

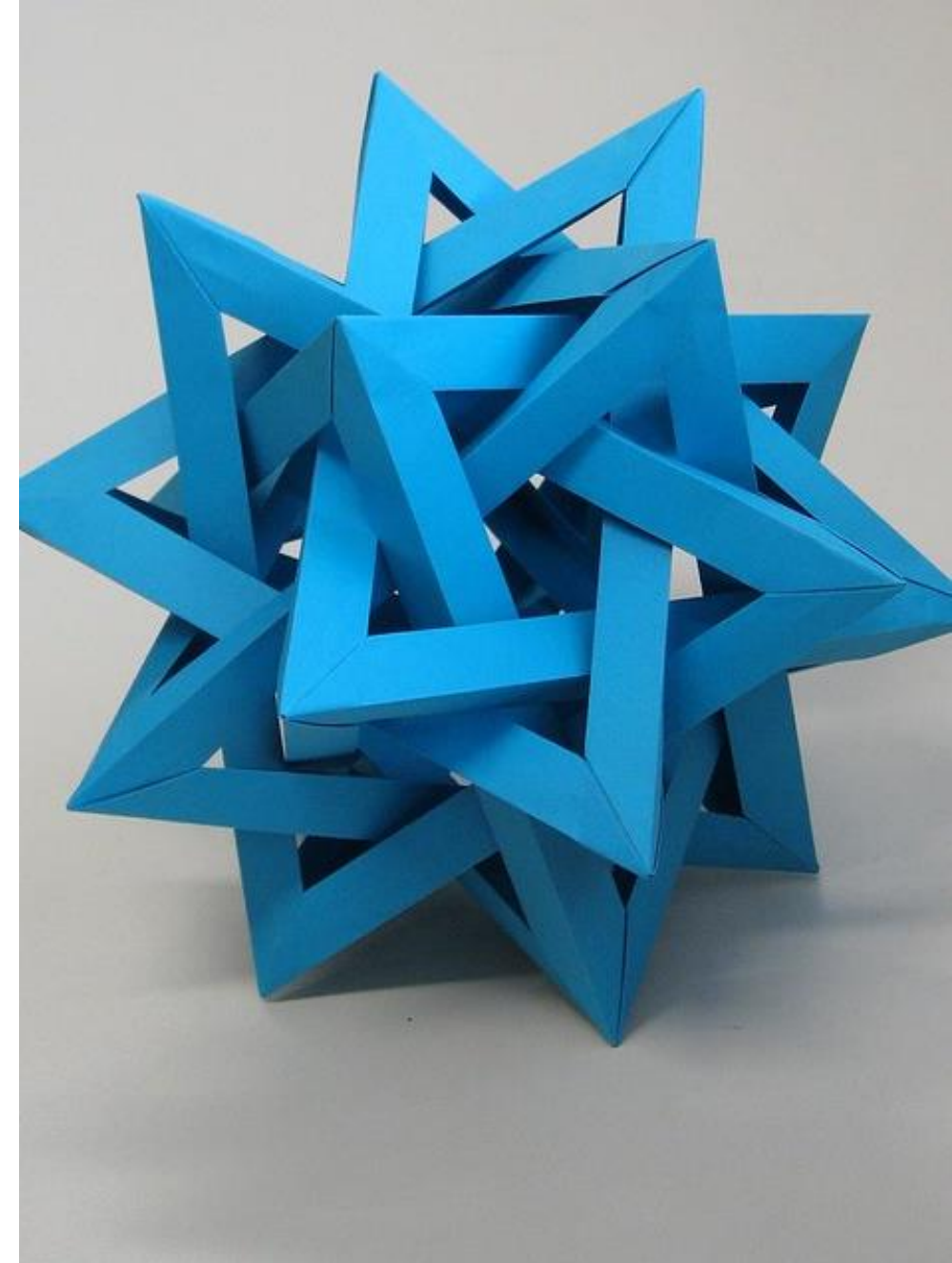


# Unit P2: Numbers and Strings

VARIABLES, VALUES, TYPES, EXPRESSIONS



Chapter 2



# Introduction

- **Numbers** and character **strings** (text sequences) are important data types in any Python program
  - These are also the building blocks we use to build more complex data structures
- In this Unit, you will learn how to work with numbers and text.
- We will write several simple programs that use them

