



# Assignment 1


**Out:** Today 5:00 p.m.

**Due:** Tuesday 11:59 p.m.

I will make every effort to give each of you the attention and feedback you need to be successful in this course – but there’s only one of me! Therefore, I rely on the coaches to help me help answer your questions.

In addition to working during our labs each week, each coach will be available to help you in the Agile Lab (sc 006) at scheduled times.

**Important:** The coaches are prohibited from giving you the solutions to labs and assignments, but they are able to guide you as you work to solve your programming tasks. When this works well, they will help you answer your own questions!

September 2025							
	SUN	MON	TUE	WED	THU	FRI	SAT
	7	8	9	10	11	12	13
07:00							
08:00							



We've been using Python to write expressions using

data, including

*integers* like 0, 4, and -10;

*floating-point numbers* like 4.0, 0.3, and -12.5; and

*strings* like "", "hi", and "111",

which we combine or transform using operators like + and \* and functions like **max** and **abs**.

We've seen that we can create more complicated programs by composing operations or function calls, e.g.,

$$1 + (2 / 3)$$

or

$$\text{abs}(\text{min}(4, 5, -1))$$

And we can give a name to the result of an expression, e.g.,

*total* = 2 + 3

And we can give a name to the result of an expression, e.g.,

*total* = 2 + 3

*Directory*

<i>Directory</i>	
<i>Name</i>	<i>Value</i>
<i>total</i>	



And we can give a name to the result of an expression, e.g.,

*total* = 2 + 3  
→ *total* = 5

Directory	
Name	Value
<i>total</i>	

And we can give a name to the result of an expression, e.g.,

*total* = 2 + 3  
→ *total* = 5

Directory	
Name	Value
<i>total</i>	5

And we can give a name to the result of an expression, e.g.,

*total* = 2 + 3  
→ *total* = 5

*new\_total* = total + 1

Directory	
Name	Value
<i>total</i>	5

And we can give a name to the result of an expression, e.g.,

*total* = 2 + 3  
→ *total* = 5  
  
*new\_total* = total + 1

Directory	
Name	Value
<i>total</i>	5
<i>new_total</i>	

And we can give a name to the result of an expression, e.g.,

*total* = 2 + 3  
→ *total* = 5

*new\_total* = total + 1

Directory	
Name	Value
<i>total</i>	5
<i>new_total</i>	

And we can give a name to the result of an expression, e.g.,

$total = 2 + 3$   
→  $total = 5$

$new\_total = total + 1$   
→  $new\_total = 5 + 1$

Directory	
Name	Value
$total$	5
$new\_total$	

And we can give a name to the result of an expression, e.g.,

*total* = 2 + 3  
→ *total* = 5

*new\_total* = total + 1  
→ *new\_total* = 5 + 1  
→ *new\_total* = 6

Directory	
Name	Value
<i>total</i>	5
<i>new_total</i>	

And we can give a name to the result of an expression, e.g.,

```
total = 2 + 3  
→ total = 5  
  
new_total = total + 1  
→ new_total = 5 + 1  
→ new_total = 6
```

Directory	
Name	Value
<i>total</i>	5
<i>new_total</i>	6



When we're writing Python, we'll make mistakes, so we'll see error messages.

*Syntax error*: Code doesn't follow syntactic requirements

E.g., `9+-`

*Runtime error*: Valid syntax; can't evaluate for other reasons

E.g., `5/0`

*Bug*: Code runs – but not the way you intended!

Python will let you re-assign the value for some of its *built-in* names, including functions like `min`, `max`, and `pow` – even though you probably shouldn't!

