Lecture 2: Objects, Expressions & Variables

Comp 102

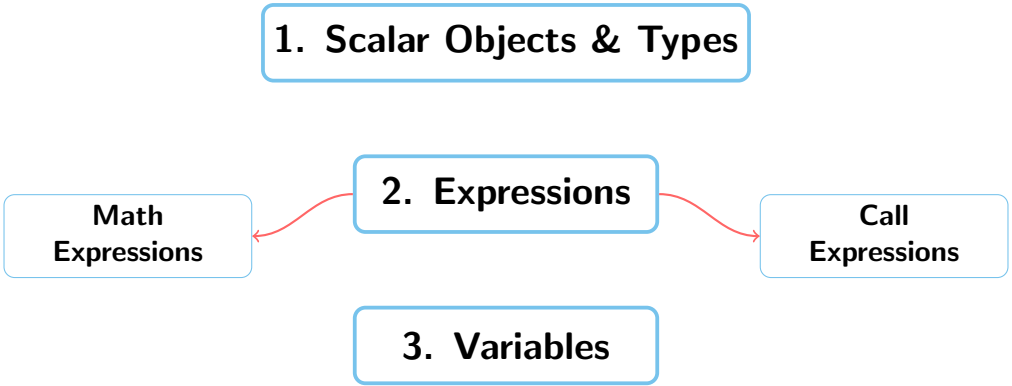
Forman Christian University

Recap from Lecture 1

- **Programming** = giving instructions to computers
- **Programmatic thinking** = breaking problems into steps
- **Machine language** vs **High-level languages**
- **Syntax** (rules) and **Semantics** (meaning)
- **Objects**: Everything in Python is an object
- **Scalar** vs **Non-scalar** objects

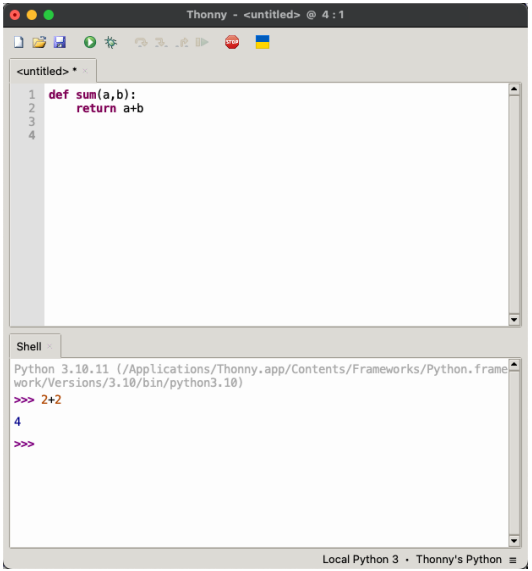
Today's Agenda

Building Blocks of Programming



Programming Environment

Thonny



Editor

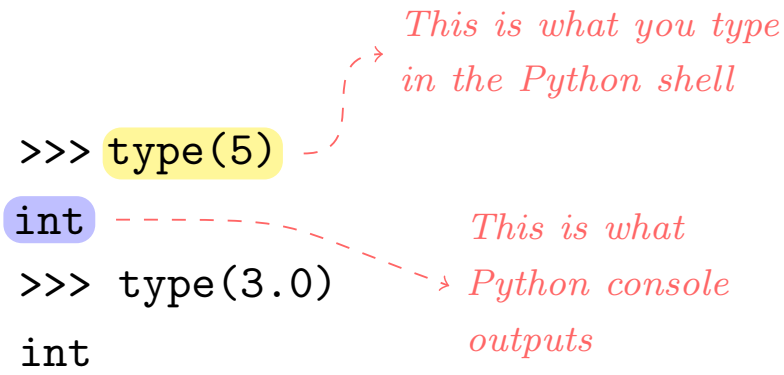
Shell

Scalar Objects

Scalar Objects *(can't be subdivided)*

- `int` - *represents integers:* 5, −100 *etc*
- `float` - *represents real numbers:* 3.27, 2.0 *etc*
- `bool` - `True`, `False`
- `NoneType` - *special, has only one value:* `None`

Use `type()` to see type of an object:



int

0, 1, 2, …
300, 301 …
-1, -2, -3, …
-400, -401, …

float

0.0, …, 0.21, …
1.0, …, 3.14, …
-1.22, …, -500.0 , …

bool

True
False

NoneType

None

You Try

Step 1: Predict the type of each WITHOUT using the computer:

Step 2: Verify your predictions in Python console:

1. `1234`
2. `8.99`
3. `9.0`
4. `True`
5. `False`
6. `None`

Type Conversions *(a.k.a Type Casting)*

- You can **convert object of one type to another**
  - `float(3)` casts the `int` 3 to a `float` 3.0
  - `int(3.9)` casts the `float` 3.9 to an `int` 3
    (note the truncation!)

