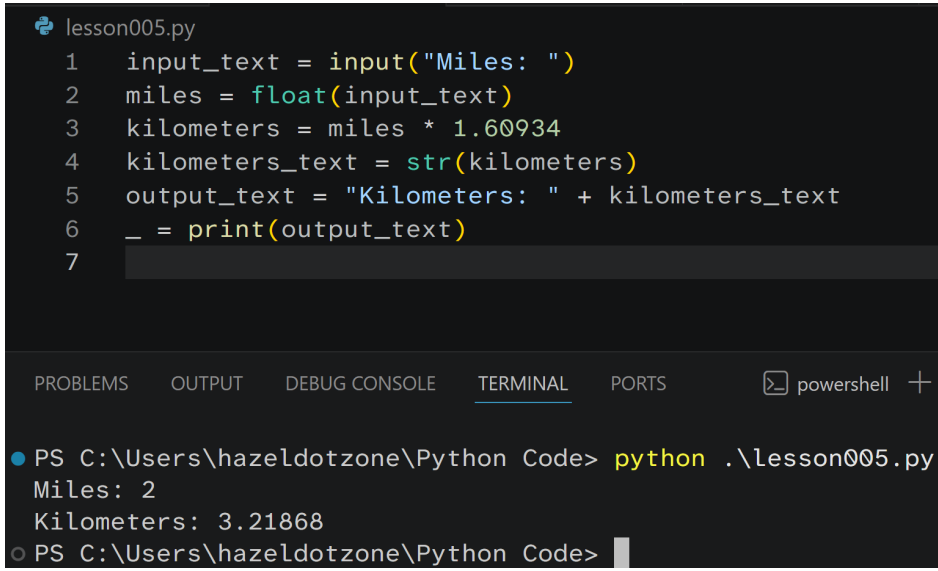
"Three Address Style"

Each line has 3 names or literals and an operator, using the format name = thing op thing

Programming The Computer

```
input_text = input("Miles: ")
```

name = name call literal



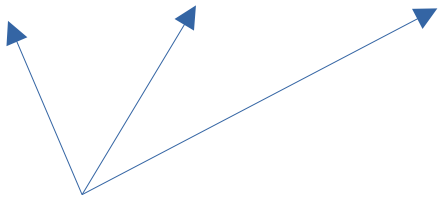
"Three Address Style"

Each line has 3 names or literals and an operator, using the format name = thing op thing

Programming The Computer

```
2   miles = float(input_text)
```

name = name call name



the three addresses

```
lesson005.py
1   input_text = input("Miles: ")
2   miles = float(input_text)
3   kilometers = miles * 1.60934
4   kilometers_text = str(kilometers)
5   output_text = "Kilometers: " + kilometers_text
6   _ = print(output_text)
7

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  powershell

PS C:\Users\hazeldotzone\Python Code> python .\lesson005.py
Miles: 2
Kilometers: 3.21868
PS C:\Users\hazeldotzone\Python Code>
```

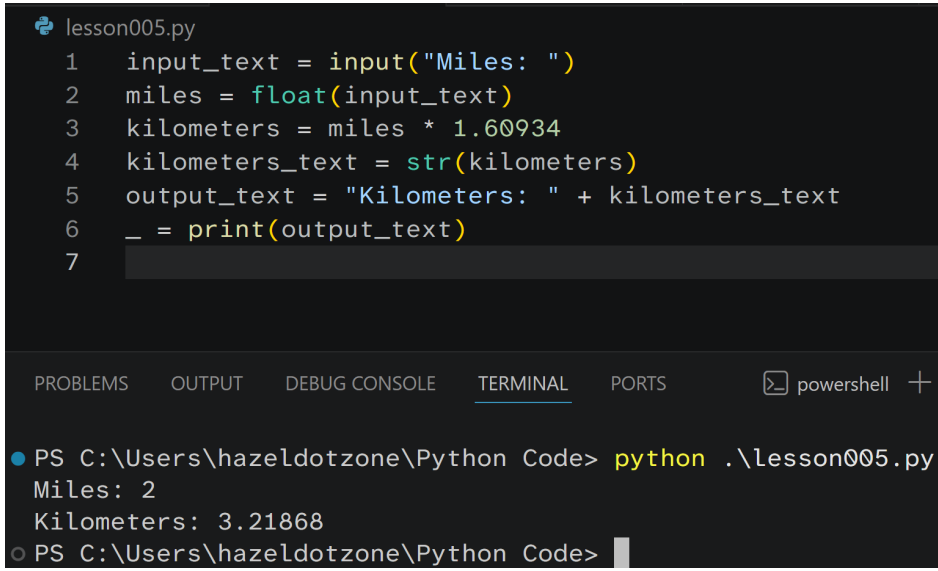
"Three Address Style"

