Variables & Data Types

Outline

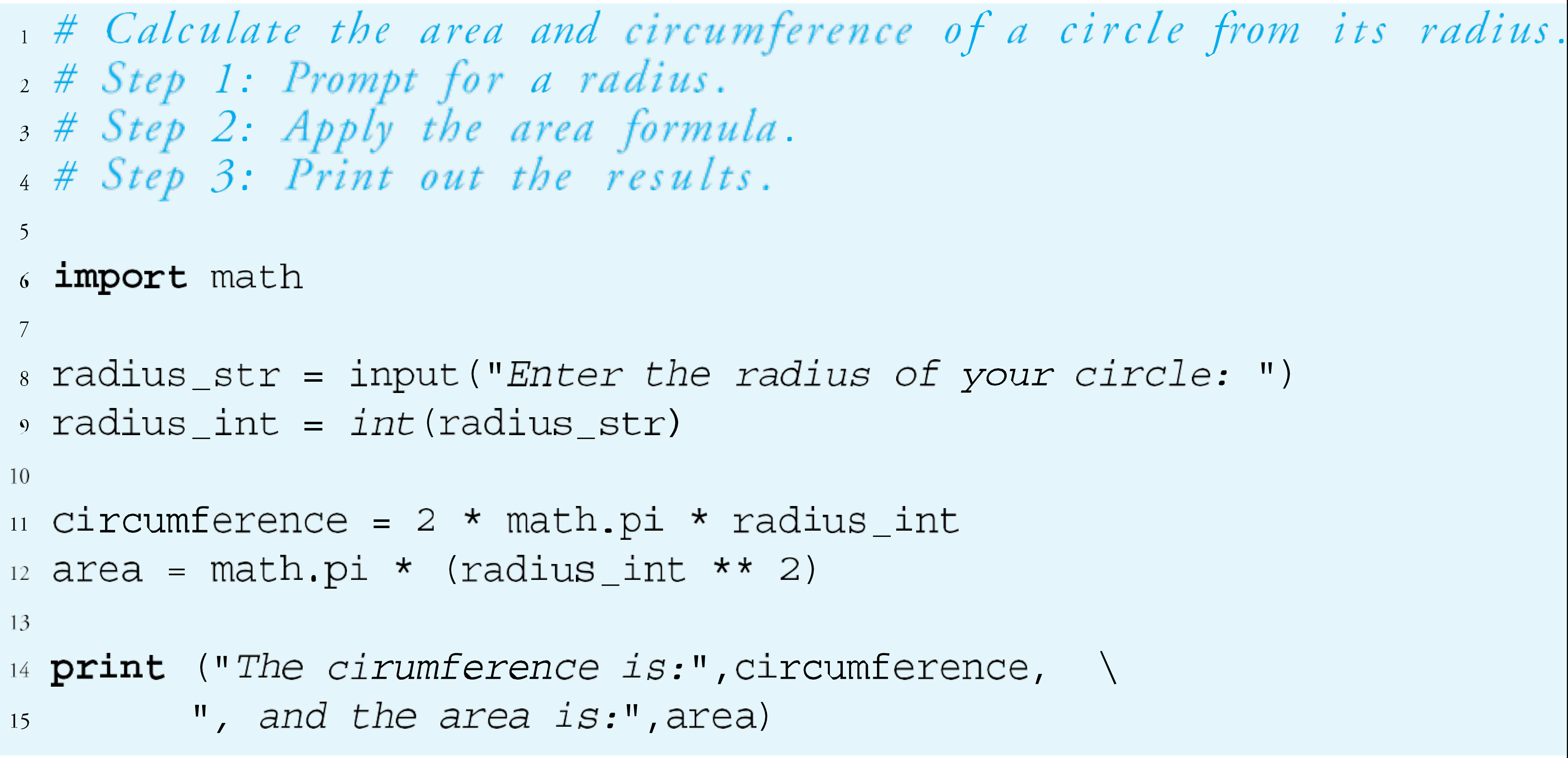
- Program
- Variables
- Objects and Types
- Operators

Program

Program

- A program is a sequence of instructions.
- To run a program is to:
  - create a sequence of instructions according to your design and the language rules
  - turn that program into the binary commands the processor understands
  - give the binary code to the OS, so it can give it to the processor
  - OS tells the processor to run the program
  - when finished (or it dies :-), OS cleans up.