# Unit Goals

- To **declare** and **initialize** variables and constants
- To understand the properties and limitations of **integers and floating-point numbers**
- To appreciate the importance of **comments** and good code layout
- To write arithmetic expressions and assignment statements
- To create programs that read, and process inputs, and display the results
- To learn how to use Python strings

# Variables

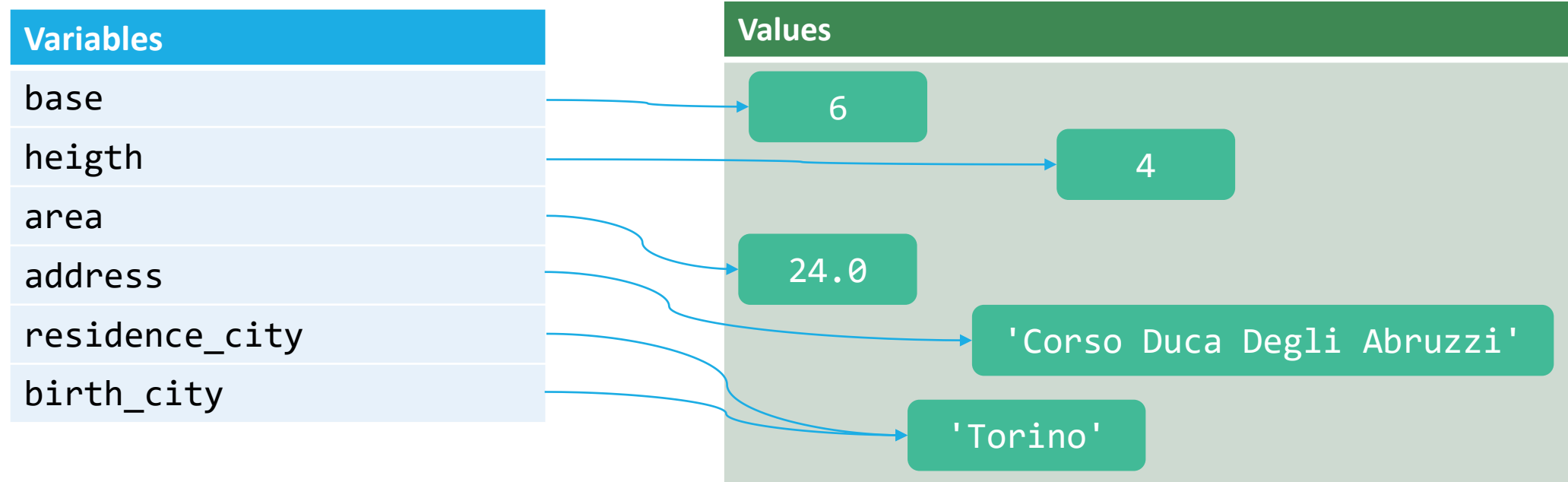
---



2.1

# Variables

- A **variable** is a named storage location in a computer program, that refers to a specific **value**
- There are many different types of **values**, each type used to store different things



# Variables

- You 'define' a variable by telling the interpreter:
  - What **name** you assign to the variable (and you will later use to refer to it)
  - The initial **value** of the variable