Each line has 3 names or literals and an operator, using the format name = thing operation thing

Programming The Computer

```
kilometers = miles * 1.60934
```

name = name multiply literal



"Three Address Style"

Each line has 3 names or literals and an operator, using the format name = thing operation thing

Programming The Computer

```
kilometers_text = str(kilometers)
```

name = name call name

lesson005.py

```
1  input_text = input("Miles: ")
2  miles = float(input_text)
3  kilometers = miles * 1.60934
4  kilometers_text = str(kilometers)
5  output_text = "Kilometers: " + kilometers_text
6  _ = print(output_text)
7
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell

```
PS C:\Users\hazeldotzone\Python Code> python .\lesson005.py
Miles: 2
Kilometers: 3.21868
PS C:\Users\hazeldotzone\Python Code>
```

"Three Address Style"

Each line has 3 names or literals and an operator, using the format name = thing operation thing

Programming The Computer

```
output_text = "Kilometers: " + kilometers_text
```

name = literal + name

```
005.py
1  input_text = input("Miles: ")
2  miles = float(input_text)
3  kilometers = miles * 1.60934
4  kilometers_text = str(kilometers)
5  output_text = "Kilometers: " + kilometers_text
6  _ = print(output_text)
7
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  powershell
PS C:\Users\hazeldotzone\Python Code> python .\lesson005.py
Miles: 2
Kilometers: 3.21868
PS C:\Users\hazeldotzone\Python Code>
```

"Three Address Style"

Each line has 3 names or literals and an operator, using the format name = thing operation thing

Programming The Computer

```
_ = print(output_text)
```

name = name call name

lesson005.py

```
1  input_text = input("Miles: ")
2  miles = float(input_text)
3  kilometers = miles * 1.60934
4  kilometers_text = str(kilometers)
5  output_text = "Kilometers: " + kilometers_text
6  _ = print(output_text)
7
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell

```
PS C:\Users\hazeldotzone\Python Code> python .\lesson005.py
Miles: 2
Kilometers: 3.21868
PS C:\Users\hazeldotzone\Python Code>
```

"Three Address Style"

Each line has 3 names or literals and an operator, using the format name = thing operation thing

Programming The Computer

```
_ = print(output_text)
```

name = name call name

```
lesson005.py
1  input_text = input("Miles: ")
2  miles = float(input_text)
3  kilometers = miles * 1.60934
4  kilometers_text = str(kilometers)
5  output_text = "Kilometers: " + kilometers_text
6  _ = print(output_text)
7

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  powershell

PS C:\Users\hazeldotzone\Python Code> python .\lesson005.py
Miles: 2
Kilometers: 3.21868
PS C:\Users\hazeldotzone\Python Code>
```

