Functions
What we'll discuss today...

- Zoom rules
- Values
- Expressions
- Functions
The Rules of the Zoom

- Chat will be enabled. Please hide it and/or disable notifications if distracting.
- Chat is for questions & comments on the current topic.
- If the chat goes off-topic, we'll ask you to focus or we'll disable it. 😐
- You can also post questions in the Zoom Q&A.
- If you have unanswered or tangential questions, post in the Piazza Q&A thread.
Community guidelines

Your goal should be to learn and help others learn.

Even if everyone here has programming experience, there is still a wide range of experience levels. All are welcome!

There are no "stupid" questions. Ask all your questions and welcome everyone else's questions.
Expressions & Values
What do programs do?

- Programs work by manipulating **values**
- **Expressions** in programs evaluate to values
  - Expression: `'a' + 'hoy'`
  - Value: `'ahoy'`
- The Python interpreter evaluates expressions and displays their values
Values

Programs manipulate **values**.

Each value has a certain **data type**.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Example values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers</td>
<td>2 44 -3</td>
</tr>
<tr>
<td>Floats</td>
<td>3.14 4.5 -2.0</td>
</tr>
<tr>
<td>Booleans</td>
<td>True False</td>
</tr>
<tr>
<td>Strings</td>
<td>'¡hola!' 'its python time!'</td>
</tr>
</tbody>
</table>

Try in a Python interpreter, like on [code.cs61a.org](http://code.cs61a.org).
Expressions (with operators)

An expression describes a computation and evaluates to a value.

Some expressions use operators:

18 + 69

6/23

2 * 100

2 ** 100

Try in a Python interpreter, like on code.cs61a.org.
Call expressions

Many expressions use function calls:

\[ \text{pow}(2, 100) \]

\[ \text{max}(50, 300) \]

\[ \text{min}(-1, -300) \]
Expressions (both ways)

Expressions with operators can also be expressed with function call notation:

```plaintext
2 ** 100
pow(2, 100)
```
Expressions (both ways)

Expressions with operators can also be expressed with function call notation:

```
2 ** 100
pow(2, 100)
```

```
from operator import add

18 + 69
add(18, 69)
```

The `pow()` function is a **built-in**; it's provided in every Python environment. Other functions (**add()**, **div()**, etc) must be imported from the **operator** module in the Python standard library.
Anatomy of a Call Expression

```
add ( 18 , 69 )
```

How Python evaluates a call expression:
Anatomy of a Call Expression

Addition (18, 69)

Operator

How Python evaluates a call expression:

1. Evaluate the operator
Anatomy of a Call Expression

\[
\text{add} \quad ( \ \ 18 \quad , \quad 69 \quad )
\]

Operator \quad Operand \quad Operand

How Python evaluates a call expression:

1. Evaluate the operator
2. Evaluate the operands
Anatomy of a Call Expression

\[
\text{add} \quad ( \quad 18 \quad , \quad 69 \quad )
\]

Operator \quad Operand \quad Operand

How Python evaluates a call expression:

1. Evaluate the \text{operator}
2. Evaluate the \text{operands}
3. Apply the \text{operator (a function)} to the evaluated \text{operands (arguments)}
Anatomy of a Call Expression

\[ \text{add} \ ( \ 18 \ , \ 69 \ ) \]

Operator \hspace{2em} Operand \hspace{2em} Operand

How Python evaluates a call expression:

1. Evaluate the **operator**
2. Evaluate the **operands**
3. Apply the **operator (a function)** to the evaluated operands (arguments)