Type Conversions *(a.k.a Type Casting)*

- You can **convert object of one type to another**
  - `float(3)` casts the `int` 3 to a `float` 3.0
  - `int(3.9)` casts the `float` 3.9 to an `int` 3
    (note the truncation!)
- Some operations perform **implicit casts** *(automatic conversions)*
  - `round(3.9)` returns the `int` 4

You Try

First: Predict the type & value WITHOUT using the computer.

Then: Verify in Python console:

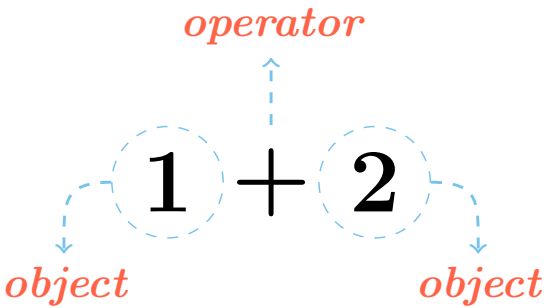
- `float(123)`
- `round(7.9)`
- `float(round(7.9))`
- `int(7.2)`
- `int(7.9)`

Expressions

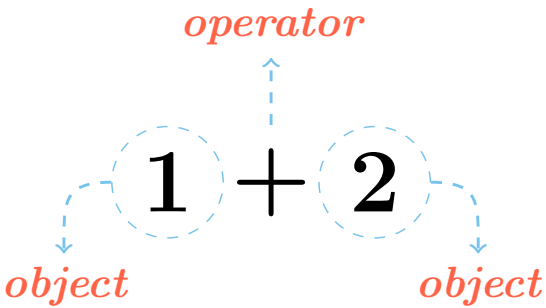
Math Expressions

(a.k.a Arithmetic Expressions)

Combine Objects and Operators to form **Expressions**



Combine Objects and Operators to form **Expressions**



Syntax:

```
<object> <operator> <object>
```

Examples:

```
>>> 1 + 2
3
>>> 5 / 2
2.5
```

Examples:

```
>>> 1 + 2
3
>>> 5 / 2
2.5
>>> (4 + 2) * 6 - 1
```

Examples:

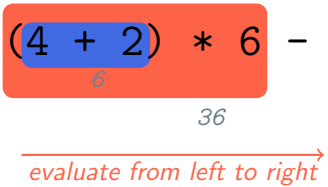
```
>>> 1 + 2
3
>>> 5 / 2
2.5
>>> (4 + 2) * 6 - 1
```

6

evaluate from left to right →

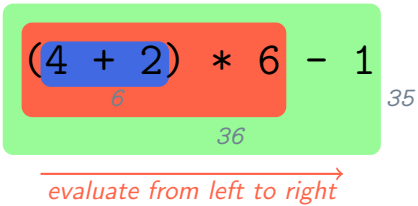
Examples:

```
>>> 1 + 2
3
>>> 5 / 2
2.5
>>> (4 + 2) * 6 - 1
```



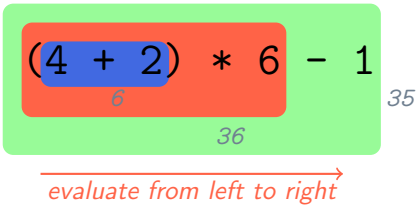
Examples:

```
>>> 1 + 2
3
>>> 5 / 2
2.5
>>> (4 + 2) * 6 - 1
```



Examples:

```
>>> 1 + 2
3
>>> 5 / 2
2.5
>>> (4 + 2) * 6 - 1
35
>>> 35
```



You Try:

Predict the answer FIRST, then verify with Python!

```
>>> (2 + 3) - 1
>>> 2 + (3 - 1)
>>> 3 / 2
>>> 3 ** 2
>>> type(3 ** 2)
```

You Try:

Predict the answer FIRST, then verify with Python!

```
>>> (2 + 3) - 1
```
5 - 1 → 4

```
>>> 2 + (3 - 1)
```
2 + 2 → 4 *(brackets first)*

```
>>> 3 / 2
```
1.5

```
>>> 3 ** 2
```
9

```
>>> type(3 ** 2)
```
int

Warning:

Do not use other types of brackets in expressions.

