http://www.cs.cornell.edu/courses/cs1110/2022sp

# Lecture 2:
# Variables & Assignments

(2.1-2.3,2.5, 2.6 or videos (see Schedule))

## CS 1110
## Introduction to Computing Using Python

Cornell Bowers C·IS
Computer Science

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]

http://www.cs.cornell.edu/courses/cs1110/2022sp

# Lecture 2:
# Variables & Assignments
## (Sections 2.1-2.3,2.5, 2.6)

*Have pencil and paper (or stylus and tablet) ready. We'll do visualization exercises that involve drawing diagrams today.*

*Recommendations for note taking:*
- *Print out posted lecture slides and write on them*
- *Have the slides pdf ready and annotate electronically*

# Lecture Afterthoughts

There were many questions about what certain operators do (/, //, %). You do not need to memorize their behavior. We want you to know about them so that *when the need arises*, you can make use of them.

Similarly, we want you to know that operator precedence *exists* so you can understand how Python works. Instead of memorizing these slides, you can reference the exact ordering when it matters to the code you are writing.

# Helping you succeed in this class

http://www.cs.cornell.edu/courses/cs1110/2022sp/staff/

**Ed Discussions.** Online forum to ask/answer questions

**Consulting/Office Hours.**

- See calendar for which are 1-on-1 help (managed by QueueMeIn") versus "public help" (all students there form a single audience)

**Prof Office Hours (on same calendar)**

- After lecture: public help.
- Bookable 1-on-1 appointments with Professor Bracy
- Bookable 1-on-1 appointments with Professor Lee

**AEW (ENGRG 1010).** "Academic Excellence Workshops"

- *Optional* discussion course that runs parallel to this class. See website for more info

# Lab 1 Activities

**Activity 1:** if you aren't passing the "banner" question, get help with your installation! *In the meantime,* you can add the following string to your answer to get the system to accept it:

`Python 3.x Anaconda`

**Activity 2:** the password is:

**learn.by.testing.hypotheses**

(no spaces, all one "word", period separators)

**Activity 3:**

**Q3:** add the name of Python's behavior to your answer:

**short circuit evaluation**

**Q8:** password:

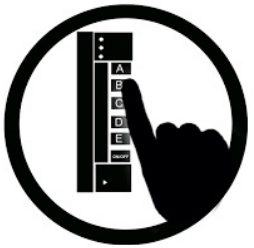
**shortcircuit** (no spaces)

# Which of the following is **false**?

A **type**…

(a) is a set of values & operations on these values
(b) represents something
(c) can be determined by using `type()` in Python
(d) can be changed by using `type()` in Python
(e) determines the meaning of an operation

*If there are multiple false answers, pick one!*



# From last time: **Types**

## Type: set of values & operations on them

Type **float:**

- Values: real numbers
- Ops: +, -, *, /,//, **,%

Type **int:**

- Values: integers
- Ops: +, -, *, //, %, **

Type **bool:**

- Values: `True`, `False`
- Ops: `not`, `and`, `or`

Type **str:**

- Values: strings
  - Double quotes: `"abc"`
  - Single quotes: `'abc'`
- Ops: + (concatenation)

# Converting from one type to another

aka "casting"

```
>>> float(2)
2.0

>>>int(2.6)
2



>>>type(2)
<class 'int'>
```

converts value `2` to type `float`

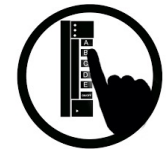
converts value `2.6` to type `int`

…different from:

`type(<value>)`

which *tells you* the type

# What does Python do?



```
>>> 1/2.6
```

(A) turn 2.6 into the integer 2, then calculate 1/2 → 0.5

(B) turn 2.6 into the integer 2, then calculate 1//2 → 0

(C) turn 1 into the float 1.0, then calculate 1.0/2.6 → 0.3846…

(D) Produce a `TypeError` telling you it cannot do this.