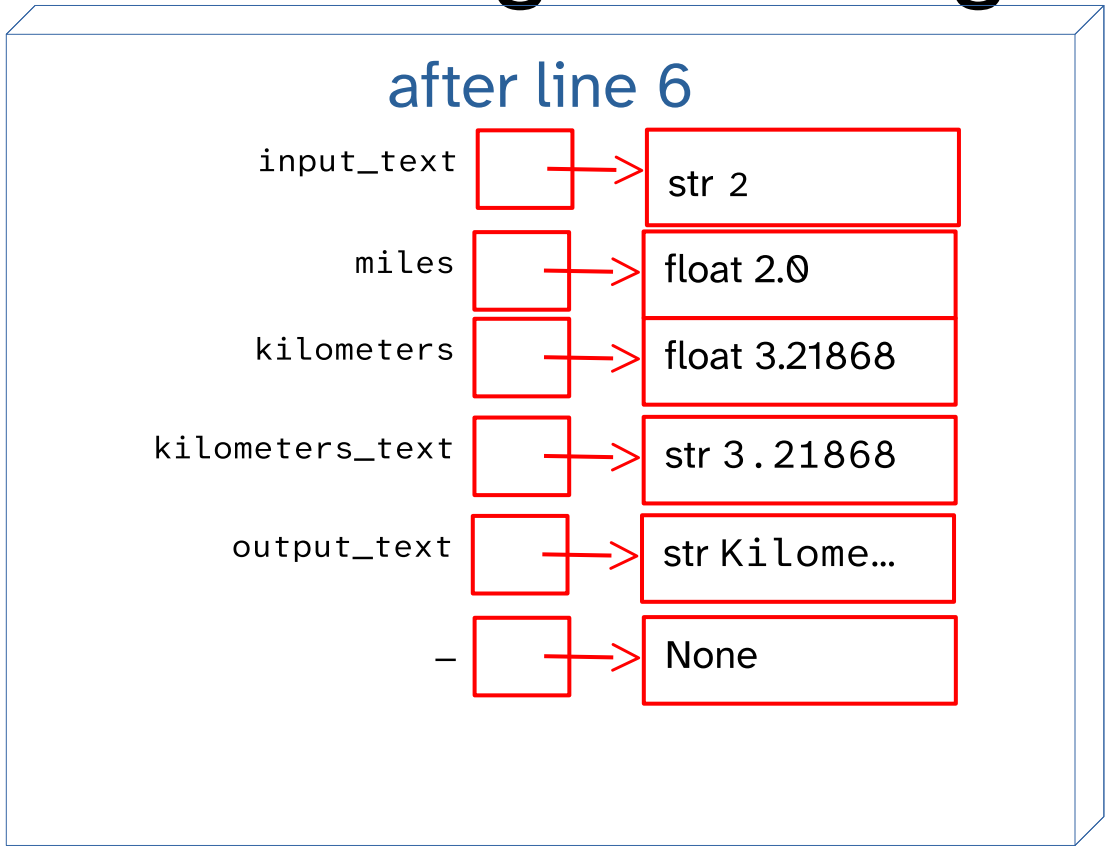
Remember, idiomatic Python uses the name _ to indicate the assigned object won't be used

_ should never show up in an expression!

"Three Address Style"

Each line has 3 names or literals and an operator, using the format name = thing operation thing

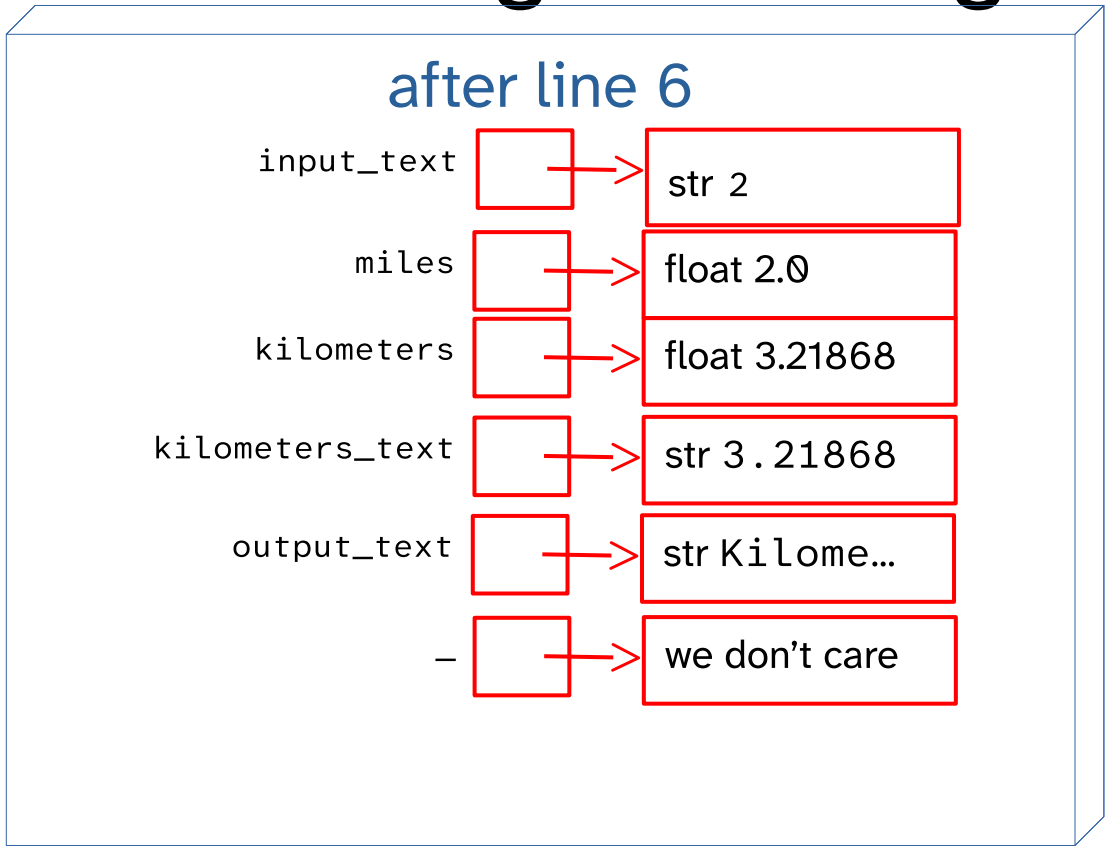
Programming The Computer




```
lesson005.py
1  input_text = input("Miles: ")
2  miles = float(input_text)
3  kilometers = miles * 1.60934
4  kilometers_text = str(kilometers)
5  output_text = "Kilometers: " + kilometers_text
6  _ = print(output_text)
7
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  powershell

PS C:\Users\hazeldotzone\Python Code> python .\lesson005.py
Miles: 2
Kilometers: 3.21868
PS C:\Users\hazeldotzone\Python Code>
```

