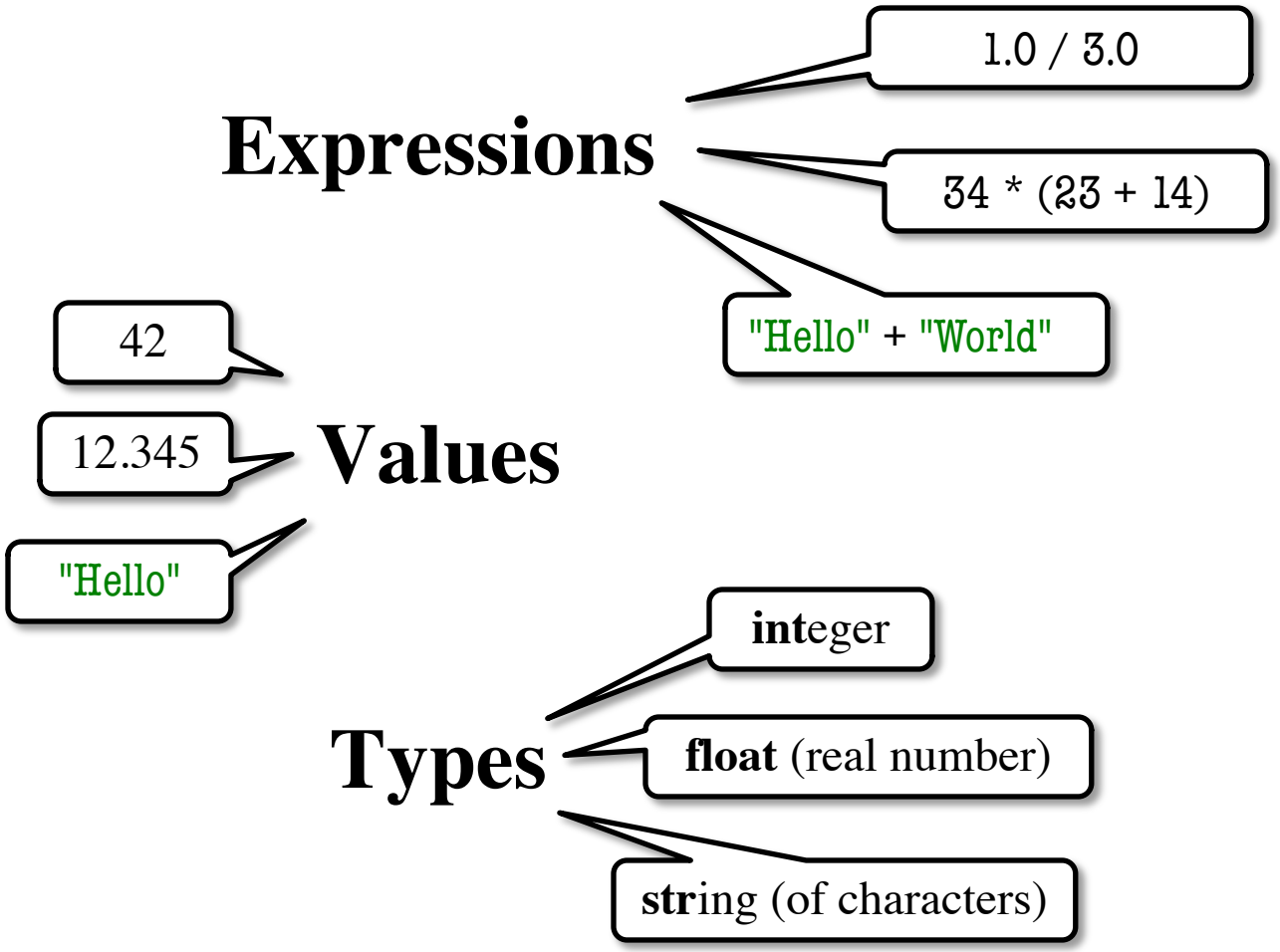
Module 1

Expressions and Types

The Three Main Concepts



Expressions

- **Expression:** something you type into Python
  - Right now, type after the >>>
  - Will see how to put into files in later on
- Can just be simple numbers (e.g. 34)
- Or can be mathematical formula
  - `1.0/3.0`
  - `34 * (23 + 14)`
  - `"Hello" + "World"`