Your First Program



```
# Calculate the area and circumference of a circle from its radius.
# Step 1: Prompt for a radius.
# Step 2: Apply the area formula.
# Step 3: Print out the results.

import math

radius_str = input("Enter the radius of your circle: ")
radius_int = int(radius_str)

circumference = 2 * math.pi * radius_int
area = math.pi * (radius_int ** 2)

print ("The cirumference is:",circumference,  \
      ", and the area is:",area)
```

Live coding!

Import of math

- One thing we did was to import the math module with **`import math`**
- This brought in python statements to support math (try it in the python window)
- We precede all operations of math with **`math.xxx`**
- **`math.pi`**, for example, is pi.
- **`math.pow(x,y)`** raises x to the y^{th} power.

Getting input

The function:

```
input("Give me a value")
```

- prints Give me a value on the python screen and waits till the user types something (anything), ending with Enter
- Warning, it returns a string (sequence of characters), no matter what is given, even a number ('1' is not the same as 1, different types)

Assignment

The **=** sign is the **assignment** statement

- The value on the right is associated with the variable name on the left
- It does ***not*** stand for equality!
- More on this later

Conversion

Convert from string to integer

- The user's response returned by input is stored as sequence of characters, called **a string**.
- For this program, we want to work with **numbers**
- Python requires that you must convert a sequence of characters to an integer
- Once converted, we can do math on the integers
- Use **`int`** function:

```
radius_int = int(str_radius)
```

Printing output

```
my_var = 12
print('My var has a value of: ',myVar)
```

- `print` takes a list of elements in parentheses separated by commas
  - if the element is a string, prints it as is
  - if the element is a variable, prints the value associated with the variable
  - after printing, moves on to a new line of output

At the core of any language

- Control the flow of the program
- Construct and access data elements
- Operate on data elements
- Construct functions
- Construct classes
- Libraries and built-in classes

Save as a “module”

- When you save a file, such as our first program, and place a `.py` suffix on it, it becomes a python module
- You run the module from the IDE menu to see the results of the operation
- A module is just a file of python commands

Errors

- If there are interpreter errors, that is Python cannot run your code because the code is somehow malformed, you get an error
- You can then import the program again until there are no errors

Common Error

- Using most IDEs, if you save the file without a `.py` suffix, it will stop colorizing and formatting the file.
- Resave with the .py, everything is fine

Parts of Python Program

Outline:

- Modules
- Statements & Expressions
- Whitespace
- Comments
- Python Special Elements
- Naming Objects
- Recommendation on Naming

Syntax

- Lexical components.
- A Python program is:
  - A module (perhaps more than one)
  - Each module has python statements
  - Each statement has expressions

Modules

- We've seen modules already, they are essentially files with Python statements.
- There are modules provided by Python to perform common tasks (math, database, web interaction, etc.)
- The wealth of these modules is one of the great features of Python

Statements

- Statements are commands in Python.
- They perform some action, often called a side effect, but they **do not return any values**

```
In [1]: my_int = 5
```

# statement, no return value

but my_int now has value 5

```
In [2]: my_int
Out [2]: 5
```

Expressions

- Expressions perform some operation and **return a value**
- Expressions can act as statements, but statements cannot act as expressions (more on this later).
- Expressions typically do not modify values in the interpreter

```
In [3]: my_int + 5    # expression, value associated
                         to my_int added to 5
Out [3]: 10
In [4]: my_int       # no side effect of expression
Out [4]: 5
```

Side effects and returns

What is the difference between side effect and return?

- `1 + 2` returns a value (it's an expression). You can “catch”/assign the return value. However, nothing else changed as a result
- `print("hello")` doesn't return anything, but something else, the side effect, did happen. Something printed!

