CMPU 100 · Programming with Data

Asking Questions

Class 3

Assignment 1

Out: Tomorrow 5:00 p.m.

Due: Next Thursday 11:59 p.m.

cs.vassar.edu/~cs100/resources/coaching.html

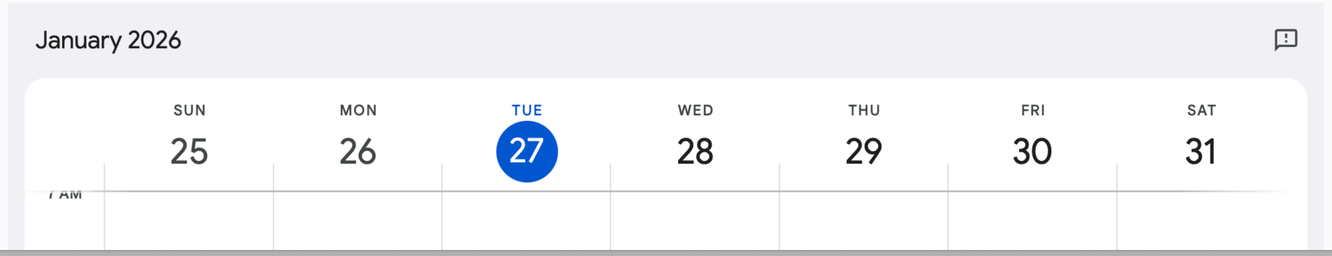
CMPU 100 / Resources / Coaching

⌂ LABS ASSIGNMENTS RESOURCES GRADESCOPE

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!



Autograder Results

Results | Code

Public Tests

```
Exercise 1: within_five results: All test cases passed!

Exercise 2: hobby_relates results: All test cases passed!

Exercise 3: in_range results:
    Exercise 3: in_range - 1 result:
        ✅ Test case passed

    Exercise 3: in_range - 2 result:
        ✅ Test case passed

    Exercise 3: in_range - 3 result:
        ❌ Test case failed
        Trying:
            in_range("", True)
        Expecting:
            False
        **********************************************************************
        Line 2, in Exercise 3: in_range 2
        Failed example:
            in_range("", True)
        Expected:
            False
        Got:
            True
```

Where are we?

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

```
abs(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

Name	***Value***
total	

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

```
   total = 2 + 3
→  total = 5
```

Directory

Name	***Value***
total	

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

```
   total = 2 + 3
→  total = 5
```

Directory

Name	***Value***
total	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***
total	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***
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
```

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
```

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***
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***
total	5
new_total	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!

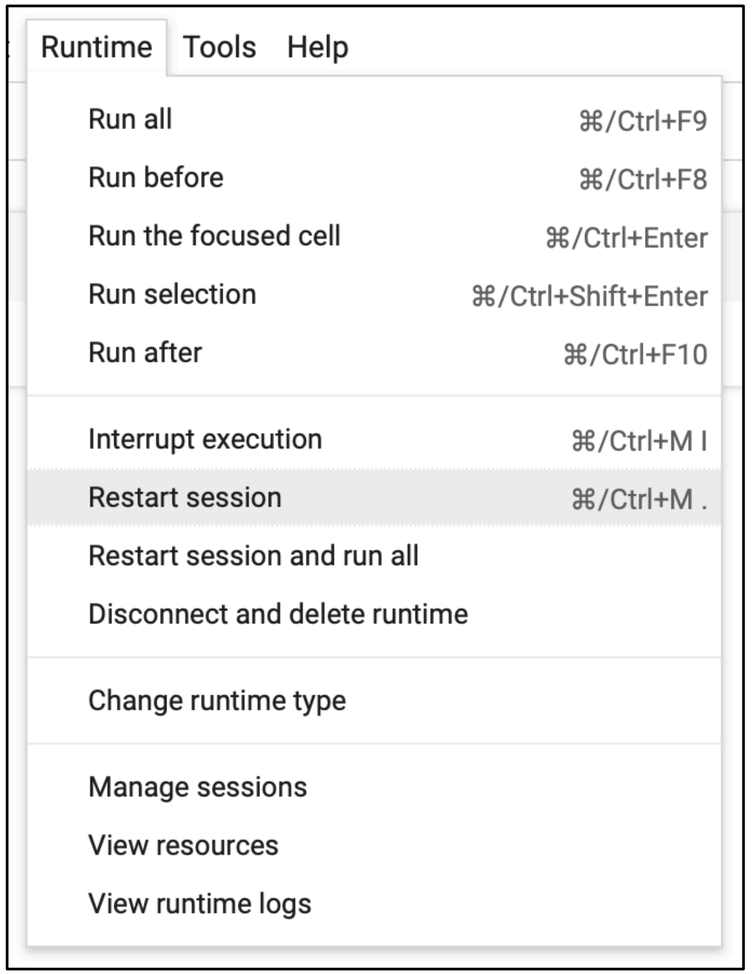
```
max = 9
max(2, 3)
```

```
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



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**?”