(E) Exit Python

# Widening Conversion (OK!)

From a **narrower** type to a **wider** type
(e.g., `int` → `float`)

Width refers to information capacity. "Wide" → more information capacity

Python does automatically if needed:

- Example: `1/2.0` evaluates to a `float`: `0.5`

From narrow to wide:
**`bool`** → **`int`** → **`float`**

Note: does not work for **`str`**

- Example: `2 + "ab"` produces a `TypeError`

# Narrowing Conversion (is it OK???)

From a **wider** type to a **narrower** type

(e.g., `float` → `int` )

- causes information to be lost
- Python **never** does this automatically

What about:

```
>>> 1/int(2.6)
0.5
```

*Python casts the 2.6 to 2 but / is a float division, so Python casts 1 to 1.0 and 2 to 2.0*

# Types matter!

You Decide:

- What is the right type for my data?
- When is the right time for conversion (if any)

- Zip Code as an `int`?
- Grades as an `int`?
- Lab Grades as a `bool`?
- Interest level as `bool` or `float`?

# Operator Precedence

What is the difference between:

| 2*(1+3) | 2*1 + 3 |
| --- | --- |
| ***add, then multiply*** | ***multiply, then add*** |

Operations performed in a set order

- Parentheses make the order explicit

What if there are no parentheses?

→ **Operator Precedence:** fixed order to process operators when no parentheses

# Precedence of Python Operators

- **Exponentiation**: **
- **Negation** : –
- **Binary arithmetic**: * / // %
- **Binary arithmetic**: + –
- **Comparisons**: < > <= >=
- **Equality relations**: == !=
- **Logical not**
- **Logical and**
- **Logical or**

- Precedence goes downwards
  - Parentheses highest
  - Logical ops lowest
- Same line = same precedence
  - Read "ties" left to right (except for **)
  - Example: 1/2*3 is (1/2)*3