Operators and operands are also expressions, so they must be evaluated to discover their values.
Evaluating nested expressions

\[
\text{add}(\text{add}(6, \text{mul}(4, 6)), \text{mul}(3, 5))
\]
Evaluating nested expressions

```
add(add(6, mul(4, 6)), mul(3, 5))
```

```
add
```
Evaluating nested expressions

\[ \text{add}(\text{add}(6, \text{mul}(4, 6)), \text{mul}(3, 5)) \]
Evaluating nested expressions

\[ \text{add}(\text{add}(6, \text{mul}(4, 6)), \text{mul}(3, 5)) \]
Evaluating nested expressions

\[
\text{add}\left(\text{add}(6, \text{mul}(4, 6)), \text{mul}(3, 5)\right)
\]
Evaluating nested expressions

```
add(add(6, mul(4, 6)), mul(3, 5))
```

```
add
  add
    add
      6
      mul(4, 6)
```
Evaluating nested expressions

\[ \text{add}(\text{add}(6, \text{mul}(4, 6)), \text{mul}(3, 5)) \]
Evaluating nested expressions

\[ \text{add}(\text{add}(6, \text{mul}(4, 6)), \text{mul}(3, 5)) \]

\[ \text{add}(\text{add}(6, \text{mul}(4, 6))) \]

\[ \text{add} \]
\[ 6 \]
\[ \text{mul}(4, 6) \]
\[ \text{mul} \]
\[ 4 \]
Evaluating nested expressions

\[ \text{add}(\text{add}(6, \text{mul}(4, 6)), \text{mul}(3, 5)) \]
Evaluating nested expressions

\[
\text{add} (\text{add}(6, \text{mul}(4, 6)), \text{mul}(3, 5))
\]
Evaluating nested expressions

\[\text{add}(\text{add}(6, \text{mul}(4, 6)), \text{mul}(3, 5))\]
Evaluating nested expressions

```
add(add(6, mul(4, 6)), mul(3, 5))
```

```
30
```

```
add
add(6, mul(4, 6))
add
6
mul(4, 6)
mul
4
6
mul(3, 5)
```
Evaluating nested expressions

\[
add(add(6, mul(4, 6)), mul(3, 5))
\]
Evaluating nested expressions

```
add(add(6, mul(4, 6)), mul(3, 5))
```

Evaluation steps:
1. `mul(4, 6) = 24`
2. `add(6, 24) = 30`
3. `add(30, mul(3, 5)) = 30 + 15 = 45`

Result: 45
Evaluating nested expressions

\[ \text{add}(\text{add}(6, \text{mul}(4, 6)), \text{mul}(3, 5)) \]

Diagram:

```
add
  /
add 30
  /
add(6, mul(4, 6))
    /
add 6
    /
mul(4, 6)
      /
mul 4 6
mul
  /
mul(3, 5)
    /
mul 3 5
```
Evaluating nested expressions

\[
\text{add}\left(\text{add}\left(6, \text{mul}(4, 6)\right), \text{mul}(3, 5)\right)
\]

\[
\text{add}\left(6, \text{mul}(4, 6)\right)
\]

\[
\text{mul}(4, 6)
\]

\[
\text{add}\left(6, \text{mul}(4, 6)\right)
\]

\[
\text{mul}(3, 5)
\]

\[
\text{mul}(3, 5)
\]

\[
\text{mul}(4, 6)
\]

\[
30
\]

\[
24
\]

\[
15
\]
Evaluating nested expressions

\[ \text{add} \left( \text{add}(6, \text{mul}(4, 6)), \text{mul}(3, 5) \right) \]

\[ \text{add} \left( \begin{array}{c} 30 \\ \text{add}(6, \text{mul}(4, 6)) \end{array} \right), \begin{array}{c} 15 \\ \text{mul}(3, 5) \end{array} \right) \]

\[ \text{add} \left( \begin{array}{c} 30 \\ \text{add}(6, \text{add}(4, 6)) \end{array} \right), \begin{array}{c} 15 \\ \text{mul}(3, 5) \end{array} \right) \]

\[ \text{add} \left( \begin{array}{c} 24 \\ \text{add}(6, \text{mul}(4, 6)) \end{array} \right), \begin{array}{c} 15 \\ \text{mul}(3, 5) \end{array} \right) \]

\[ \text{add} \left( \begin{array}{c} 24 \\ \text{add}(6, \text{mul}(4, 6)) \end{array} \right), \begin{array}{c} 15 \\ \text{mul}(3, 5) \end{array} \right) \]

\[ \text{add} \left( \begin{array}{c} 24 \\ \text{add}(6, \text{mul}(4, 6)) \end{array} \right), \begin{array}{c} 15 \\ \text{mul}(3, 5) \end{array} \right) \]

\[ \text{add} \left( \begin{array}{c} 24 \\ \text{add}(6, \text{mul}(4, 6)) \end{array} \right), \begin{array}{c} 15 \\ \text{mul}(3, 5) \end{array} \right) \]
Evaluating nested expressions

This is called an expression tree.
Exercise: Expressions

After the lecture, you can try out this exercise. (Not graded, just another way to engage with the material!)
Names
Names

A **name** can be bound to a value.