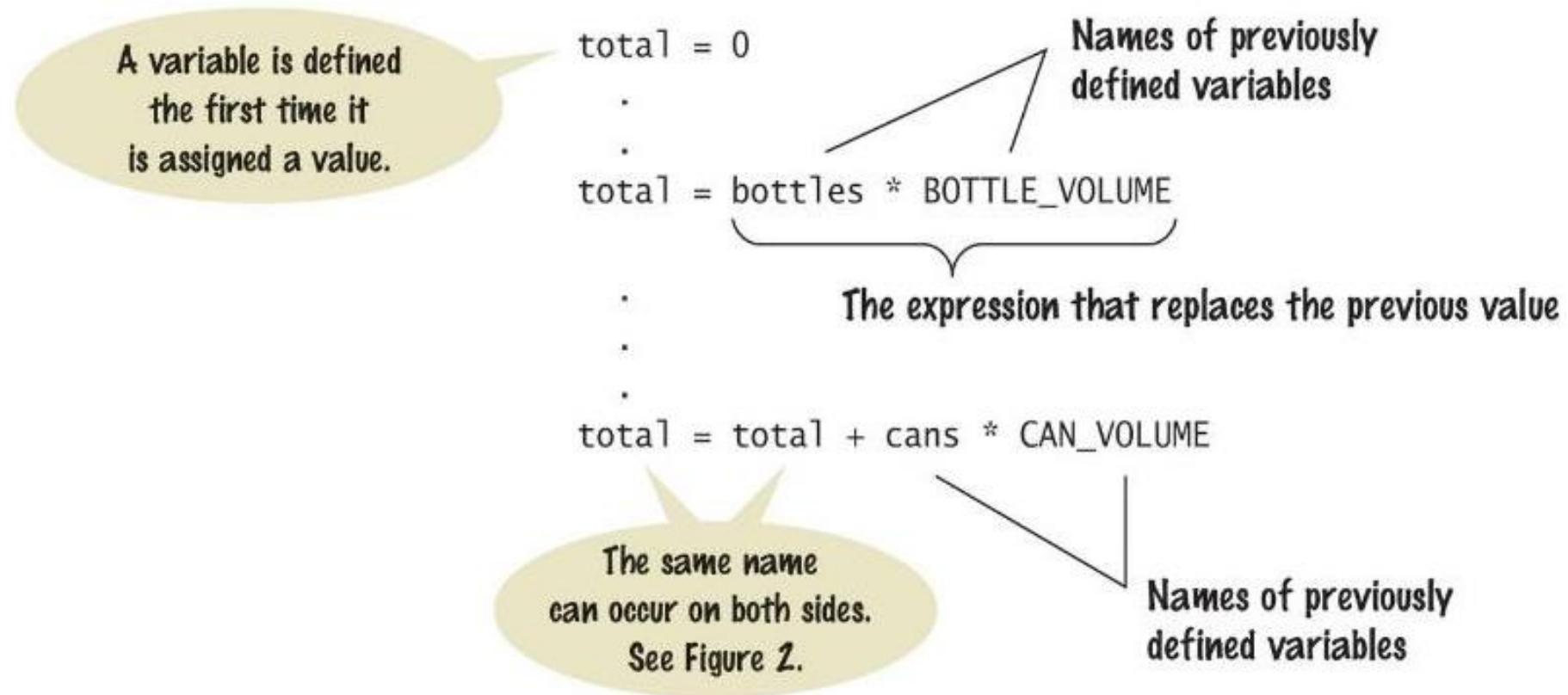
```
cans = 4    # defines & initializes the variable cans
```

- You use an **assignment statement** to place a value into a variable
  - Initial value or new value (that replaces a previous value)

```
cans = 7    # changes the value
```

# Variable Definition

- To **define** a variable, you must specify a **name** and an **initial value**.
  - `cans = 4`    # defines & initializes the variable cans