- Section 2.5 in your text
- See website for more info
- Part of Lab 1

# Operators and Type Conversions

**Operator Precedence**

**Exponentiation**: **

**Negation**: –

**Binary arithmetic**: * / %

**Binary arithmetic**: + –

**Comparisons**: < > <= >=

**Equality relations**: == !=

**Logical not**

**Logical and**

**Logical or**

Evaluate this expression:

```
7 + 3.0 / 3
```

**A.** 3

**B.** 3.0

**C.** 3.333333335

**D.** 8

**E.** 8.0

# Operators and Type Conversions

**Operator Precedence**

**Exponentiation**: **

**Negation**: –

**Binary arithmetic**: * / %

**Binary arithmetic**: + –

**Comparisons**: < > <= >=

**Equality relations**: == !=

**Logical not**

**Logical and**

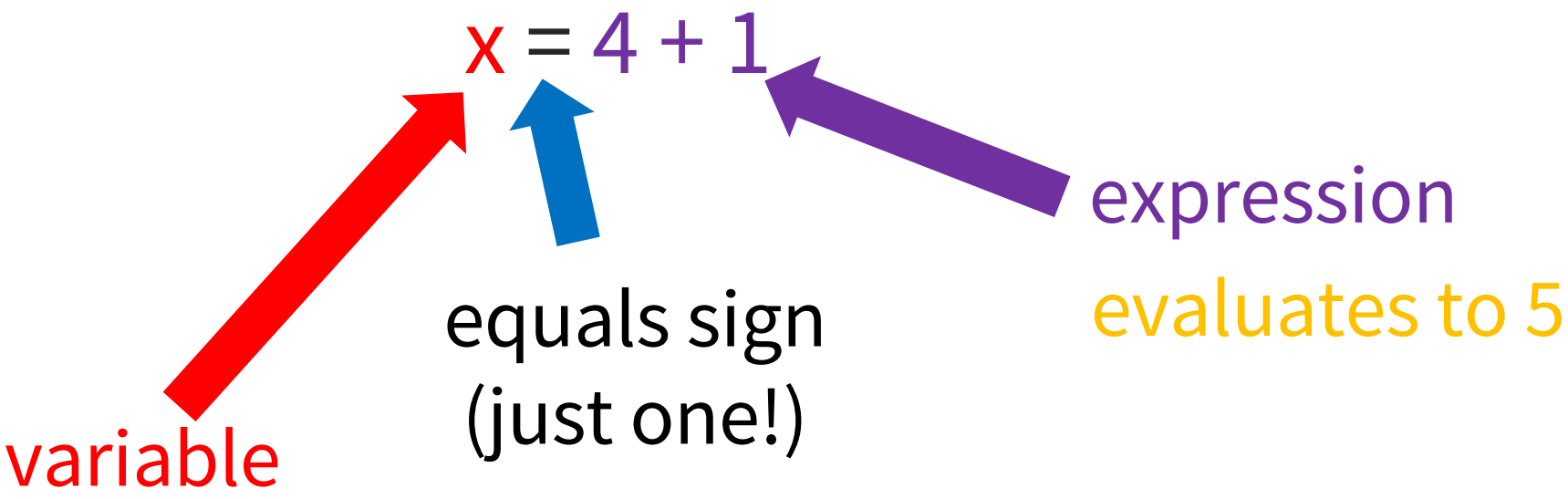
**Logical or**

Evaluate this expression:

```
7 + 3.0 / 3
    -------
      1.0
7 +     1.0
  -----------
      8.0
```

# New Tool: Variable Assignment

**Example**:




An *assignment statement:*

- takes an *expression*
- evaluates it, and
- stores the *value* in a *variable*

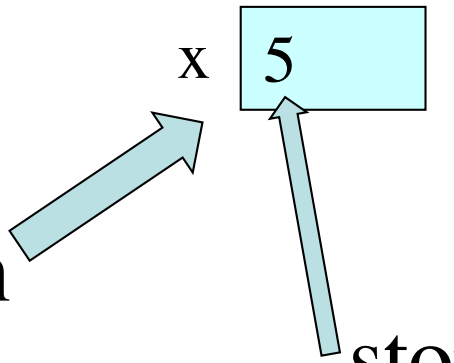
# Executing Assignment Statements

```
>>> x = 5
>>>
```

Press ENTER and…

Hmm, looks like nothing happened…

- But something did happen!
- Python *assigned* the *value* 5 to the *variable* x
- Internally (and invisible to you):




memory location

stored value

```
>>> terminal time >>>
```

# Retrieving Variables in Interactive Mode

```
>>> x = 5
>>> x
5
>>>
```

Press ENTER and…

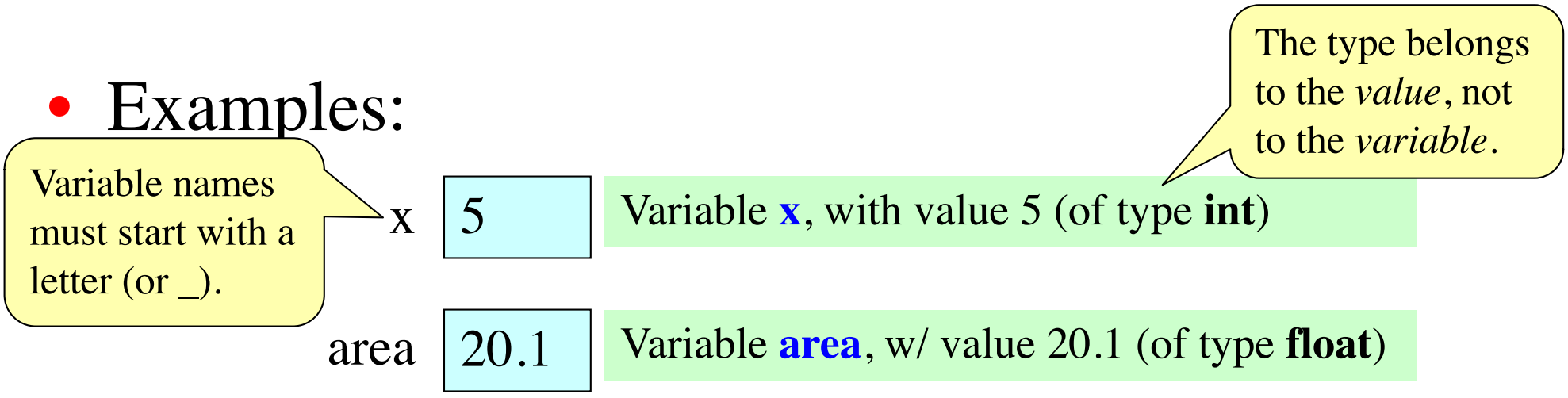
Interactive mode tells me the value of x

```
>>> terminal time >>>
```