Programming The Computer



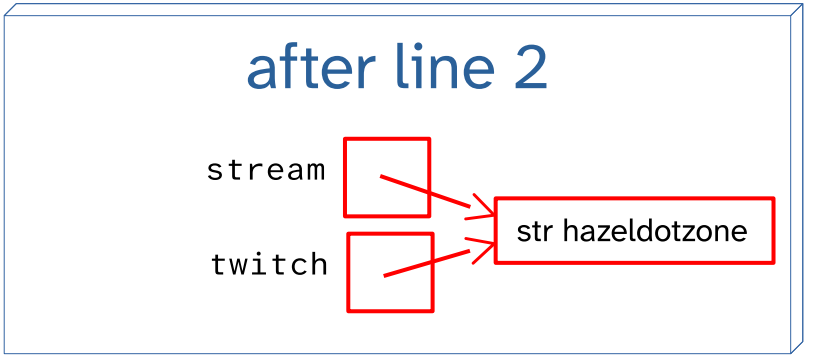

```
lesson005.py
1  input_text = input("Miles: ")
2  miles = float(input_text)
3  kilometers = miles * 1.60934
4  kilometers_text = str(kilometers)
5  output_text = "Kilometers: " + kilometers_text
6  _ = print(output_text)
7
```

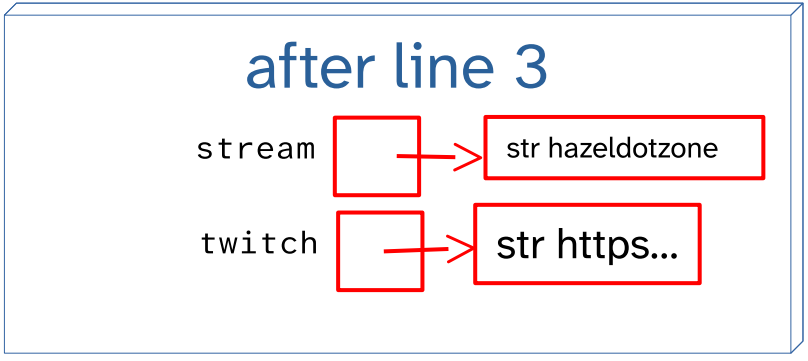
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  powershell

PS C:\Users\hazeldotzone\Python Code> python .\lesson005.py
Miles: 2
Kilometers: 3.21868
PS C:\Users\hazeldotzone\Python Code>
```

Names Are References

lesson005.py