Use **ONLY** paranthesis: ()

Do **NOT** use brackets: [] **or** braces: {}

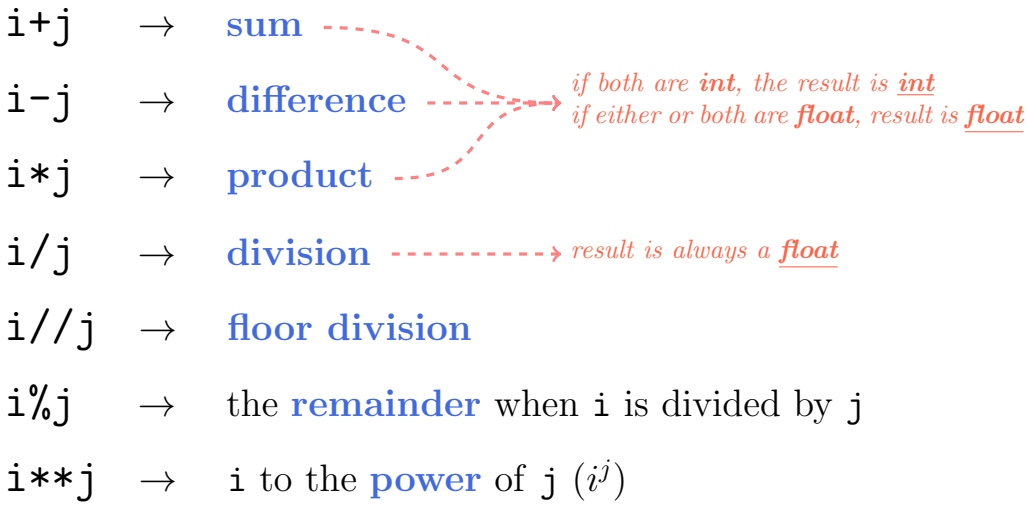
✓

```
2 + (3 - 1)
```

✗

```
2 + [3 - {1}]
```

Math Operators



You Try:

IMPORTANT: Write your predictions on paper FIRST!

Then evaluate in console to check:

- `(13-4) / (12*12)`
- `type(4 * 3)`
- `type(4.0 * 3)`
- `3 // 2`
- `5 % 2`
- `2 ** 5`

Big Idea

All *"arithmetic expressions"* evaluate to a **single value**.

Call Expressions

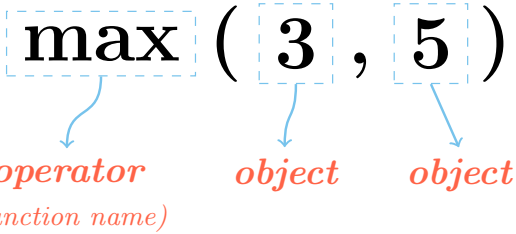
(a.k.a Function Calls)

Call Expressions

```
>>> max(3, 5)
5
```

Call Expressions

```
>>> max(3, 5)
5
```



Multiple objects can be passed to a function:

```
>>> max(3, 5, 7)          (comma separated)
7
>>> min(3, 5, 7, 1)
1
```

Multiple objects can be passed to a function:

```
>>> max(3, 5, 7)          (comma separated)
7
>>> min(3, 5, 7, 1)
1
```

Order of objects is important:

```
>>> pow(10, 2)
100
>>> pow(2, 10)
1024
```

Importing Library Functions

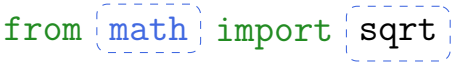
- Python has a large number of functions, **unavailable** by default.
- You must import functions from a **package (a.k.a module)** to use them:

```
>>> from math import sqrt
>>> sqrt(16)
4.0
```

Importing Library Functions

- Python has a large number of functions, **unavailable** by default.
- You must import functions from a **package (a.k.a module)** to use them:

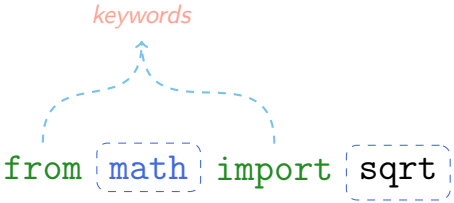
```
>>> from math import sqrt
>>> sqrt(16)
4.0
```



Importing Library Functions

- Python has a large number of functions, **unavailable** by default.
- You must import functions from a **package (a.k.a module)** to use them:

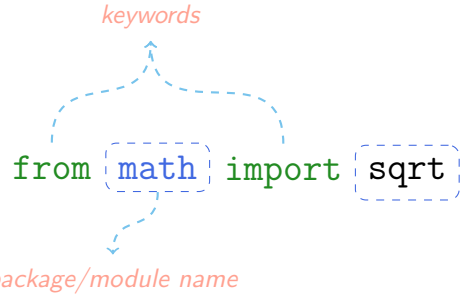
```
>>> from math import sqrt
>>> sqrt(16)
4.0
```



Importing Library Functions

- Python has a large number of functions, **unavailable** by default.
- You must import functions from a **package (a.k.a module)** to use them:

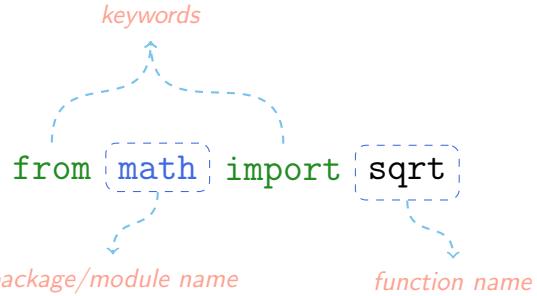
```
>>> from math import sqrt
>>> sqrt(16)
4.0
```



Importing Library Functions

- Python has a large number of functions, **unavailable** by default.
- You must import functions from a **package (a.k.a module)** to use them:

```
>>> from math import sqrt
>>> sqrt(16)
4.0
```



Call Expression Examples

```
>>> pow(2,3)
8
```

already available, no need to import

Call Expression Examples

```
>>> pow(2,3)
8
>>> from math import sqrt
>>> sqrt(256)
16.0
```

already available, no need to import

Call Expression Examples

```
>>> pow(2,3)
8
>>> from math import sqrt
>>> sqrt(256)
16.0
>>> from operator import add, sub, mul
>>> add(1, 1)
2
```

already available, no need to import

equivalent to → 1 + 1

Call Expression Examples

```
>>> pow(2,3)        already available, no need to import
8
>>> from math import sqrt
>>> sqrt(256)
16.0
>>> from operator import add, sub, mul
>>> add(1, 1)       equivalent to → 1 + 1
2
```

The `operator` module provides functions for arithmetic operations such as `+`, `-`, `*`, `/` etc.

Big Idea

All *"call expressions"* evaluate to a **single value**.

Evaluating Nested Expressions

"**Evaluation Tree**":

```
sub(10, mul(3, add(1, 2)))
```

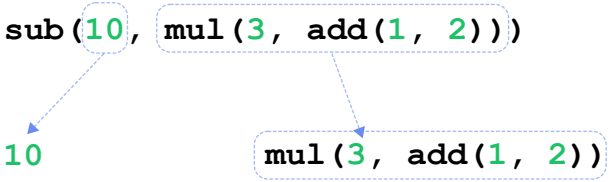
Evaluating Nested Expressions

"Evaluation Tree":

```
sub(10, mul(3, add(1, 2)))
```

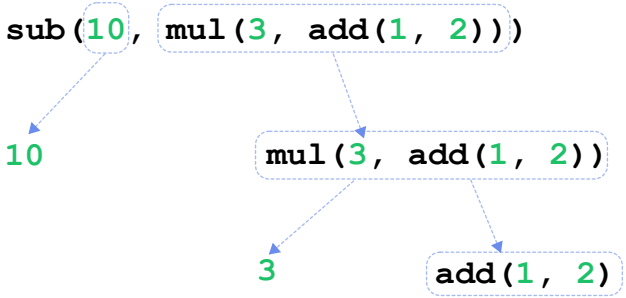
Evaluating Nested Expressions

"Evaluation Tree":



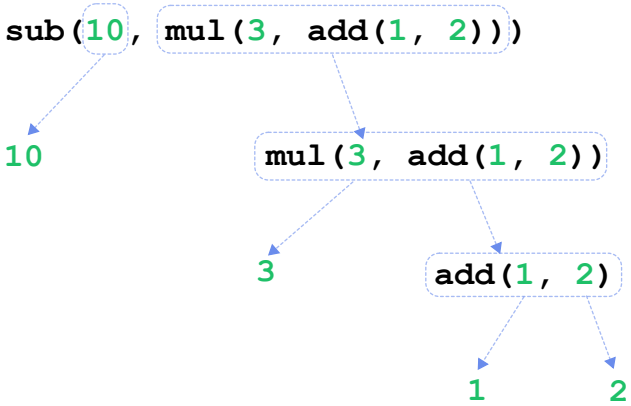
Evaluating Nested Expressions

"**Evaluation Tree**":



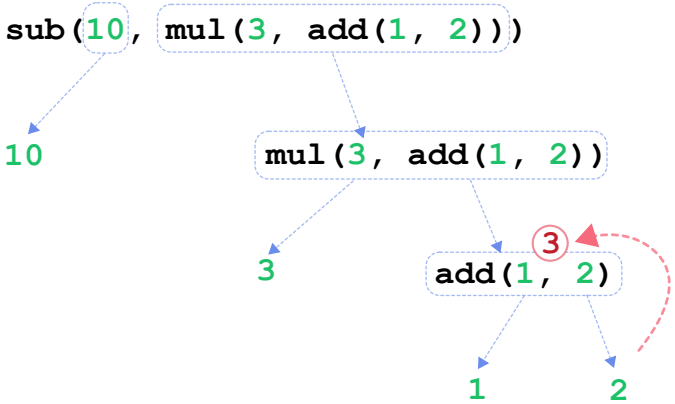
Evaluating Nested Expressions

"Evaluation Tree":