Can be things other than numbers (i.e. text)

Values

- **Values:** what Python produces from expressions
  - A expression **represents** something
  - Python *evaluates it* (turns it into a value)
  - Similar to what a calculator does
- Examples:

```
>>> 2.3
2.3
>>> (3 * 7 + 2) * 0.5
11.5
```

Literal
(evaluates to self)

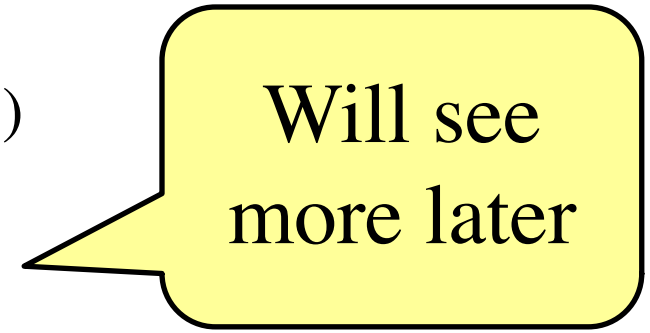
An expression with four literals and some **operators**

Types

- **Everything** on a computer reduces to numbers
  - Letters represented by numbers (ASCII codes)
  - Pixel colors are three numbers (red, blue, green)
  - So how can Python tell all these numbers apart?
- **Type: Set of values and operations on them**
  - Examples of operations: +, -, /, *
  - The meaning of these depends on the type
  - **Example**: `1+1` vs `"Hello" + "World"`

Type int: the Integers

- **Values** are positive and negative whole numbers
  - **Examples**: …, –3, –2, –1, 0, 1, 2, 3, 4, 5, …
  - Literals should only have digits (no commas or periods)
  - Good: 43028030 , **BAD**: 43,028,030
- **Operations** are typical math operations
  - **Addition**: +
  - **Subtraction**: - (but also **MINUS**)
  - **Multiply**: *
  - **Divide**: //
  - **Exponent**: ** (to the power of)



Understanding Operations

- Operations on **int** values must yield an **int**
  - **Example:** `1 // 2` rounds result down to 0
  - Companion operation: % (remainder)
  - `7 % 3` evaluates to 1, remainder when dividing 7 by 3
- Operator / is not an **int** operation in Python 3
  - This is an operator for the float type (separate video)
  - You won't get an error, but Python does something different
  - Will address in a later video
  - For now, restrict operations on int to those meant for it

Type `float`: Real Numbers

- **Values** are distinguished by decimal points
  - A number with a "." is a **float** literal (e.g. `2.0`)
  - Without a decimal a number is an **int** literal (e.g. `2`)
- **Operations** are *almost* the same as for **int**
  - float has a different division operator
  - **Example**: `1.0/2.0` evaluates to 0.5
  - But also supports the // operation
  - And the % operation

Using Big Numbers

- **Exponent notation** is useful for large (or small) values
  - `-22.51e6` is $-22.51 * 10^{6}$ or -22510000
  - `22.51e-6` is $22.51 * 10^{-6}$ or 0.00002251

A second kind of **float** literal

- Python prefers this in some cases

```
>>> 0.00000000001
1e-11
```

Remember: Values look like literals

Floats Have Finite Precision

- Try this example:

```
>>> 0.1+0.2
0.30000000000000004
```

- The problem is **representation error**
  - Not all fractions can be **represented** as (finite) decimals
  - **Example**: calculators represent 2/3 as 0.666667
- Python does not use decimals
  - It uses IEEE 754 standard (beyond scope of course)
  - Not all decimals can be **represented** in this standard
  - So Python picks something close enough