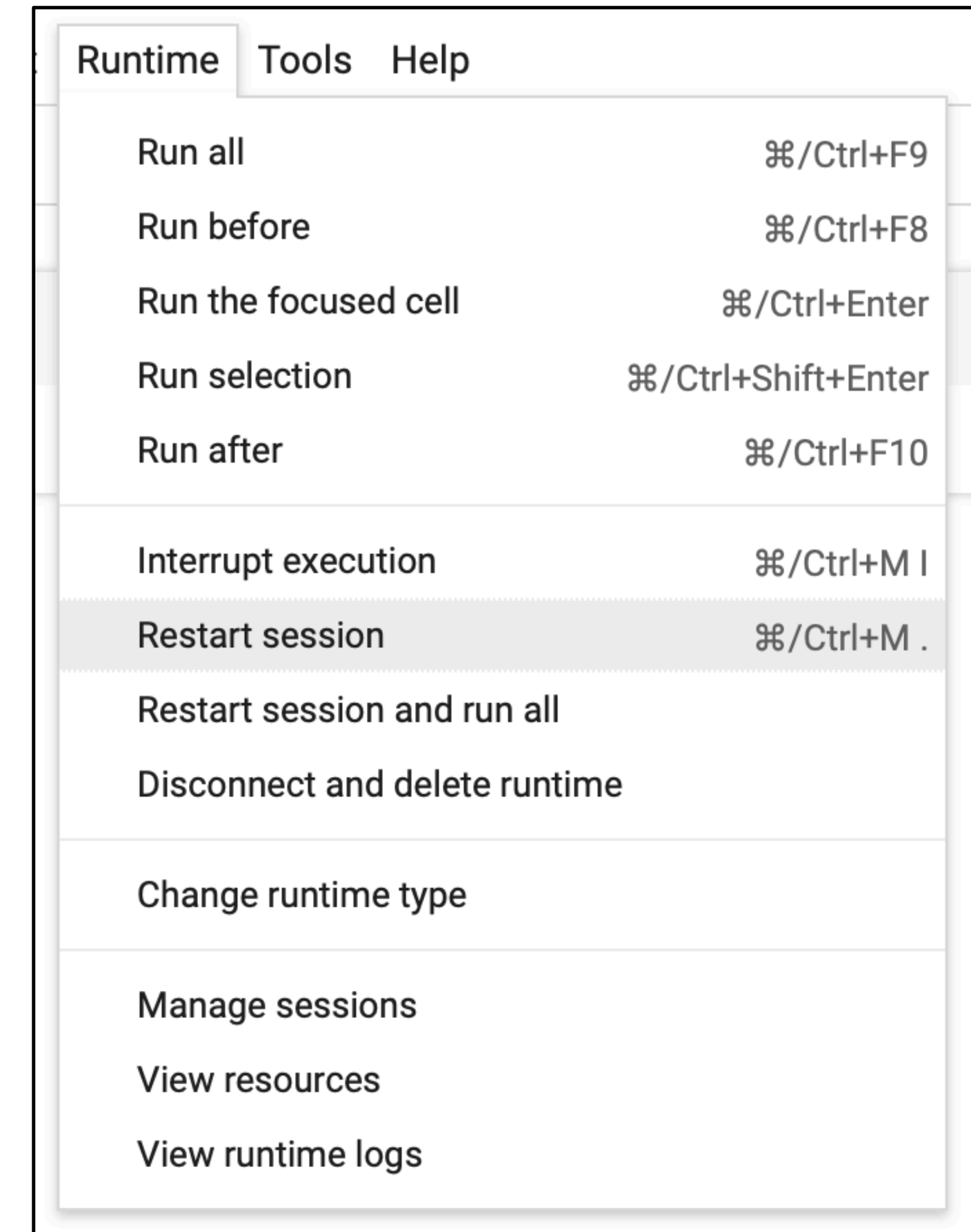
```
max = 9  
max(2, 3)
```

```
TypeError: 'int' object is not callable
```

*You broke Python. What now?*

If you want to restore names to their default values, do this:

1. Save your notebook
2. Restart your session



# Names and readability

*Including material from  
the end of last class*

There are a lot of factors for writing *good code*:

- your code runs

- your code runs quickly (efficiency)

- your code is concise (fewer lines is usually better)

- your code can easily be read by other people who know Python.

The last point is called *readability*.

“Programs must be written for people to read, and only incidentally for machines to execute.”

Hal Abelson & Gerald Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, 1979

Choosing good names

# Names are arbitrary

This is silly, but legal:

<i>five</i> = 6 five
6

<i>six</i> = 5 six
5



Several constants may have the same value:

<i>seven</i> = 7 seven
7

<i>septem</i> = 7 septem
7

Names in Python are *case-sensitive*.

So,

Cat  
CAT  
cat  
cAT

are all distinct names, which can have different associated values – but doing this is a bad idea because it's confusing!