One way to bind a name is with an **assignment statement**:

\[
\begin{array}{c c}
  x & = & 7 \\
  \text{Name} & \text{Value}
\end{array}
\]
Names

A **name** can be bound to a value.

One way to bind a name is with an **assignment statement**:

\[
x = 7
\]

The value can be any expression:

\[
x = 1 + 2 \times 3 - 4 \div 5
\]
Using names

A name can be referenced multiple times:

```plaintext
x = 10
y = 3

result1 = x * y
result2 = x + y
```
Using names

A name can be referenced multiple times:

\[
\begin{align*}
x &= 10 \\
y &= 3 \\
\text{result1} &= x \times y \\
\text{result2} &= x + y
\end{align*}
\]

A name that's bound to a data value is also known as a **variable**.
Name rebinding

A name can only be bound to a single value.

```python
my_name = 'Pamela'
my_name = my_name + 'ela'
```

Will that code error? If not, what will `my_name` store?
Name rebinding

A name can only be bound to a single value.

```python
my_name = 'Pamela'

my_name = my_name + 'ela'
```

Will that code error? If not, what will `my_name` store? It will not error (similar code in other languages might, however). The name `my_name` is now bound to the value 'Pamelaela'.

Exercise

Try this after the lecture...

What will be the value of the final expression in this sequence?

\[
f = \min \\
f = \max \\
g = \min \\
h = \max \\
\text{max} = g \\
\text{max}(f(2, g(h(1, 5), 3)), 4)
\]
Environment diagrams
Environment diagrams

An environment diagram is a visualization of how Python interprets a program. Use the free website PythonTutor to generate diagrams. View example

<table>
<thead>
<tr>
<th>Code (left)</th>
<th>Frames (right)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( x = 1 )</td>
<td>Global frame</td>
</tr>
<tr>
<td>2 ( y = x )</td>
<td>( x \ 1 )</td>
</tr>
<tr>
<td>3 ( x = 2 + x )</td>
<td>( y \ 1 )</td>
</tr>
<tr>
<td>4 ( z = x + y )</td>
<td></td>
</tr>
</tbody>
</table>

Arrows indicate the order of execution. Green = just executed, red = up next. Each name is bound to a value. Within a frame, each name cannot be repeated.
Assignments in Environment diagrams

How Python interprets an assignment statement:

- Evaluate the expression to the right of \( = \).
- Bind the expression's value to the name that's on the left side of the \( = \) sign.

```plaintext
1   x = 1
2   y = x
3   x = 2 + x
4   z = x + y
```

View in PythonTutor
Functions
What is a function?

A **function** is a sequence of code that performs a particular task and can be easily reused. 🔄

We've already used functions:

```plaintext
add(18, 69)
mul(60, sub(5, 4))
```
What is a function?

A **function** is a sequence of code that performs a particular task and can be easily reused.

We've already used functions:

```text
add(18, 69)
mul(60, sub(5, 4))
```

A function takes inputs (the **arguments**) and returns an output (the **return value**).