```
1   stream = "hazeldotzone"
2   twitch = stream
3   twitch = "https://www.twitch.tv/"+twitch
4
```

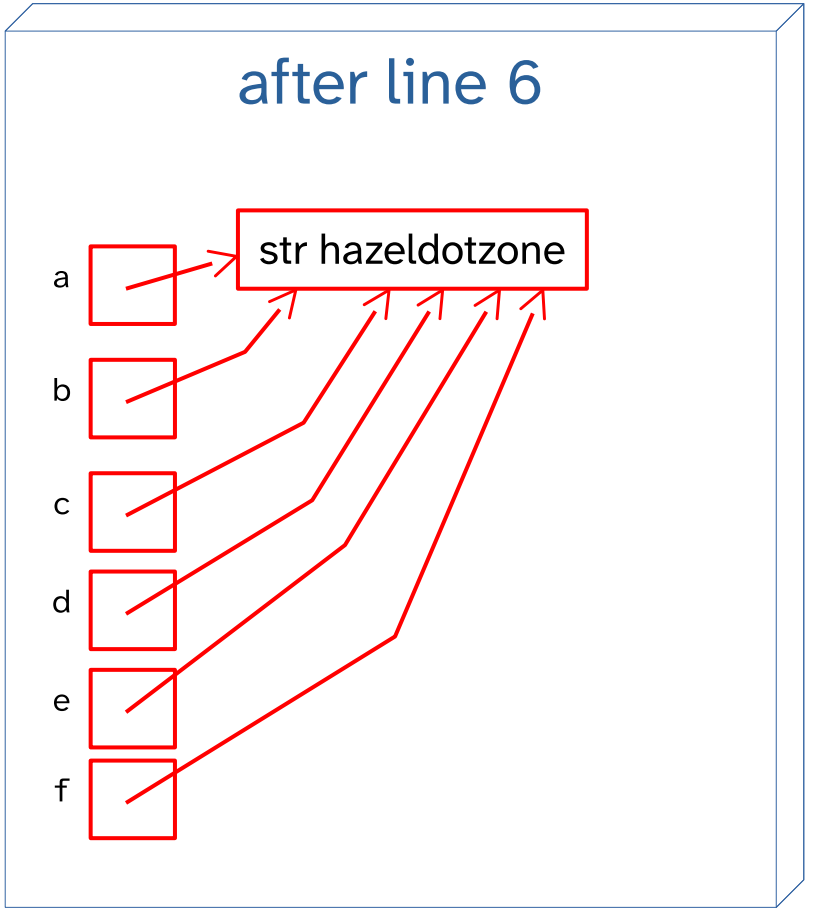




Names Are References

lesson005.py

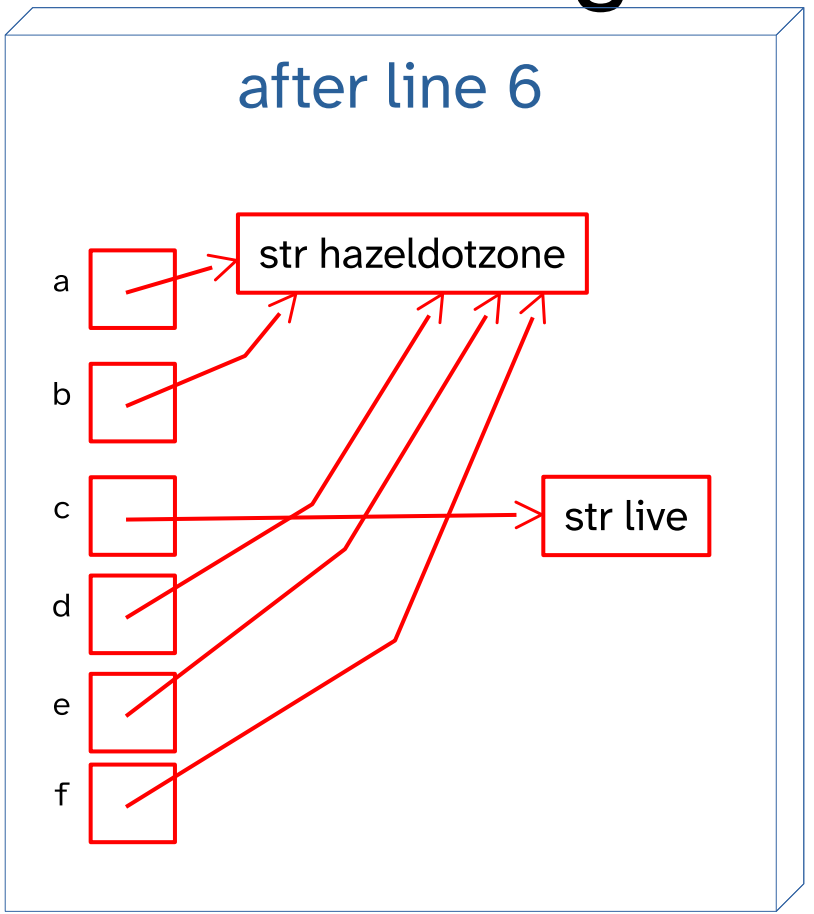
```
1    a = "hazeldotzone"
2    b = a
3    c = a
4    d = c
5    e = a
6    f = d
7
```



Remember: = changes the binding!

lesson005.py