*They look similar – but they're all distinct!*

Python is pretty flexible about what names can look like:

👍 `how_are_you`

👍 `my_AGE_is_22`

👍 `NETFLIXPASSWORD`

But it doesn't allow hyphens or other punctuation – only underscores are allowed:

👎 `this-is-bad`

👎 `worse!`

👎 `no&*way`

While names can include a number, like

👍 `pi_r_2`

They can't *start* with a number:

👎 `2_pi_r`

Prefer names that are concise:

*population\_of\_north\_carolina\_in\_2025* = 10975000

*Too long!*

*nc\_pop\_2025* = 10975000

*Much better!*

But make sure the names are descriptive of what you're trying to store in them.

*x* = 7173

*No idea what this is.*

*tb\_cases\_2023* = 7173

*Much better!*

Every programming language also has its own *conventions* for names.

In standard Python, names are usually lowercase with words joined by underscores, e.g.,

*this\_is\_a\_good\_name*

*thisMakesPythonCRY*



# Names are important!

Can you guess what this code does?

```
y = (x + 459.67) * 5/9
```



# Names are important!

Can you guess what this code does?

~~$y = (x + 459.67) * 5/9$~~

*temp\_kelvin* = (temp\_celsius + 459.67) \* 5/9

# Booleans and comparison expressions

True

False

We can compare values using the operators

`==` equal to

`!=` not equal to

`<` less than

`<=` less than or equal to

`>` greater than

`>=` greater than or equal to

which produce `True` or `False` as a result.

Be careful:

*x* = 2

is assigning the name *x* to have the value **2** in the directory.

On the other hand,

*x* == 2

is asking the question “is *x* equal to **2**?”

Boolean expressions can also be combined using the operators

**and**

**True** if both inputs are **True**;

**False** otherwise

**or**

**False** if both inputs are **False**;

**True** otherwise

$(1 < 2)$  and  $(2 > 3)$

$(1 < 2)$  and  $(2 > 3)$

→ **True** and  $(2 > 3)$



`(1 < 2) and (2 > 3)`

→ `True and (2 > 3)`

→ `True and False`

(1 < 2) and (2 > 3)

→ True and (2 > 3)

→ True and False

→ False

```
(1 <= 0) or (1 == 1)
```

`(1 <= 0) or (1 == 1)`

→ `False or (1 == 1)`

<code>(1 &lt;= 0) or (1 == 1)</code>
→ <code>False or (1 == 1)</code>
→ <code>False or True</code>

(1 <= 0) or (1 == 1)	
→	False or (1 == 1)
→	False or True
→	True

To change an expression that evaluates to **True** to be **False** – or vice versa – use the **not** operator:

```
not True
```

```
False
```

```
not 1 == 0
```

```
True
```

# Conditional statements



`if ...` is a *conditional statement*.

Conditionals allow us to *branch* – maybe we evaluate this expression; maybe we don't!

```
if 1 < 2:  
    print("All is right in the world")
```

*If the condition  
is true, the  
code indented  
under it is run.*

```
if 1 < 2:  
    print("All is right in the world")
```



```
if True:  
    print("All is right in the world")
```

*If the condition  
is true, the  
code indented  
under it is run.*

```
if 1 < 2:  
    print("All is right in the world")
```

→ 

```
if True:  
    print("All is right in the world")
```

→ 

```
print("All is right in the world")
```

*If the condition  
is true, the  
code indented  
under it is run.*

```
if 1 < 2:  
    print("All is right in the world")
```

→ 

```
if True:  
    print("All is right in the world")
```

→ 

```
print("All is right in the world")
```

**All is right in the world**

*If the condition  
is true, the  
code indented  
under it is run.*

```
if 1 > 2:  
    print("Watch out for flying pigs")
```

*If the condition  
is false, the  
code indented  
under it is  
skipped.*

```
if 1 > 2:  
    print("Watch out for flying pigs")
```



```
if False:  
    print("Watch out for flying pigs")
```

*If the condition  
is false, the  
code indented  
under it is  
skipped.*

```
if 1 > 2:  
    print("Watch out for flying pigs")
```



```
if False:  
    print("Watch out for flying pigs")
```



*If the condition  
is false, the  
code indented  
under it is  
skipped.*



```
if 1 > 2:  
    print("Watch out for flying pigs")  
  
print("Life goes on")
```

*Even if the  
condition is  
false, Python  
runs the code  
after it.*

```
if 1 > 2:  
    print("Watch out for flying pigs")  
  
print("Life goes on")
```



```
if False:  
    print("Watch out for flying pigs")  
  
print("Life goes on")
```