Whitespace

- ***white space*** are characters that don't print (blanks, tabs, carriage returns etc.
- Whitespace is *ignored* within both expressions and statements, use it to make a program more readable

```
Y=X+5
```

has exactly the same meaning as

```
Y = X + 5
```

- *Leading* whitespace, whitespace at the beginning of a line—defines ***indentation***. Indentation plays a special role in Python (see the following section).
- Blank lines are also considered to be whitespace

Continuation

However, python is sensitive to end of line stuff. To make a line continue, use the \

```
print("this is a test", \
" of continuation")
```

prints

```
this is a test of continuation
```

also, tabbing is special

- The use of tabs is also something that Python is sensitive to.
- We'll see more of that when we get to control, but be aware that the tab character has meaning to Python

Python comments

- A comment begins with a # (pound sign)
- This means that from the # to the end of that line, nothing will be interpreted by Python.
- You can write information that will help the reader with the code

Code as essay, an aside

- What is the primary goal of writing code:
    - to get it to do something
    - an essay on my problem solving thoughts
- Code is something to be read. You provide comments to help readability.

Knuth, Literate Programming (84)

Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

Python Tokens

Keywords:

You cannot use (are prevented from using) them in a variable name

and
as
assert
break
class
continue
def
del
elif
else
except
exec
finally
for
from
global
if
import
in
is
lambda
not
or
pass
print
raise
return
try
while
with
yield

Python Operators

Reserved operators in Python (expressions)

```
+     -     *     **    /     //    %
<<    >>    &     |     ^     ~
<     >     <=    >=    ==    !=    <>
```

Python Punctuators

Python punctuation/delimiters ($ and ? not allowed).

'   “   #   \

(   )   [   ]   {   }   @

,   :   .   `   =   ;

+=   -=   *=   /=   //=   %=

&= |=   ^=   >>=   <<=   **=

Literals

Literal is a programming notation for a ***fixed value***.

- For example, 123 is a fixed value, an integer
  - it would be weird if the symbol 123's value could change to be 3.14!

Python Name Conventions

- must begin with a letter or underscore _

  `Ab_123` is OK, but `123_ABC` is not.

- may contain letters, digits, and underscores

  `this_is_an_identifier_123`

- may be of any length

- upper and lower case letters are different

  `Length_Of_Rope` is not `length_of_rope`

- names starting with _ (underline) have special meaning. Be careful!

Naming conventions

- Fully described by PEP8 or Google Style Guide for Python
  - PEP 8 Style Guide for Python code: https://www.python.org/dev/peps/pep-0008/
  - Google Style for Python: https://google.github.io/styleguide/pyguide.html
- the standard way for most things named in python is **lower with under**, lower case with separate words joined by an underline:
  - this_is_a_var
  - my_list
  - square_root_function

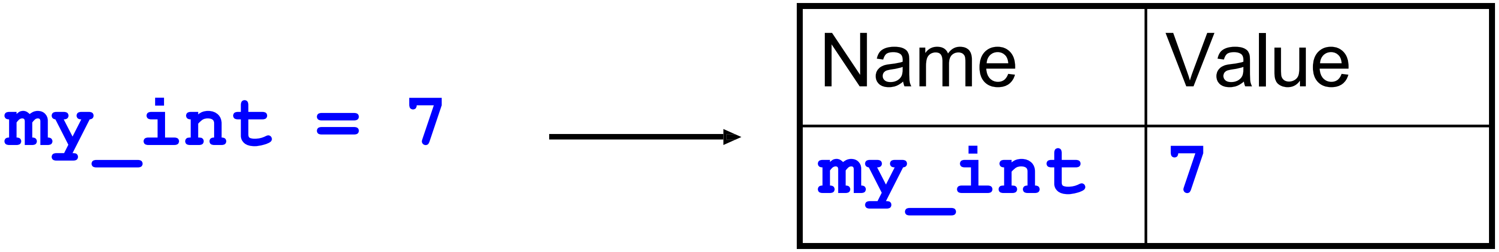
Variables

Variable

- A variable is a name we designate to represent an object (number, data structure, function, etc.) in our program
- We use names to make our program more readable, so that the object is easily understood in the program

Variable Objects

- Python maintains a list of pairs for every variable:
  - variable's name
  - variable's value
- A variable is created when a value is assigned the first time. It associates a name and a value
- subsequent assignments update the associated value.
- we say name references value



```
my_int = 7
```

Name	Value
my_int	7

Namespace

- A **namespace** is the table that contains the association of a name with a value
- We will see more about namespaces as we get further into Python, but it is an essential part of the language.