```
1    a = "hazeldotzone"
2    b = a
3    c = a
4    d = c
5    e = a
6    f = e
7    c = "live"
```

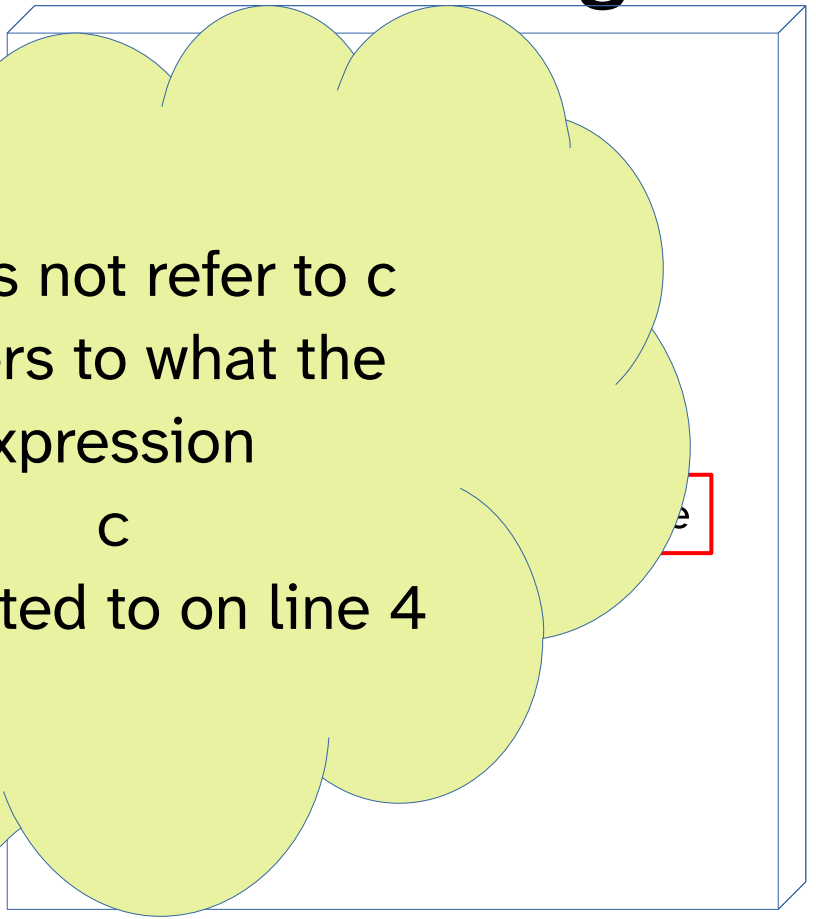


Remember: = changes the binding!

lesson005.py

```
1    a = "hazeldotz
2    b = a
3    c = a
4    d = c
5    e = a
6    f = e
7    c = "live"
```

d does not refer to c
d refers to what the expression
c
evaluated to on line 4

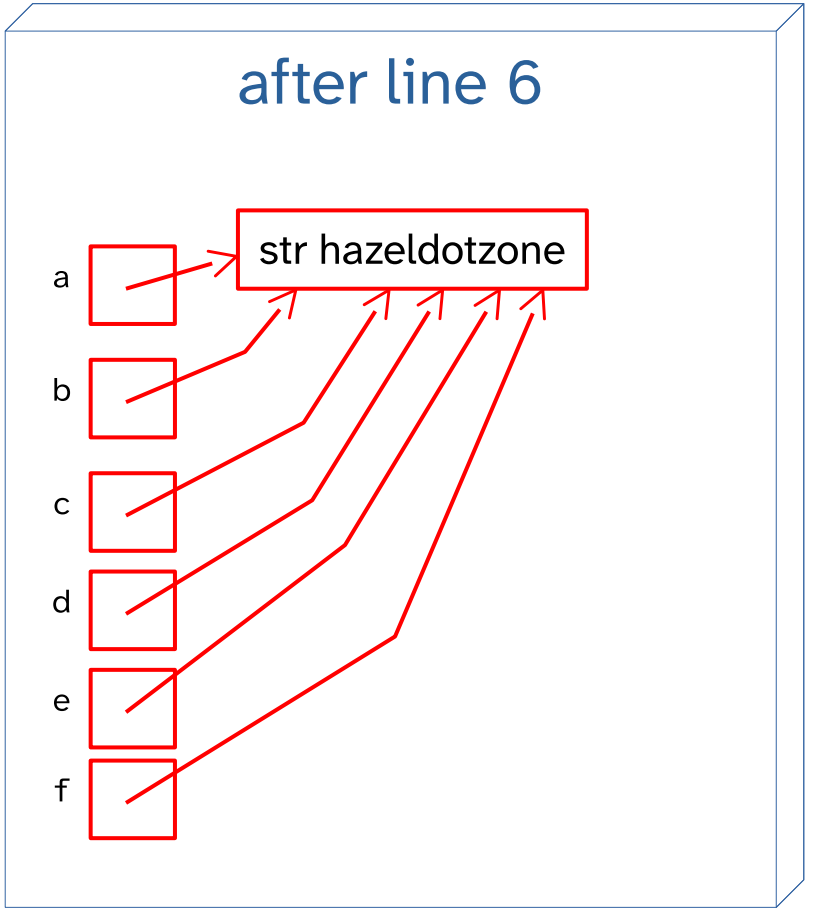


Names Are References

lesson005.py