18, 69 → add → 87
Defining functions

The most common way to define functions is Python is the `def` statement.

```python
def <name>(<parameters>):
    return <return expression>
```

Example:

```python
def add(num1, num2):
    return num1 + num2
```

Once defined, we can call it:

```python
add(2, 2)
add(18, 69)
```
Anatomy of a function definition

The first line is called the **function signature**, all lines after are considered the **function body**.

```python
def <name>(<parameters>):  # ← Function signature
    return <return expression>  # ← Function body

def add(num1, num2):  # ← Function signature
    return num1 + num2  # ← Function body
```
Anatomy of a function definition

The first line is called the **function signature**, all lines after are considered the **function body**.

```python
def <name>(<parameters>):  # ← Function signature
    return <return expression>  # ← Function body
```

```python
def add(num1, num2):
    return num1 + num2  # ← Function signature
``` 

The function body can have multiple lines:

```python
def add(num1, num2):
    sum = num1 + num2  # ← Function signature
    return sum  # ← Function body
```
Function arguments

We can pass in any expressions as arguments.

```python
def add(num1, num2):
    return num1 + num2

x = 1
y = 2
add(x, y)

x = 3
add(x * x, x + x)
```

Example with strings
Return values

The return keyword returns a value to whoever calls the function (and exits the function).

```python
def add(num1, num2):
    return num1 + num2

sum = add(2, 4)
```

Reminder: You can use function calls in expressions:

```python
big_sum = add(200, 412) + add(312, 256)
```

...and nest function calls inside function calls:

```python
huge_sum = add(add(200, 412), add(312, 256))
```
Spot the bug #1

What's wrong with this code?

def add(num1, num2):
    return sum
    sum = num1 + num2

sum = add(2, 4)
Spot the bug #1

What's wrong with this code?

```python
def add(num1, num2):
    return sum
    sum = num1 + num2

sum = add(2, 4)
```

The code after the return statement will not be executed, that line belongs before the return.
Spot the bug #2

What's wrong with this code?

def add():
    return num1 + num2

sum = add(2, 4)
Spot the bug #2

What's wrong with this code?

```python
def add():
    return num1 + num2

sum = add(2, 4)
```

The function body is referring to variables that don't seem to exist. Most likely, they should be parameters in the function signature.
Spot the bug #3

What's wrong with this code?

```python
def add(num1, num2):
    sum = num1 + num2

sum = add(2, 4)
```
Spot the bug #3

What's wrong with this code?

```python
def add(num1, num2):
    sum = num1 + num2

sum = add(2, 4)
```

The function body does not return any value. However, the code that calls it tries to use the result of the expression. It should have a return statement that returns the sum.
Functions in environment diagrams

How Python interprets a def statement:

- It creates a function with the **name** and **parameters**
- It sets the function body to everything indented after the first line
- It binds the function name to that function body (similar to an assignment statement)

```python
1 def add(num1, num2):
2     sum = num1 + num2
3     return sum
4
5 result = add(2, 4)
```

View in PythonTutor
Function calls in environment diagrams

How Python interprets a function call:

- It creates a new **frame** in the environment
- It binds the function call's arguments to the parameters in that frame
- It executes the body of the function in the new frame
```python
1 def add(num1, num2):
2     sum = num1 + num2
3     return sum
4
5 result = add(2, 4)
```

**Edit this code**

- /line that just executed
- /next line to execute

![View in PythonTutor](python_tutor_icon.png)
More on names
Names and environments

All Python code is evaluated in the context of an environment, which is a sequence of frames.

We've seen two possible environments:

Global frame

Function's local frame, child of Global frame
Name lookup rules

How Python looks up names in a user-defined function:

1. Look it up in the local frame
2. If name isn't in local frame, look it up in the global frame
3. If name isn't in either frame, throw a NameError

*This is simplified since we haven't learned all the Python features that complicate the rules.*
Name lookup example #1

```python
def exclamify(text):
    start_exclam = "¡"
    end_exclam = "!
    return start_exclam + text + end_exclam

exclamify("the snails are eating my lupines")
```

- On line 4, which frame is `start_exclam` found in?
- On line 4, Which frame is `text` found in?
- On line 6, which frame is `exclamify` found in?

View in PythonTutor
Name lookup example #1

def exclamify(text):
    start_exclaim = "¡"
    end_exclaim = "!"
    return start_exclaim + text + end_exclaim

exclamify("the snails are eating my lupines")

• On line 4, which frame is `start_exclaim` found in?  
The local frame for exclamify
• On line 4, Which frame is `text` found in?

• On line 6, which frame is `exclamify` found in?