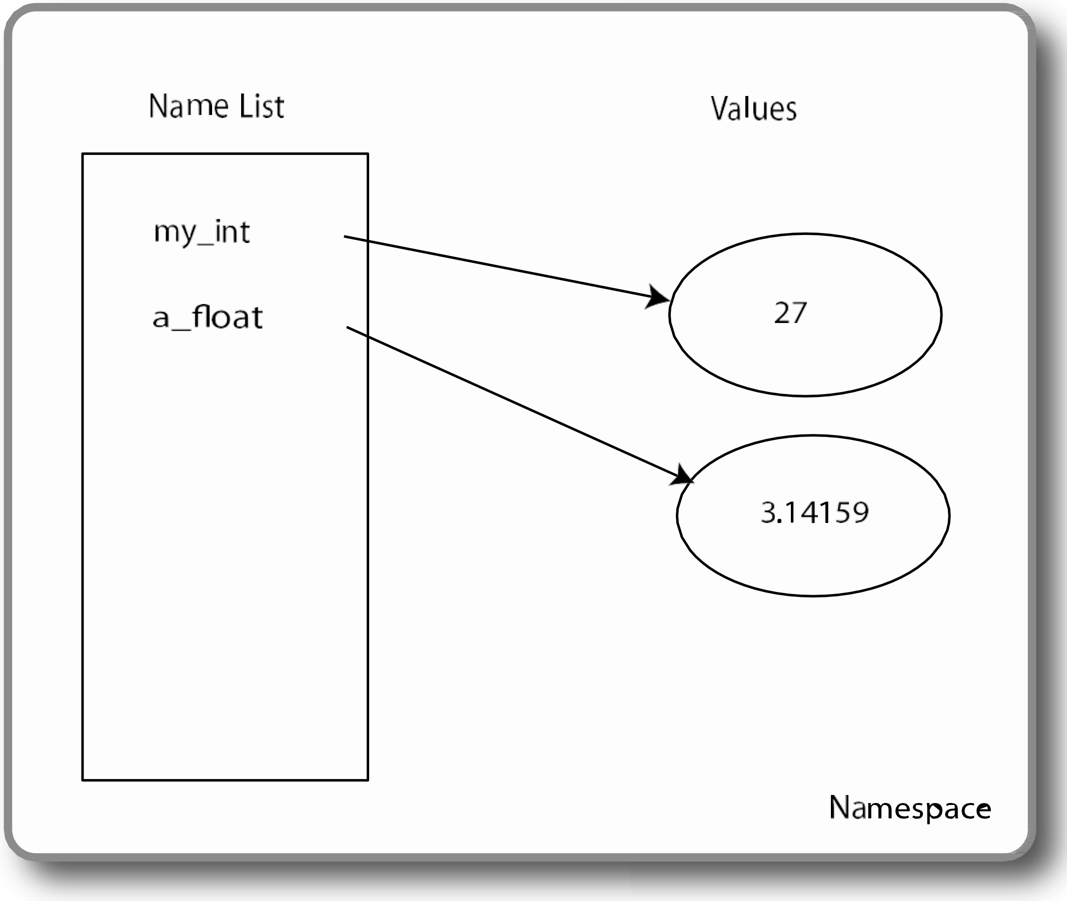


FIGURE 1.1 Namespace containing variable names and associated values.