Notebook: *Booleans and comparison expressions*

Boolean expressions can also be combined using the operators

`and`

`True` if both inputs are `True`;

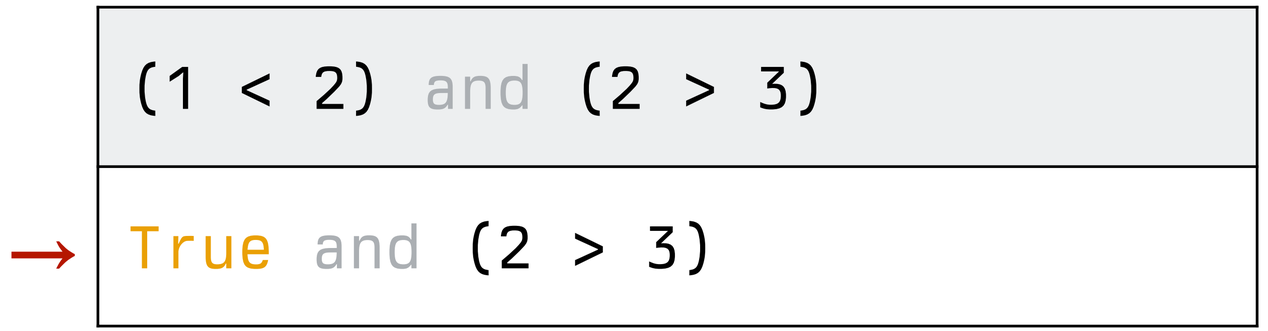
`False` otherwise

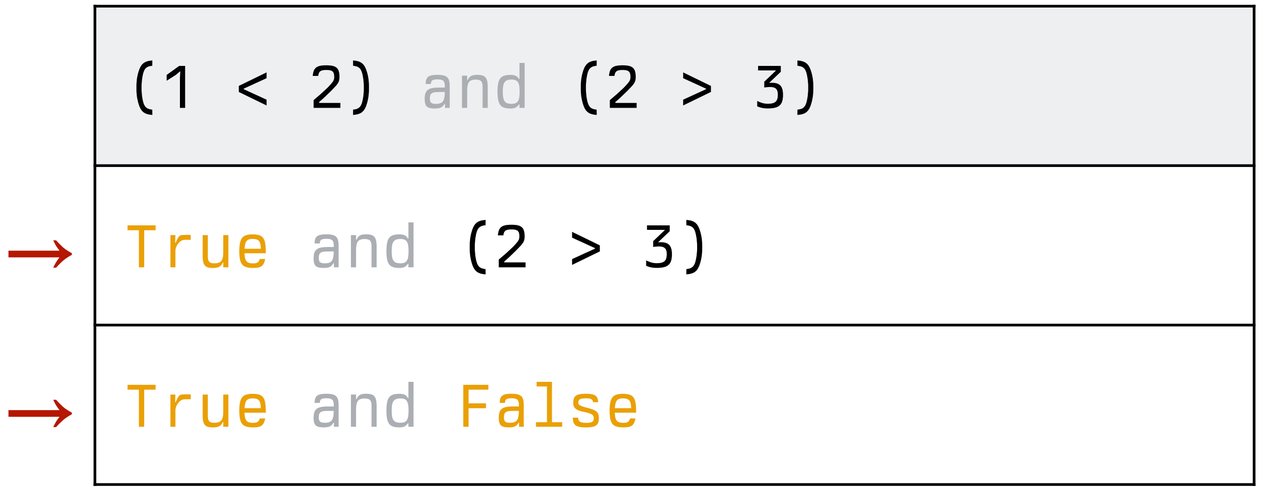
`or`

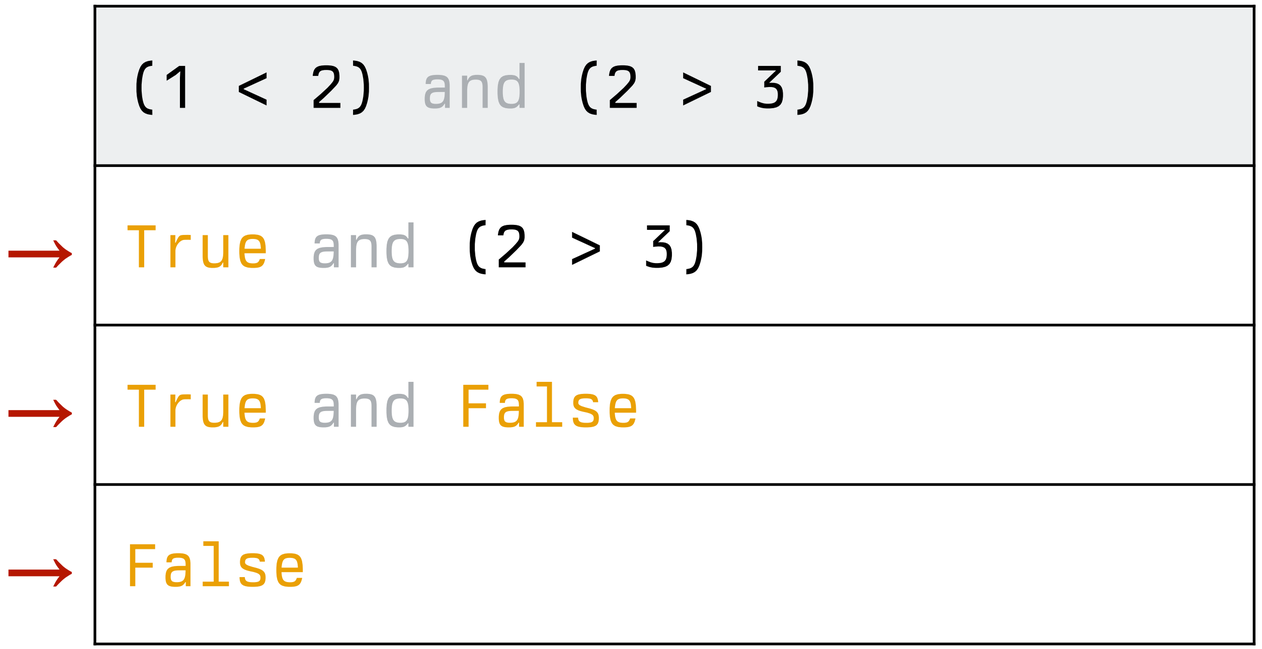
`False` if both inputs are `False`;