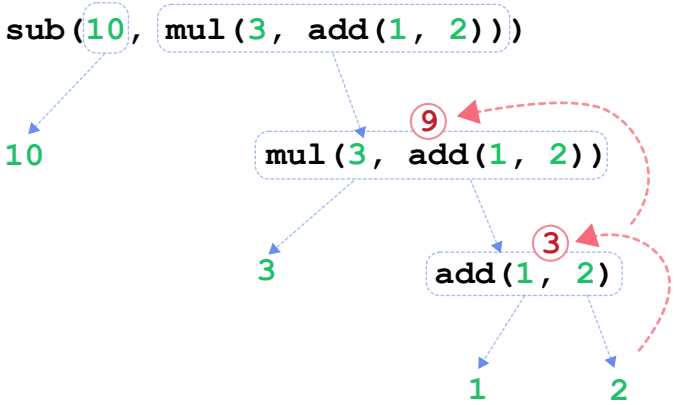
Evaluating Nested Expressions

"**Evaluation Tree**":



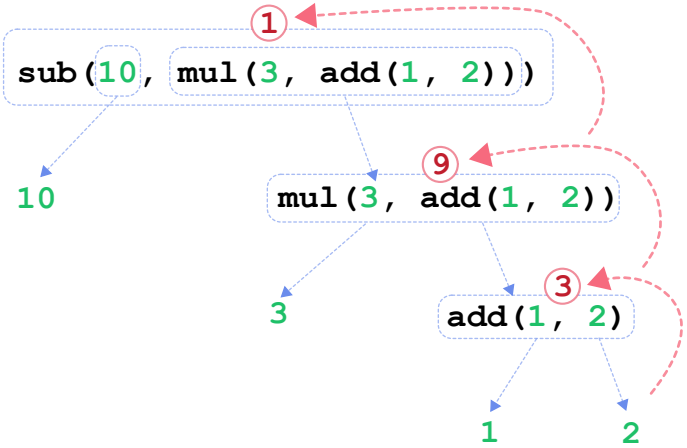
Evaluating Nested Expressions

"**Evaluation Tree**":



Evaluating Nested Expressions

"**Evaluation Tree**":



You Try:

Trace through each expression step-by-step on paper!

```
>>> from operator import add, sub, mul

>>> 1 + add(2,3)

>>> pow(2, sub(3,1))

>>> mul(2*(3+1),add(1,2))
```

You Try:

Trace through each expression step-by-step on paper!

```
>>> from operator import add, sub, mul
>>> 1 + add(2,3)
>>> pow(2, sub(3,1))
>>> mul(2*(3+1),add(1,2))
```