When = doesn't mean equal

- It is most confusing at first to see the following kind of expression:

```
my_int = my_int + 7
```

- You don't have to be a math genius to figure out something is wrong there.
- What's wrong is that `=` doesn't mean equal

= is assignment

- In many computer languages, = means assignment.

```
my_int = my_int + 7
lhs = rhs
```

- What assignment means is:
  - evaluate the rhs of the =
  - take the resulting value and associate it with the name on the lhs

More Assignment

- Example: `my_var = 2 + 3 * 5`
  - evaluate expression `(2+3*5): 17`
  - change the value of `my_var` to reference 17
- Example (`my_int` has value 2):

```
my_int = my_int + 3
```

  - evaluate expression `(my_int + 3): 5`
  - change the value of `my_int` to reference 5

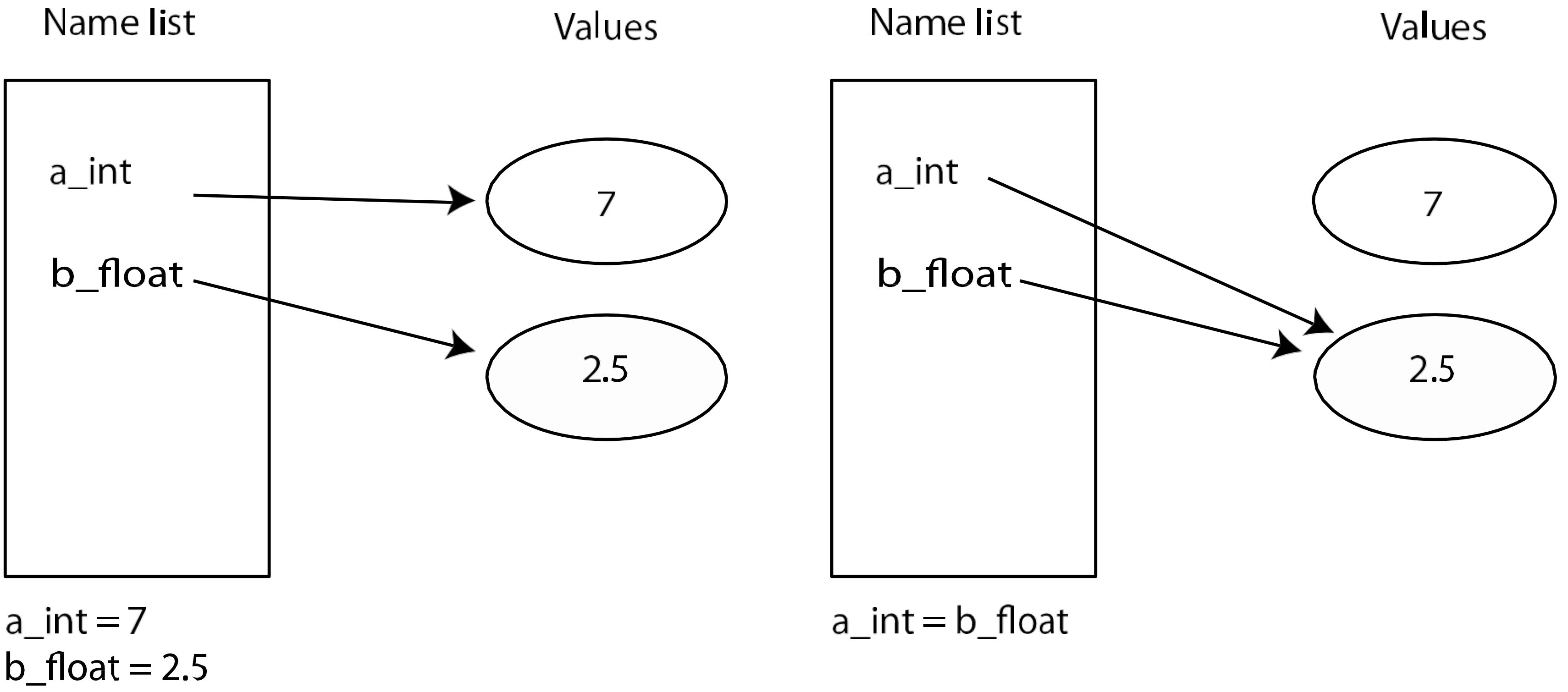


FIGURE 1.2 Namespace before and after the final assignment.