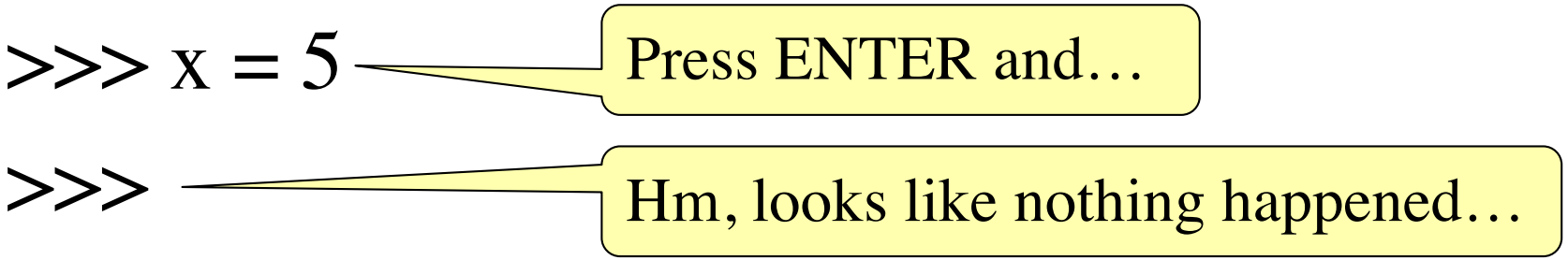
# In More Detail: Variables (Section 2.1)

- A **variable**
  - is a **named** memory location (**box**)
  - contains a **value** (in the box)

- Examples:



# In More Detail: Statements




- This is a **statement**, not an **expression**
  - Tells the computer to DO something (not give a value)
  - Typing it into >>> gets no response (but it is working)

# Expressions vs. Statements

## Expression

- **Represents** something
  - Python *evaluates it*
  - End result is a value
- Examples:
  - `2.3`
  - `(3+5)/4`
  - `x == 5`

Value

Complex Expression

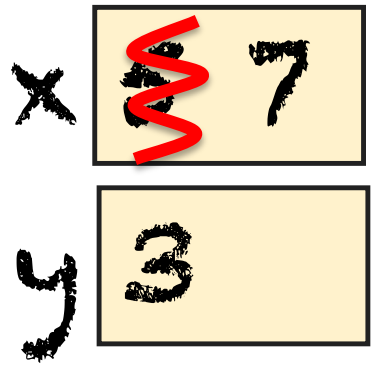
## Statement

- **Does** something
  - Python *executes it*
  - Need not result in a value
- Examples:
  - `x = 2 + 1`
  - `x = 5`

*Look so similar*

***but they are not!***

# Keeping Track of Variables




- Draw boxes on paper:

```
>>> x = 5
```

- New variable declared?

```
>>> y = 3
```

Write a new box.

- Variable updated?

```
>>> x = 7
```

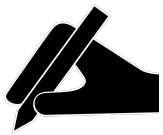
Cross out old value. Insert new value.

Start with variable x having value 5. Draw it on paper:


## Task: Execute the Statement: x = x + 2

1. Evaluate the RHS expression, **x + 2**
   - For x, use the value in variable **x**
   - What value doe the RHS expression evaluate to?

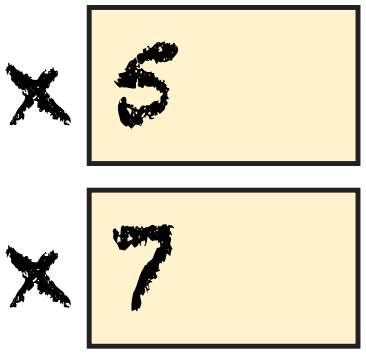
2. Store the value of the RHS expression in x in variable names on LHS, **x**
   - Cross off the old value in the box
   - Write the new value in the box for x



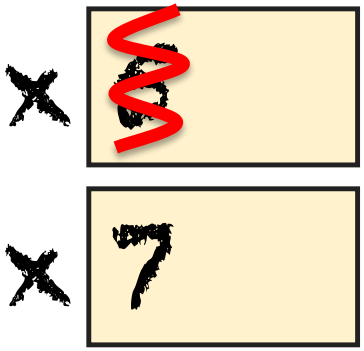
# Which one is closest to your answer?

A.


B.




C.




D.

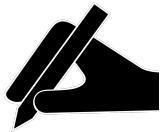
¯_(ツ)_/¯

$x = x + 2$

## Execute the Statement: `x = 3.0*x+1.0`

Begin with this:


1. **Evaluate** the expression `3.0*x+1.0`
2. **Store** its value in `x`



# Which one is closest to your answer?

A.

x 22.0

B.

x 7

x 22.0

C.