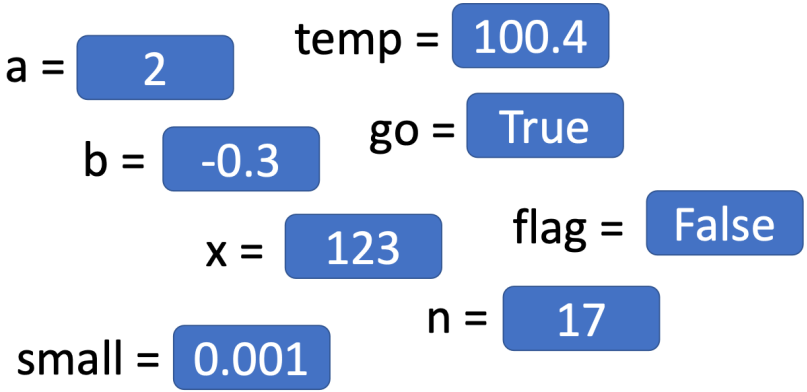
1 + 5 → 6

pow(2,2) → 4

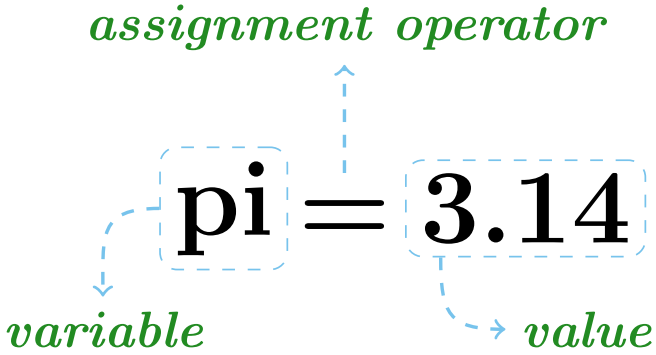
mul(8,3)→ 24

Variables

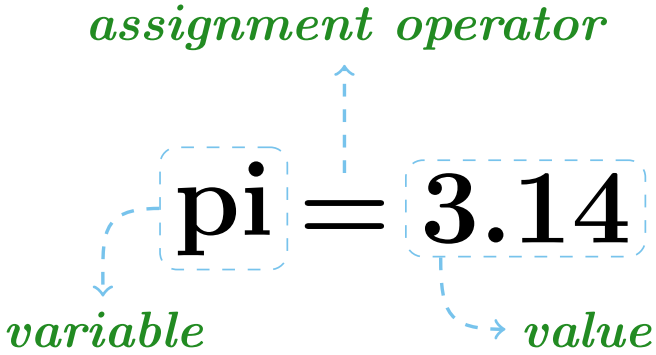
Bind names to objects:



Assignment Statement:



Assignment Statement:



WARNING: This is NOT Equal Sign! You are NOT comparing!

Math variables vs CS variables

Math Variables

1. = means equality

CS Variables

1. = means assignment

Math Variables

1. = means equality
2. can represent many values: $y = x^2$

CS Variables

1. = means assignment
2. bound to a single value

```
g = 9.81
```

Math Variables

1. = means equality
2. can represent many values:
   $y = x^2$
3. `b = (3+2)*1.5`
   Comparing `b` with the right side

CS Variables

1. = means assignment
2. bound to a single value
   `g = 9.81`
3. `b = (3+2)*1.5`
   `b` is bound to an expression
   (which will evaluate to a single value)

You Try:

Predict first: Which are allowed in Python?

Then verify in the console:

1. `x = 6`
2. `6 = x`
3. `x * y = 3 + 4`
4. `xy = 3 + 4`

Expression Abstraction

Why give names to values and expressions?

Expression Abstraction

Why give names to values and expressions?

- to reuse names instead of values
- to make code easier to read and modify

```
# Compute approximate value for pi
pi = 355/113
radius = 2.2
area = pi * (radius ** 2)
circumference = 2 * pi * radius
```

Expression Abstraction

Why give names to values and expressions?

- to reuse names instead of values
- to make code easier to read and modify

```
# Compute approximate value for pi
pi = 355/113
radius = 2.2
area = pi * (radius ** 2)
circumference = 2 * pi * radius
```

assignment operator

Expression Abstraction

Why give names to values and expressions?

- to reuse names instead of values
- to make code easier to read and modify

```
# Compute approximate value for pi
pi = 355/113
radius = 2.2
area = pi * (radius ** 2)
circumference = 2 * pi * radius
```

Expression Abstraction

Why give names to values and expressions?

- to reuse names instead of values
- to make code easier to read and modify

```
# Compute approximate value for pi
pi = 355/113
radius = 2.2
area = pi * (radius ** 2)
circumference = 2 * pi * radius
```

comment
*is **not** a part of the program*

Choose variable names wisely:

- You'll be fine if you stick to **letters** and **underscores**
- Just **don't** start the name with numbers

You Try:

Which of the following are valid variable names?

1. `lunchPrice`
2. `fried rice`
3. `greek-salad`
4. `chilli_sauce`
5. `7up`

You Try:

Which of the following are valid variable names?

1. `lunchPrice` ✓ — *valid variable name, note the camel case*
2. `fried rice` ✗ — *invalid variable name, spaces are not allowed*
3. `greek-salad` ✗ — *this is a subtraction expression*
4. `chilli_sauce` ✓ — *valid variable name*
5. `7up` ✗ — *variable names cannot start with numbers*

Which one is the Best Code Style?

```
# do calculations
a = 355/113 * (2.2 ** 2)
c = 355/113 * (2.2 * 2)
```

```
p = 355/133
r = 2.2
# multiply p with r squared
a = p*(r**2)
# multiply p with r times 2
c = p*(r*2)
```

```
# calculate area and circumference of a circle
# using an approximation of pi
pi = 355/113
radius = 2.2
area = pi * (radius ** 2)
circumference = pi * (radius * 2)
```

Which one is the Best Code Style?

```
# do calculations
a = 355/113 * (2.2 ** 2)
c = 355/113 * (2.2 * 2)
```

```
p = 355/133
r = 2.2
# multiply p with r squared
a = p*(r**2)
# multiply p with r times 2
c = p*(r*2)
```

Horrible...