# Visualizing variables: <http://pythontutor.com/>

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1 # Constants
2
3 BOTTLE_VOLUME = 0.75
4 CAN_VOLUME = 0.33
5
6
7 # Input data
8
9 bottles = 4
10 cans = 6
11
12 # Computation
13
14 total = 0
15
16 total = bottles * BOTTLE_VOLUME
17
18 total = total + cans * CAN_VOLUME
```

→ line that just executed  
→ next line to execute

Done running (7 steps)

Frames

Global frame

- BOTTLE\_VOLUME → float 0.75
- CAN\_VOLUME → float 0.33
- bottles → int 4
- cans → int 6
- total → float 4.98

Objects

Variable names

Values and types

Variable	Type	Value
BOTTLE_VOLUME	float	0.75
CAN_VOLUME	float	0.33
bottles	int	4
cans	int	6
total	float	4.98

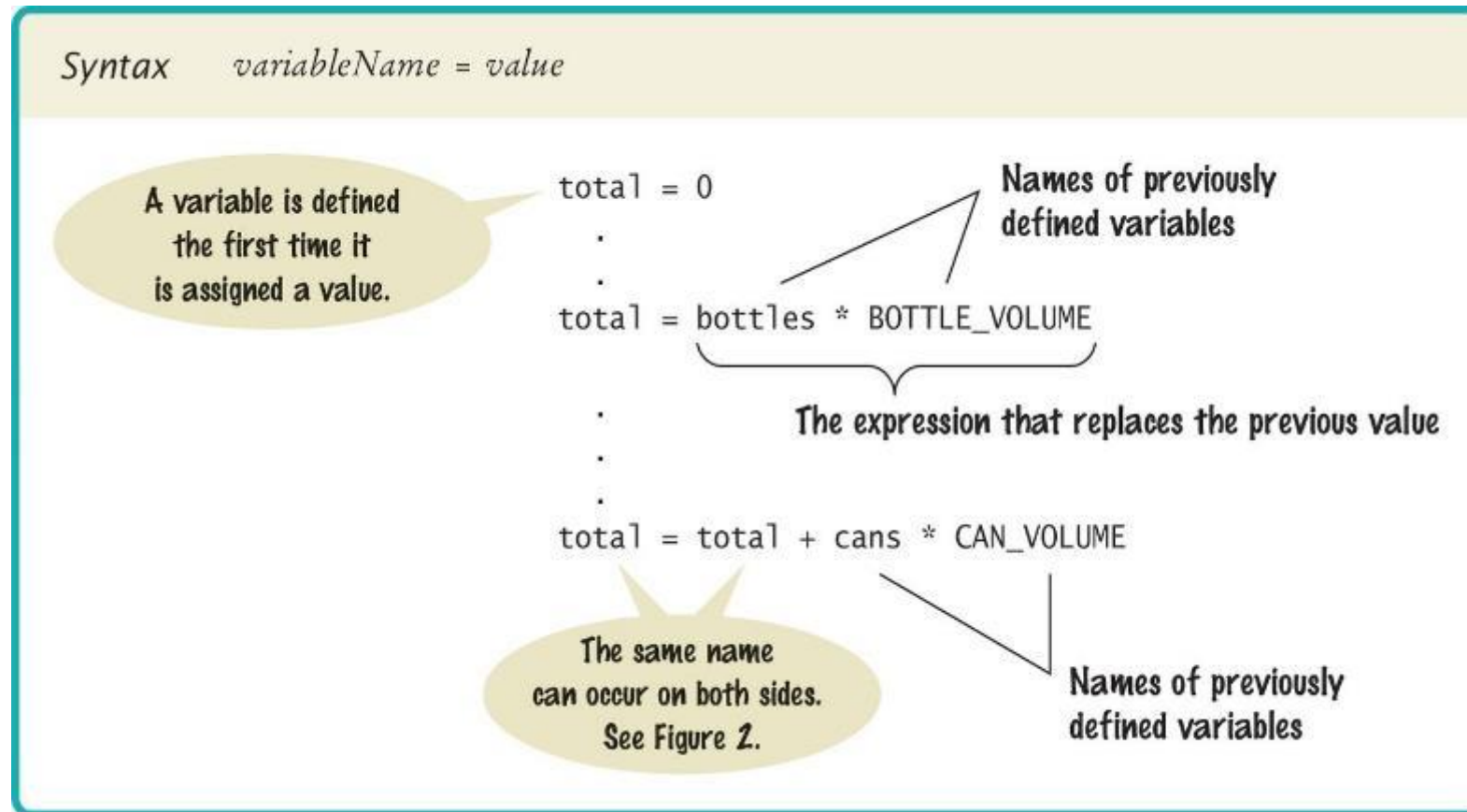


# The assignment statement

- Use the **assignment statement** `=` to place a new value into a variable: the variable name will now refer to the new value
  - `cans = 4` # define cans and initialize to 4
  - `cans = 6` # **change the value** referred by cans, now refers to the value 6
- Beware: the `=` sign is **NOT** used for comparison:
  - It copies the value on the right side into the variable on the left side
  - You will learn about the comparison operator in the next chapter

# Assignment syntax

- The value computed on the right of the '=' sign is assigned to the variable on the left



# An example: soda deal

- Soft drinks are sold in cans and bottles. A store offers a six-pack of 12-ounce cans for the same price as a two-liter bottle. Which should you buy? (12 fluid ounces equal approximately 0.355 liters.)

List of variables:

Number of cans per pack

Ounces per can

Ounces per bottle

Type of Number

Whole number

Whole number

Number with fraction