`True` otherwise

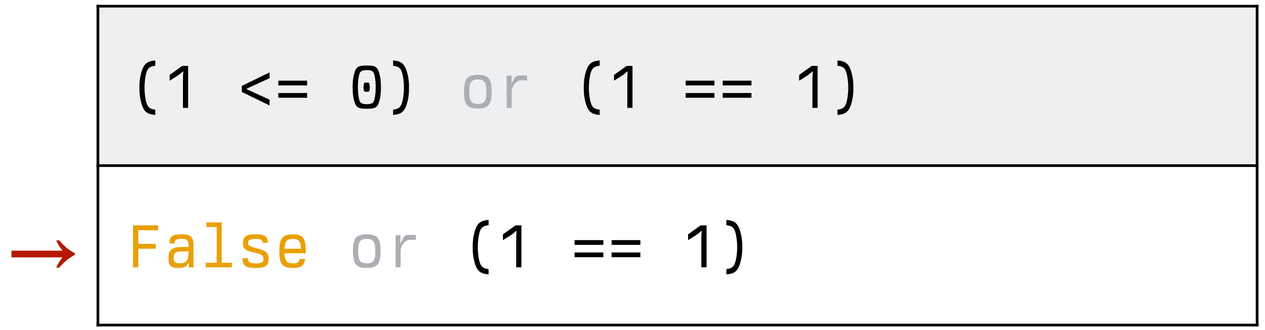
```
(1 < 2) and (2 > 3)
```

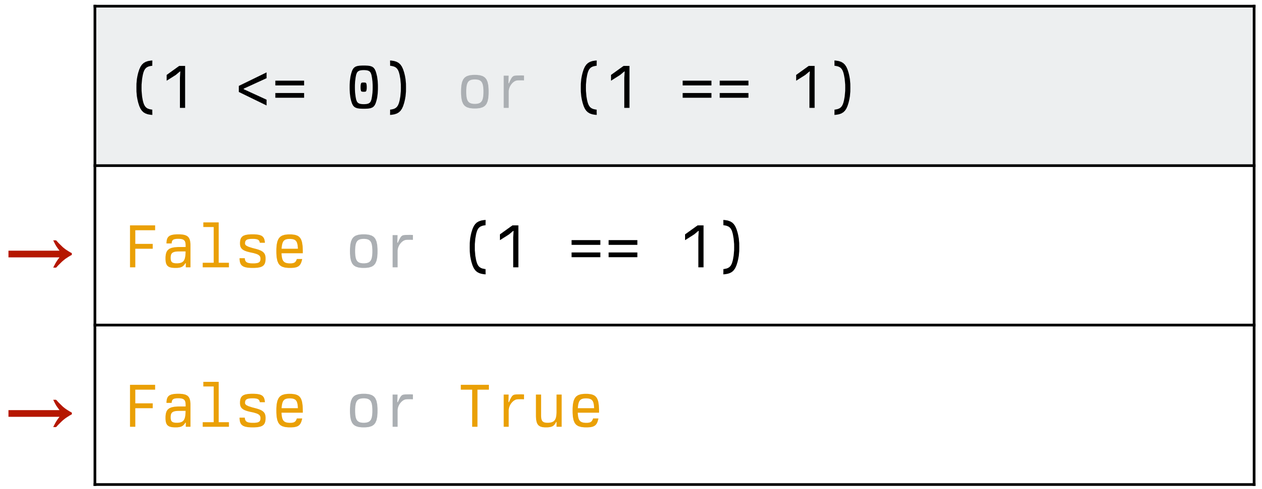

(1 < 2) and (2 > 3)
True and (2 > 3)

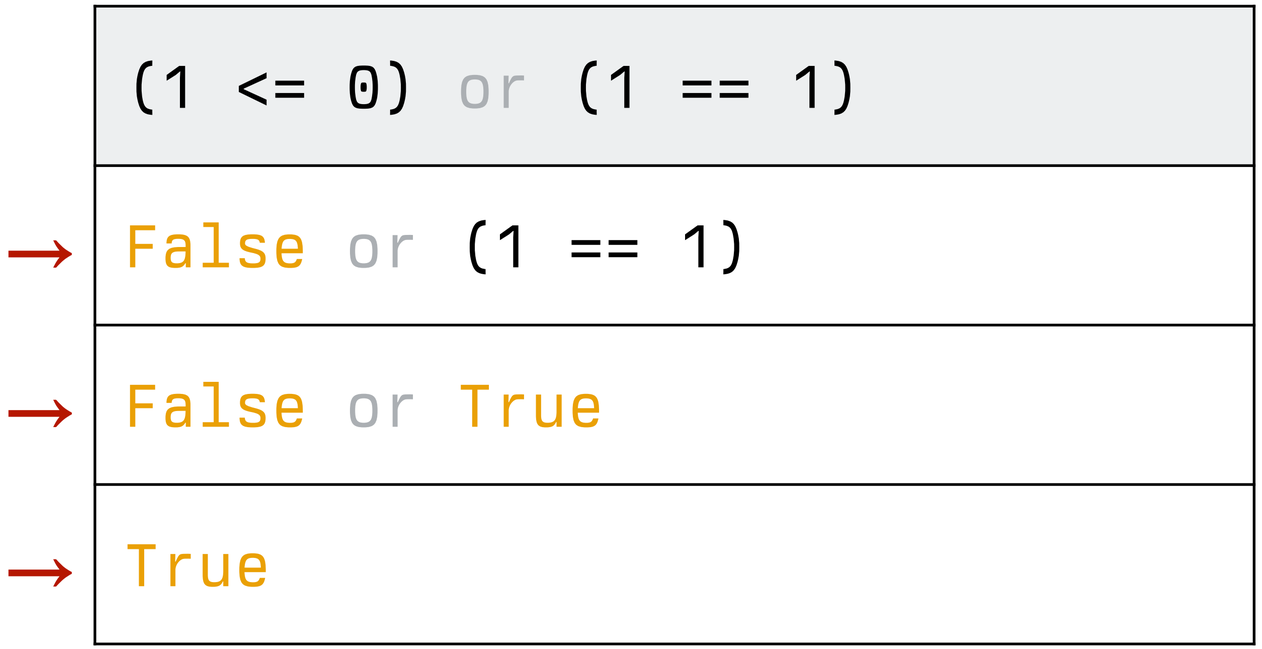

(1 < 2) and (2 > 3)
True and (2 > 3)
True and False


(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)


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


(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
```

Notebook: *Practice*

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`**.

Notebook: *Conditional statements*

Preview: Defining functions

Recall how you defined functions in middle-school math:

Given $f(x) = |x| + 2$

The parameter x stands for varying values

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

$= 3 + 2$

$= 5$

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

Name	***Value***
x	-3

Python functions work much the same way:

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

f(-3)
→ abs(x) + 2
```

Directory

Name	***Value***
x	-3

Python functions work much the same way:

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

f(-3)
→ abs(x) + 2
→ abs(-3) + 2
```

Directory

Name	***Value***
x	-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

Name	***Value***
x	-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

Name	***Value***
x	-3

Notebook: *Preview: Defining functions*

Acknowledgments

This class incorporates material from:

Kathi Fisler, Brown University

Katie Keith and Steve Freund, Williams College

Data 6, University of California, Berkeley (CC BY-NC-SA)

Data 8, University of California, Berkeley (CC BY-NC-SA)