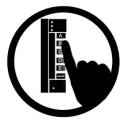
x

x 22.0

D.

¯_(ツ)_/¯

```
x = 3.0*x+1.0
```



# Executing an Assignment Statement

The command: `x = 3.0*x+1.0`

"Executing the command":

1. **Evaluate** right hand side `3.0*x+1.0`
2. **Store** the value in the variable `x`'s box

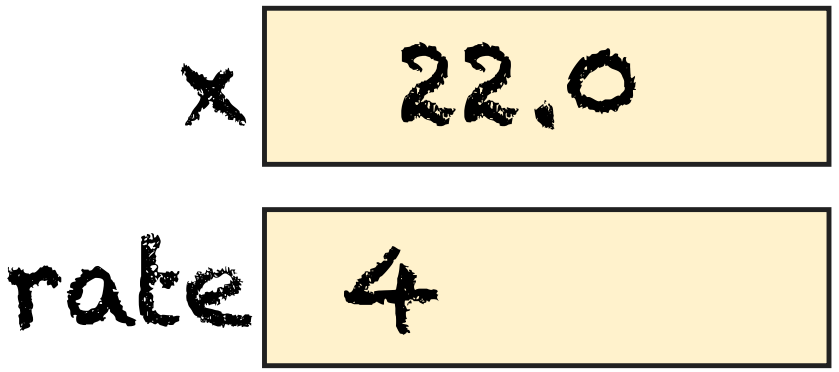
- Requires both evaluate AND store steps
- Critical mental model for learning Python

# Exercise 1: Understanding Assignment

Have variable **x** already from previous

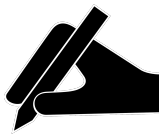
Create a new variable:

```
>>> rate = 4
```




Execute this assignment:

```
>>> rate = x / rate
```



# Which one is closest to your answer?

A.

x 22.0 5.5

rate 5.5

B.

x 22.0

rate 5

rate 5.5

C.

x 22.0

rate 5.5

D.

x 22.0

rate 5

E. ¯_(ツ)_/¯

```
rate = x / rate
```

# Dynamic Typing

Python is a **dynamically typed** language

- Variables can hold values of any type
- Variables can hold different types at different times

The following is acceptable in Python:

```
>>> x = 1
>>> x = x / 2.0
```

← x contains an **int** value

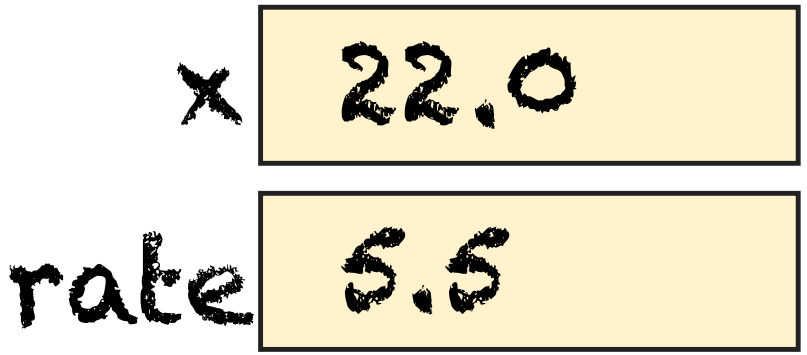
← x now contains a **float** value

Alternative: a **statically typed** language

- Examples: Java, C
- Each variable restricted to values of just one type

# Exercise 2: Understanding Assignment

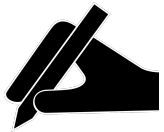
Begin with:




Execute this assignment:

```
>>> rat = x + rate
```

Did you do the same thing as your neighbor ? If not, *discuss*.



# Which one is closest to your answer?

A.

x 27.5

rate 5.5

B.

x 22.0

rate 5.5

rat 27.5

C.

x 22.0

rate 27.5

D.

x 22.0

rate 5.5

rat 27.5

E. ¯_(ツ)_/¯

```
rat = x + rate
```

# More Detail: Testing Types

Command: `type(<value>)`

Can test a variable:

```
>>> x = 5
>>> type(x)
<class 'int'>
```

Can test a type with a Boolean expression:

```
>>> type(2) == int
True
```