- unclear comments,
- single letter variable names,
- not assigning names to expressions

```
# calculate area and circumference of a circle
# using an approximation of pi
pi = 355/113
radius = 2.2
area = pi * (radius ** 2)
circumference = pi * (radius * 2)
```

Which one is the Best Code Style?

```
# do calculations
a = 355/113 * (2.2 ** 2)
c = 355/113 * (2.2 * 2)
```

```
p = 355/133
r = 2.2
# multiply p with r squared
a = p*(r**2)
# multiply p with r times 2
c = p*(r*2)
```

Still Bad

- atleast the comments are meaningful,
- expressions are bound to variables,
- variable names are still not very descriptive

```
# calculate area and circumference of a circle
# using an approximation of pi
pi = 355/113
radius = 2.2
area = pi * (radius ** 2)
circumference = pi * (radius * 2)
```

Which one is the Best Code Style?

```
# do calculations
a = 355/113 * (2.2 ** 2)
c = 355/113 * (2.2 * 2)
```

```
p = 355/133
r = 2.2
# multiply p with r squared
a = p*(r**2)
# multiply p with r times 2
c = p*(r*2)
```

```
# calculate area and circumference of a circle
# using an approximation of pi
pi = 355/113
radius = 2.2
area = pi * (radius ** 2)
circumference = pi * (radius * 2)
```

Best!

- very descriptive comments throughout the code,
- expressions are bound to variables,
- variable names are very descriptive

Change Bindings

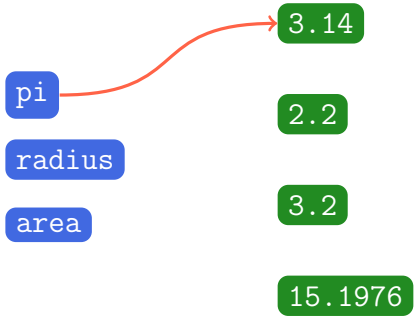
- Variables can **re-bind** to other values

```
pi = 3.14
radius = 2.2
area = pi*(radius**2)
radius = 3.5
print(area)
```

Change Bindings

- Variables can **re-bind** to other values

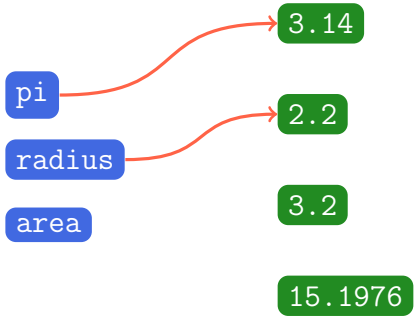
```
pi = 3.14
radius = 2.2
area = pi*(radius**2)
radius = 3.5
print(area)
```



Change Bindings

- Variables can **re-bind** to other values

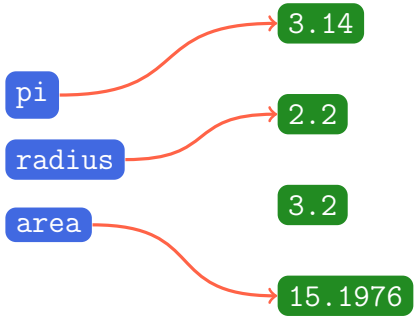
```
pi = 3.14
radius = 2.2
area = pi*(radius**2)
radius = 3.5
print(area)
```



Change Bindings

- Variables can **re-bind** to other values

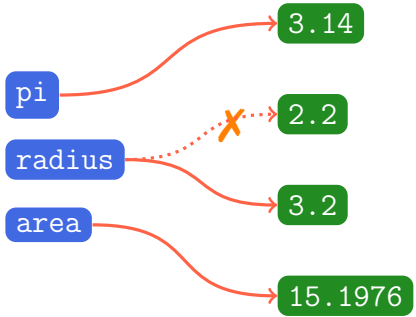
```
pi = 3.14
radius = 2.2
area = pi*(radius**2)
radius = 3.5
print(area)
```



Change Bindings

- Variables can **re-bind** to other values

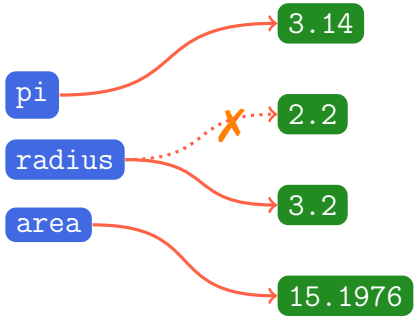
```
pi = 3.14
radius = 2.2
area = pi*(radius**2)
radius = 3.5
print(area)
```



Change Bindings

- Variables can **re-bind** to other values

```
pi = 3.14
radius = 2.2
area = pi*(radius**2)
radius = 3.5
print(area)
```



the area is still **15.1976**
have to calculate the area again!

Big Idea

Lines are evaluated one after the other.

No skipping around, yet.

We'll see how lines can be skipped/repeated later.