Floats Have Finite Precision

- Try this example:

```
>>> 0.1+0.2
0.30000000000000004
```

- The pro
  - Not a mals
  - **Example**: calculators represent 2/3 as 0.666667
- Python does not use decimals
  - It uses IEEE 754 standard (beyond scope of course)
  - Not all decimals can be **represented** in this standard
  - So Python picks something close enough

Again: **Expressions** vs **Values**

int versus float

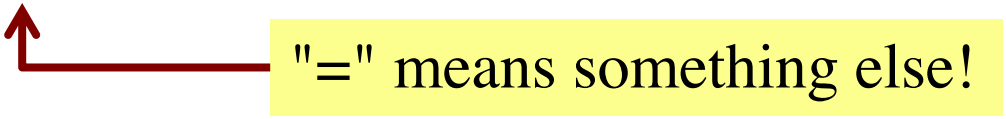
- This is why Python has two number types
  - int is limited, but the answers are always exact
  - float is flexible, but answers are approximate
- Errors in float expressions can propagate
  - Each operation adds more and more error
  - Small enough not to matter day-to-day
  - But important in scientific or graphics apps (high precision is necessary)
  - Must think in terms of **significant digits**

Type `bool` : Logical Statements

- **Values** are `True`, `False` (no more)
  - Capitalization is necessary!
  - Different from most other languages (lower case)
- **Operations** are `not`, `and`, `or` (and a few more)
  - not b: **True** if b is false and **False** if b is true
  - b and c: **True** if both b and c are true; else **False**
  - b or c: **True** if b is true or c is true; else **False**
- One of the most important Python types

Often Come from Comparisons

- Order comparisons:
  - Less than (`1 < 2`), less-than-or-equal (`1 <= 2`)
  - Greater than (`1 > 2`), greater-than-or-equal (`1 >= 2`)
- Equality comparisons
  - Equality (`1 == 2`), Inequality (`1 != 2`)



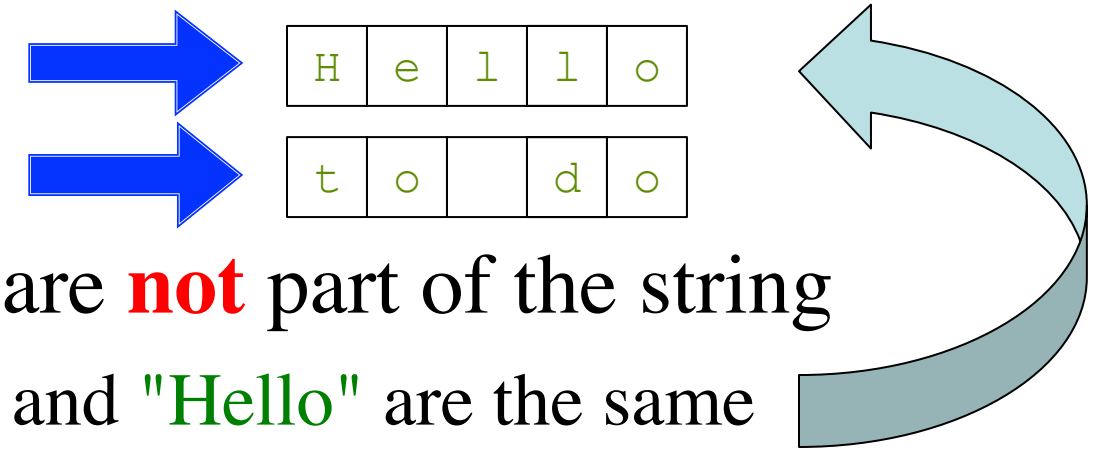
  - **Warning**: Equality is *unpredictable* on floats
- Can combine with `not`, `and`, `or`
  - **Example**: `(1 < 2) and (4 > 3)`