Variables and Types

- Python does not require you to pre-define what type can be associated with a variable
- What type a variable holds can change
- Nonetheless, knowing the type can be important for using the correct operation on a variable. Thus proper naming is important!

What can go on the lhs

- There are limits therefore as to what can go on the lhs of an assignment statement.
- The lhs must indicate a name with which a value can be associated
- must follow the naming rules

`myInt = 5`	Yes
`myInt + 5 = 7`	No

Objects and Types

Python “types”

- integers: **5**
- floats: **1.2**
- booleans: **True**
- strings: "anything" or 'something'
- lists: [,] ['a',1,1.3]
- others we will see

What is a type

- a type in Python essentially defines two things:
  - the internal structure of the type (what is contains)
  - the kinds of operations you can perform
- `'abc'.capitalize()` is a method you can call on strings, but not integers
- some types have multiple elements (collections), we'll see those later

Fundamental Types

- **Integers**
  - `-100, 0, 1000000000000`
- **Floating Point (Real)**
  - `3.14, 10. (= 10.0), .001 (= 0.001), 1.23E-7 (= 0.000000123), 5e9 (= 5000000000.0)`
- **Booleans (True or False values)**
  - `True, False`

    note the capital

Converting types

- A character '1' is not an integer 1. We'll see more on this later, but take my word for it.
- You need to convert the value returned by the `input` command (characters) into an integer
- `int("123")` yields the integer `123`

Type conversion

- `int(some_var)` returns an integer
- `float(some_var)` returns a float
- `str(some_var)` **returns a string**
- should check out what works:
  - int(2.1) → 2, int('2') → 2, but int('2.1') fails
  - float(2) → 2.0, float('2.0') → 2.0, float('2') → 2.0, float(2.0) → 2.0
  - str(2) → '2', str(2.0) → '2.0', str('a') → 'a'

Operators

Operators

- Integer
    - addition and subtraction: +, -
    - multiplication: *
    - division
        - quotient: /
        - integer quotient: //
        - remainder: %
- Floating point
    - add, subtract, multiply, divide: +, -, *, /

Live coding!

Binary operators

The operators addition(+), subtraction(-) and multiplication(*) work normally:

- `a_int = 4`
- `b_int = 2`
- `a_int + b_int` ➔ yields 6
- `a_int - b_int` ➔ yields 2
- `a_int * b_int` ➔ yields 8

Two types of division

The standard division operator (/) yields a floating point result no matter the type of its operands:

- `2/3` ➔ yields `0.6666666666666666`
- `4.0/2` ➔ yields `2.0`

Integer division (//) yields only the integer part of the divide (its type depends on its operands):

- `2//3` ➔ `0`
- `4.0//2` ➔ `2.0`

Modulus Operator

The modulus operator (`%`) give the integer remainder of division:

- `5 % 3` ➔ `2`
- `7.0 % 3` ➔ `1.0`

Again, the type of the result depends on the type of the operands.

Mixed Types

What is the difference between `42` and `42.0` ?