```
1    a = "hazeldotzone"
2    b = "hazeldotzone"
3    c = "hazeldotzone"
4    d = "hazeldotzone"
5    e = "hazeldotzone"
6    f = "hazeldotzone"
7
```

Wait really?



The `is` Binary Comparison Operator

lesson005.py

```
1   a = "hazeldotzone"
2   b = "hazeldotzone"
3   c = "hazeldotzone"
4   d = "hazeldotzone"
5   e = "hazeldotzone"
6   f = "hazeldotzone"
7   print(b is a)
8   print(c is a)
9   print(d is a)
10  print(e is a)
11  print(f is a)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell

```
PS C:\Users\hazeldotzone\Python Code> python .\lesson005.py
True
True
True
True
True
```

The is operator evaluates to true if both sides are the same object

The `is` Binary Comparison Operator

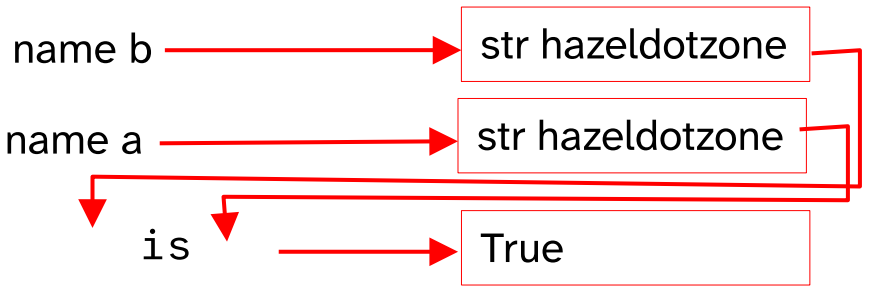
lesson005.py

```
1   a = "hazeldotzone"
2   b = "hazeldotzone"
3   c = "hazeldotzone"
4   d = "hazeldotzone"
5   e = "hazeldotzone"
6   f = "hazeldotzone"
7   print(b is a)
8   print(c is a)
9   print(d is a)
10  print(e is a)
11  print(f is a)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell

```
PS C:\Users\hazeldotzone\Python Code> python .\lesson005.py
True
True
True
True
True
```

The is operator evaluates to true if both sides are the same object



Both of these strings are the same object!

The `is` Binary Comparison Operator

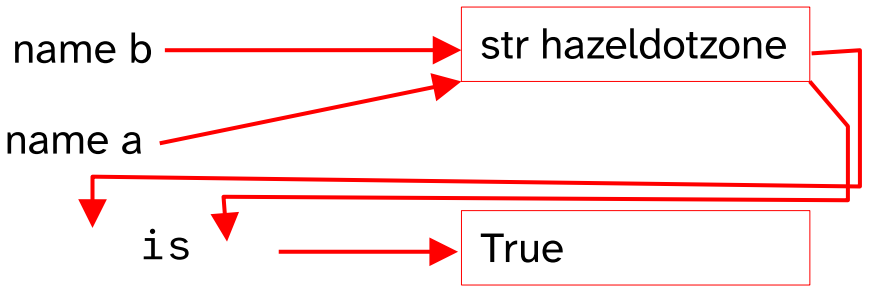
lesson005.py

```
1   a = "hazeldotzone"
2   b = "hazeldotzone"
3   c = "hazeldotzone"
4   d = "hazeldotzone"
5   e = "hazeldotzone"
6   f = "hazeldotzone"
7   print(b is a)
8   print(c is a)
9   print(d is a)
10  print(e is a)
11  print(f is a)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell

```
PS C:\Users\hazeldotzone\Python Code> python .\lesson005.py
True
True
True
True
True
```

The is operator evaluates to true if both sides are the same object



Both of these strings are the same object!