Type **str**: Text data

- **Values** are any sequence of characters
  - Character is anything we might type in text
  - Could be letters, punctuation, numbers, emoji
  - If you can type it, it is likely a character
- How distinguish text numbers from int, float?
- **String literal**: sequence of characters in quotes
  - Single quotes: `'Hello World!'` (Python prefers)
  - Double quotes: `"Hello World!"`
  - So `3` is an **int**, but `'3'` is a string

Visualizing Strings

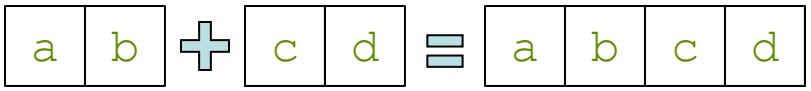
- Python treats each character a separate value
  - Can imagine string as a collection of boxes
  - Each character gets its own box
- Examples:
  - 'Hello'
  - 'to do'



- Quotes are **not** part of the string
  - 'Hello' and "Hello" are the same
  - In fact, 'Hello' == "Hello"

Operations (For Now)

- **Operation** +: s_1 + s_2
  - Glues if s_2 to end of s_1
  - Called *concatenation*
  - Evaluates to a string
- **Examples**:
  - 'ab' + 'cd' is 'abcd'



  - 'ab' + ' ' + 'cd' is 'ab cd'
  - Empty string '' is no boxes

- **Operation** in: s_1 in s_2
  - Tests if s_1 "a part of" s_2
  - If the *boxes* of s_1 are in s_2
  - Say s_1 a *substring* of s_2
  - Evaluates to a boolean
- **Examples**:
  - 'a' in 'abcde' is True
  - 'ab' in 'abcde' is True
  - 'ac' in 'abcde' is False

Operator Precedence

- What is the difference between the following?
  - 2*(1+3) **add, then multiply**
  - 2*1 + 3 **multiply, then add**
- Operations are performed in a set order
  - Parentheses make the order explicit
  - What happens when there are no parentheses?
- **Operator Precedence**: The *fixed* order Python processes operators in *absence* of parentheses

Precedence of Python Operators

- **Exponentiation**: **
- **Unary operators**: + –
- **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
  - Example: 1/2*3 is (1/2)*3

Labs are secretly training you to learn all this

Precedence of Python Operators

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

More complex than

- P (parentheses)
- E (exponentiation)
- M (multiplication)
- D (division)
- A (addition)
- S (subtraction)

Labs are secretly training you to learn all this

An Interesting Example

```
>>> 1 + 2 < 5 + 7
 3 < 12
True
>>> 1 + (2 < 5) + 7
1 + True + 7
9
```

Not an error!

Motivation for next video

Mixing Types

- Some operators allow us to mix (certain) types
  - **Example**: `1 + 2.5` is `3.5`
  - But `'ab' + 2` is an error
- What is Python doing? It is converting types
  - Addition needs both values same type
  - So it chooses float, not int (Why?)
  - float to int would have to drop or round .5
  - This is a really bad error, so int to float instead
  - Even though some (small) error in that conversion

Type Conversion

- Python can convert between **bool, int, float**
  - String is difficult and will talk about later
  - Narrow to wide: **bool ⇒ int ⇒ float**
- **Widening**: Convert to a wider type
  - Python does automatically if needed
  - **Example**: 1/2.0 evaluates to 0.5 (converts 1 to **float**)
- **Narrowing**: Convert to narrower type
  - Python *never* does this automatically
  - They cause information to be lost

Type Casting: Explicit Conversions

- Basic form: *type*(*value*)
  - float(2) converts value 2 to type **float** (value now 2.0)
  - int(2.6) converts value 2.6 to type **int** (value now 2)
  - Only way to narrow cast
- Can *sort* of do this with string
  - str(2) converts 2 to type **str** (value now '2')
  - int('2') converts string '2' to type int (value now 2)
- But we typically do not call this casting
  - Main issue is that it can fail: int('a') is an error
  - Conversions between **bool, int, float** *never* fail