View in PythonTutor
def exclamify(text):
    start_exclaim = "¡"
    end_exclaim = "!
    return start_exclaim + text + end_exclaim
exclamify("the snails are eating my lupines")

• On line 4, which frame is **start_exclaim** found in?
  The local frame for exclamify
• On line 4, Which frame is **text** found in?
  The local frame for exclamify
• On line 6, which frame is **exclamify** found in?

View in PythonTutor
def exclaimify(text):
    start_exclaim = "¡"
    end_exclaim = "!"
    return start_exclaim + text + end_exclaim

exclaimify("the snails are eating my lupines")

• On line 4, which frame is start_exclaim found in?
  The local frame for exclaimify
• On line 4, Which frame is text found in?
  The local frame for exclaimify
• On line 6, which frame is exclaimify found in?
  The global frame

View in PythonTutor
Name lookup example #2

```python
start_exclaim = "¡"
end_exclaim = "❣️"

def exclamify(text):
    return start_exclaim + text + end_exclaim

exclamify("the voles are digging such holes")
```

- On line 5, which frame is `start_exclaim` found in?
- On line 5, Which frame is `text` found in?
- On line 6, which frame is `exclamify` found in?

View in PythonTutor
Name lookup example #2

```python
start_exclaim = "¡"
end_exclaim = "❣️"

def exclamify(text):
    return start_exclaim + text + end_exclaim

exclamify("the voles are digging such holes")
```

- On line 5, which frame is `start_exclaim` found in? The global frame
- On line 5, Which frame is `text` found in?
- On line 6, which frame is `exclamify` found in?

[View in PythonTutor]
Name lookup example #2

```python
start_exclaim = "¡"
end_exclaim = "❣️"

def exclamify(text):
    return start_exclaim + text + end_exclaim

exclamify("the voles are digging such holes")
```

- On line 5, which frame is `start_exclaim` found in?
  The global frame
- On line 5, Which frame is `text` found in?
  The local frame for exclamify
- On line 6, which frame is `exclamify` found in?

View in PythonTutor
Name lookup example #2

```python
start_exclaim = "¡"
end_exclaim = "❣️"

def exclamify(text):
    return start_exclaim + text + end_exclaim

exclamify("the voles are digging such holes")
```

- On line 5, which frame is `start_exclaim` found in? The global frame
- On line 5, which frame is `text` found in? The local frame for `exclamify`
- On line 6, which frame is `exclamify` found in? The global frame

![View in PythonTutor](View in PythonTutor)
Name lookup example #3

```python
def exclamify(text):
    end_exclaim = "!?"
    return start_exclaim + text + end_exclaim
exclamify("the voles are digging such holes")
```

- Which name will cause a **NameError**?

- When will that error happen?

[View in PythonTutor](#)
Name lookup example #3

```python
def exclamify(text):
    end_exclaim = "!?"
    return start_exclaim + text + end_exclaim

exclamify("the voles are digging such holes")
```

- Which name will cause a `NameError`? The `start_exclaim` name, since it was never assigned.
- When will that error happen?

![View in PythonTutor]
def exclamify(text):
    end_exclaim = "!\?"
    return start_exclaim + text + end_exclaim

exclamify("the voles are digging such holes")

- Which name will cause a NameError? The start_exclaim name, since it was never assigned.
- When will that error happen? It will happen when exclamify is called and Python tries to execute the return statement.

View in PythonTutor
Summary

• Programs consist of **statements**, or instructions for the computer, containing **expressions**, which describe computation and evaluate to values.

• **Values** can be assigned to **names** to avoid repeating computations.

• An **assignment statement** assigns the value of an expression to a name in the current **environment**.

• **Functions** encapsulate a series of statements that maps **arguments** to a **return value**.

• A **def statement** creates a function object with certain **parameters** and a **body** and binds it to a name in the current environment.

• A **call expression** applies the value of its **operator**, a function, to the value(s) or its **operand(s)**, some arguments.
Exercises

You can try these exercises after the lecture for some additional practice:

- Operator expressions
- Fortune Teller
- Dog Age
- Lifetime Supply
- Temperature Converter

To run the doctests, press the red test tube in the upper right corner.