You Try:

Step 1: Trace through this code BY HAND.

What are the values of **meters** and **feet** after each line?

```
meters = 100
feet = 3.2808 * meters
meters = 200
```

Step 2: Verify with PythonTutor

- Follow along with this Python Tutor LINK

Where did we tell Python to (re)calculate feet?

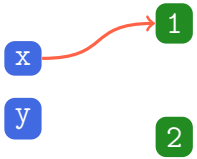
You Try:

Swap values of x and y without binding the numbers directly.

Debug (a.k.a fix) this code:

```
x = 1
y = 2

y = x
x = y
```



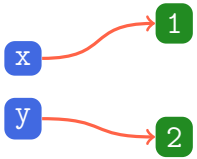
You Try:

Swap values of x and y without binding the numbers directly.

Debug (a.k.a fix) this code:

```
x = 1
y = 2

y = x
x = y
```



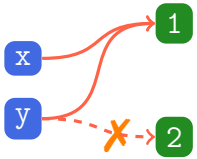
You Try:

Swap values of x and y without binding the numbers directly.

Debug (a.k.a fix) this code:

```
x = 1
y = 2

y = x
x = y
```



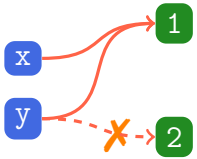
You Try:

Swap values of x and y without binding the numbers directly.

Debug (a.k.a fix) this code:

```
x = 1
y = 2

y = x
x = y
```



You Try:

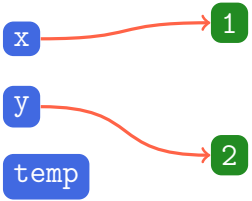
Swap values of x and y without binding the numbers directly.

Debug (a.k.a fix) this code:

```
x = 1
y = 2

y = x
x = y
```

Hint:



Python Tutor to the rescue ?

You Try:

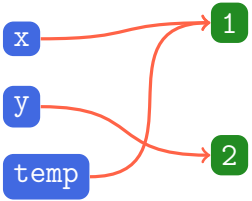
Swap values of x and y without binding the numbers directly.

Debug (a.k.a fix) this code:

```
x = 1
y = 2

y = x
x = y
```

Hint:



Python Tutor to the rescue ?

You Try:

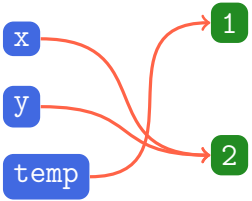
Swap values of x and y without binding the numbers directly.

Debug (a.k.a fix) this code:

```
x = 1
y = 2

y = x
x = y
```

Hint:



Python Tutor to the rescue ?

You Try:

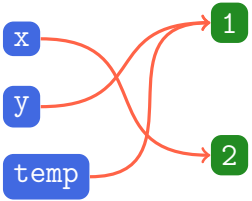
Swap values of x and y without binding the numbers directly.

Debug (a.k.a fix) this code:

```
x = 1
y = 2

y = x
x = y
```

Hint:

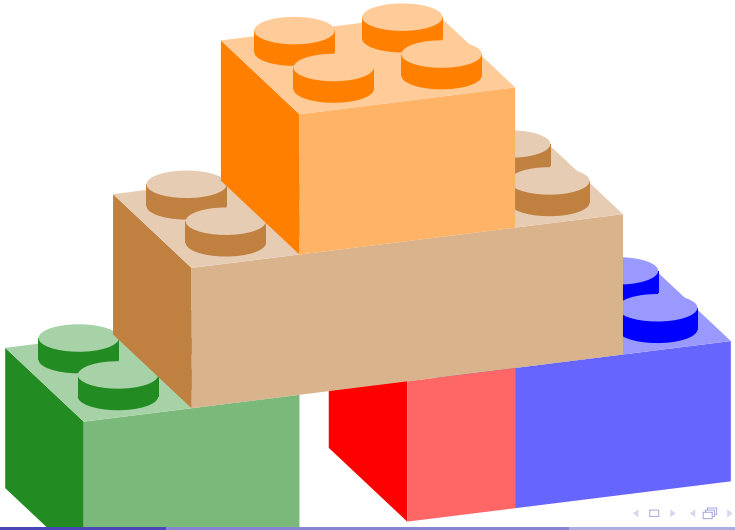


Python Tutor to the rescue ?

Summary

- **Programming Environment**
  - Thonny IDE
- **Scalar Objects**
  - `int`, `float`, `bool`, `NoneType`
  - `type()`
- **Type Casting**
  - `int()`, `float()`
- **Expressions**
  - Math expressions
  - Call expressions
  - All evaluate to single value
- **Variables**
  - Assignment statements
  - Variable naming
  - Re-binding values

Building Blocks
of Programming



Questions ?