*Even if the condition is false, Python runs the code after it.*

```
if 1 > 2:  
    print("Watch out for flying pigs")  
  
print("Life goes on")
```

→

```
if False:  
    print("Watch out for flying pigs")  
  
print("Life goes on")
```

→

```
print("Life goes on")
```

*Even if the condition is false, Python runs the code after it.*

```
if 1 > 2:  
    print("Watch out for flying pigs")  
  
print("Life goes on")
```

→

```
if False:  
    print("Watch out for flying pigs")  
  
print("Life goes on")
```

→

```
print("Life goes on")
```

**Life goes on**

*Even if the condition is false, Python runs the code after it.*

Sometimes, you need a Plan B, so you can pair **if**  
with **else**.

```
if 1 < 2:  
    print("All is right in the world")  
else:  
    print("Watch out for flying pigs")
```

```
if 1 < 2:  
    print("All is right in the world")  
else:  
    print("Watch out for flying pigs")
```



```
if True:  
    print("All is right in the world")  
else:  
    print("Watch out for flying pigs")
```

```
if 1 < 2:  
    print("All is right in the world")  
else:  
    print("Watch out for flying pigs")
```

→

```
if True:  
    print("All is right in the world")  
else:  
    print("Watch out for flying pigs")
```

→

```
print("All is right in the world")
```



```
if 1 < 2:  
    print("All is right in the world")  
else:  
    print("Watch out for flying pigs")
```

→

```
if True:  
    print("All is right in the world")  
else:  
    print("Watch out for flying pigs")
```

→

```
print("All is right in the world")
```

**All is right in the world**

```
if 1 > 2:  
    print("All is right in the world")  
else:  
    print("Watch out for flying pigs")
```

```
if 1 > 2:  
    print("All is right in the world")  
else:  
    print("Watch out for flying pigs")
```



```
if False:  
    print("All is right in the world")  
else:  
    print("Watch out for flying pigs")
```

```
if 1 > 2:  
    print("All is right in the world")  
else:  
    print("Watch out for flying pigs")
```

→

```
if False:  
    print("All is right in the world")  
else:  
    print("Watch out for flying pigs")
```

→

```
print("Watch out for flying pigs")
```

```
if 1 > 2:  
    print("All is right in the world")  
else:  
    print("Watch out for flying pigs")
```

→

```
if False:  
    print("All is right in the world")  
else:  
    print("Watch out for flying pigs")
```

→

```
print("Watch out for flying pigs")
```

**Watch out for flying pigs**

And if we want to play Twenty Questions, we can keep going, adding **elif** (“else if”) to our **if–else**.



Recall how you defined functions in middle-school math:

$$\text{Given } f(x) = |x| + 2$$

$$f(-3) = |-3| + 2$$

$$= 3 + 2$$

$$= 5$$



*The parameter  $x$  stands  
for varying values*



Python functions work much the same way:

```
def f(x): return abs(x) + 2
```

Python functions work much the same way:

```
def f(x): return abs(x) + 2
```

```
f(-3)
```

Python functions work much the same way:

```
def f(x): return abs(x) + 2
```

```
f(-3)
```

*Directory*

<i>Directory</i>	
<b><i>Name</i></b>	<b><i>Value</i></b>
<b><i>x</i></b>	<b>-3</b>

Python functions work much the same way:

```
def f(x): return abs(x) + 2
```

f(-3)

→ abs(x) + 2

*Directory*

<i>Directory</i>	
<i>Name</i>	<i>Value</i>
<i>x</i>	-3

Python functions work much the same way:

```
def f(x): return abs(x) + 2
```

f(-3)

→ abs(x) + 2

→ abs(-3) + 2

*Directory*

<i>Directory</i>	
<i>Name</i>	<i>Value</i>
<i>x</i>	-3

Python functions work much the same way:

```
def f(x): return abs(x) + 2
```

f(-3)

→ abs(x) + 2

→ abs(-3) + 2

→ 3 + 2

*Directory*

<i>Directory</i>	
<i>Name</i>	<i>Value</i>
<i>x</i>	-3

Python functions work much the same way:

```
def f(x): return abs(x) + 2
```

f(-3)

→ abs(x) + 2

→ abs(-3) + 2

→ 3 + 2

→ 5

*Directory*

<i>Directory</i>	
<i>Name</i>	<i>Value</i>
<i>x</i>	-3