The `is` Binary Comparison Operator

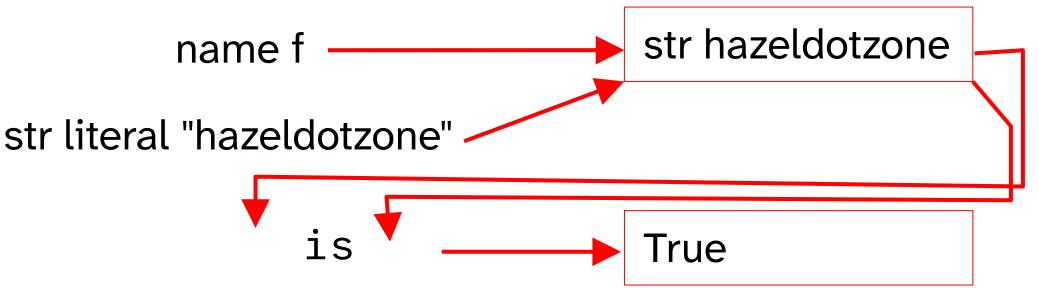
lesson005.py

```
 6   f = "hazeldotzone"
 7   print(b is a)
 8   print(c is a)
 9   print(d is a)
10   print(e is a)
11   print(f is a)
12   print(f is "hazeldotzone")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell

```
PS C:\Users\hazeldotzone\Python Code> python .\lesson005.py
True
True
True
True
True
PS C:\Users\hazeldotzone\Python Code>
```

The is operator evaluates to true if both sides are the same object



Both of these strings are the same object!

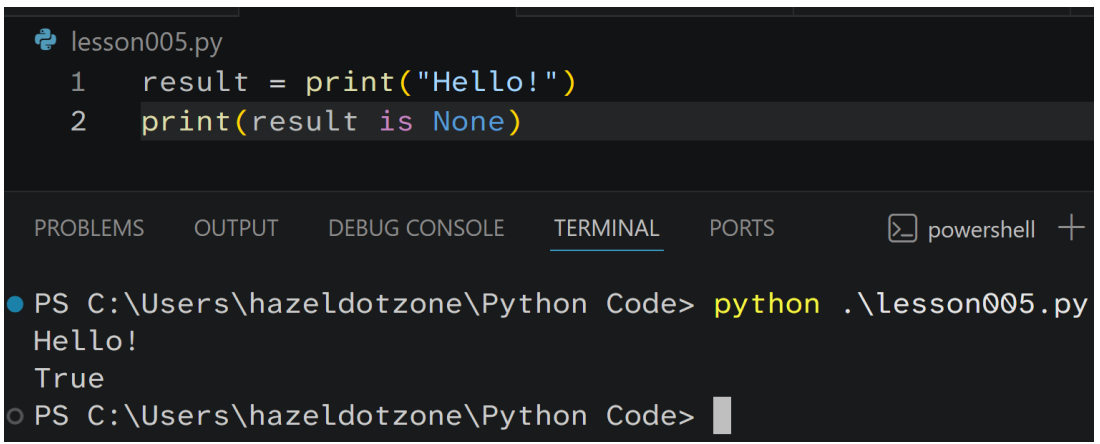
The Same Object?

- In Python, usually the following are the same object:
  - strings that are equal
  - numbers that are equal
  - True
  - False
  - None

The Same Object?

- In Python, usually the following are the same object:
  - strings that are equal
  - numbers that are equal
  - True
  - False
  - None

- You end up seeing this a lot `some_variable is None`



The Same Object?

- In Python, usually the following are the same object:
  - strings that are equal
  - numbers that are equal
  - True
  - False
  - None

- This is not the same as other programming languages.
- We shouldn't rely on it for numbers and strings
  - use == instead

(but it is useful for checking how variables work!)

The `is` Binary Comparison Operator

in type	characters	in type	result	operation
	`(  )`			delimited expression
`int`	`**`	`int`	int	exponentiation
	`+`	`int`	int	positive
	`-`	`int`	int	negative / negation
`int`	`*`	`int`	int	multiplication
`int`	`/`	`int`	float	floating point division
`int`	`//`	`int`	int	floor division
`int`	`%`	`int`	int	remainder
`int`	`+`	`int`	int	addition
`int`	`-`	`int`	int	subtraction
`int`	`<`	`int`	bool	less than
`int`	`<=`	`int`	bool	less than equal
`int`	`>`	`int`	bool	greater than
`int`	`>=`	`int`	bool	greater than equal
`int`	`==`	`int`	bool	equal
`int`	`!=`	`int`	bool	unequal
anything	`is`	anything	bool	same object
anything	`is not`	anything	bool	different object

P

E

U

MD

AS

C

`is`
and
`is not`
have the same precedence as the rest of the comparison operators