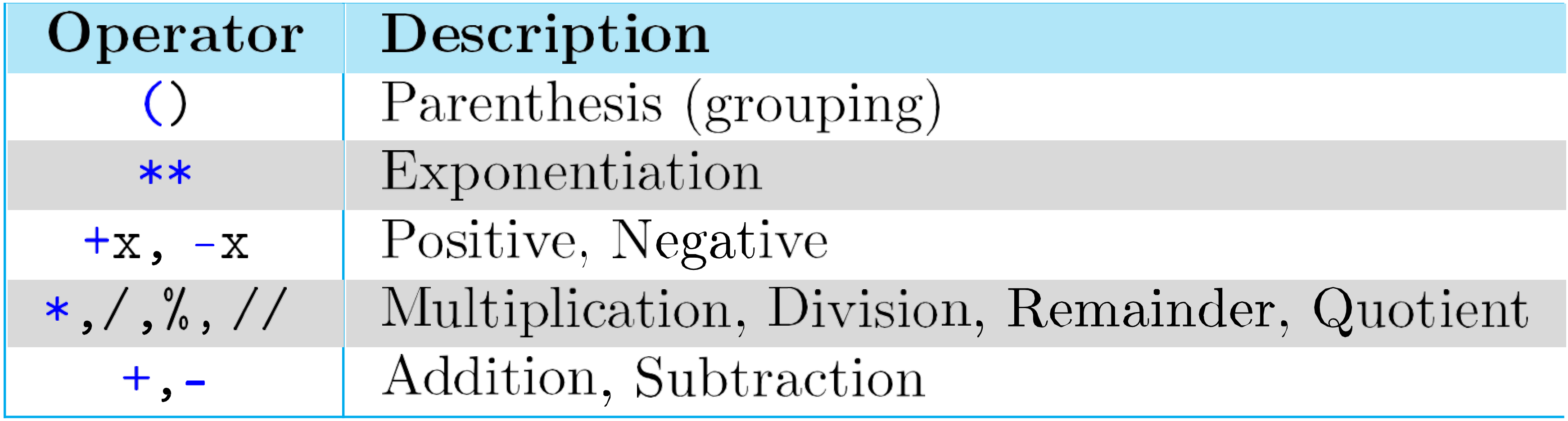
- their types: the first is an integer, the second is a float

What happens when you mix types:

- done so no information is lost

```
42 * 3 → 126
42.0 * 3 → 126.0
```

Order of operations and parentheses



Operator	Description
()	Parenthesis (grouping)
**	Exponentiation
+x, -x	Positive, Negative
*,/,%, //	Multiplication, Division, Remainder, Quotient
+,-	Addition, Subtraction

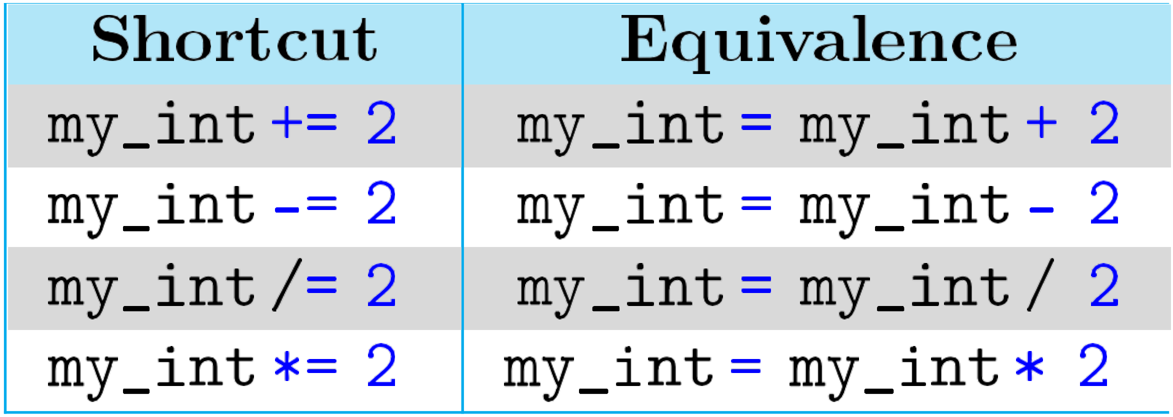
- Precedence of *,/ over +,- is the same
- Remember, parentheses always takes precedence

Live coding!

Augmented assignment

Shortcuts can be distracting, but one that is often used is augmented assignment:

- combines an operation and reassignment to the same variable
- useful for increment/decrement



Shortcut	Equivalence
my_int += 2	my_int = my_int + 2
my_int -= 2	my_int = my_int - 2
my_int /= 2	my_int = my_int / 2
my_int *= 2	my_int = my_int * 2

Reference

- The Practice of Computing Using Python, 3rd ed., Punch & Enbody, Pearson Education, Inc., 2017