# Why different types?

- We saw three different types of data:

- A **whole** number (no fractional part)      7      (**integer** or int)
- A number with a **fraction** part      8.88      (**float**)
- A sequence of **characters**      "Bob"      (**string**)

- The **type** is associated with the **value**, not the variable

- cansPerPack = 6      # int
- canVolume = 12.0      # float

# Updating a Variable (assigning a value)

- If an existing variable is assigned a new value, that value replaces the previous contents of the variable.

- For example:

- cansPerPack = 6

1 2

- cansPerPack = 8

3

1 Because this is the first assignment, the variable is created.

cansPerPack =

2 The variable is initialized.

cansPerPack =

3 The second assignment overwrites the stored value.

cansPerPack =

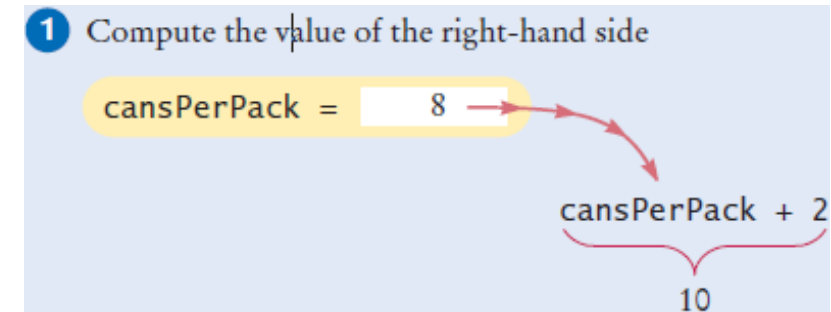
# Updating a Variable (computed)

- Executing the Assignment:

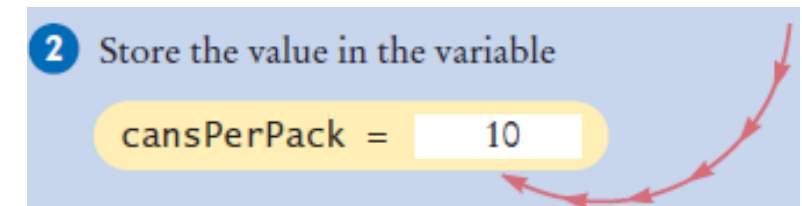
- $\text{cansPerPack} = \text{cansPerPack} + 2$

- Step by Step:

- Step 1: **Calculate** the **right** hand side of the assignment.  
Find the value of `cansPerPack`, and add 2 to it.



- Step 2: **Store the result** in the variable named on the left side of the assignment operator



# Undefined Variables

- You must **define** (and initialize) a variable **before** you use it: (i.e. it must be defined somewhere above the line of code where you first use the variable)
  - `canVolume = 12 * literPerOunce`
  - `literPerOunce = 0.0296`
  - Will cause a **NameError: name 'literPerOunce' is not defined**
- The correct order for the statements is:
  - `literPerOunce = 0.0296`
  - `canVolume = 12 * literPerOunce`

# Data types

- The data **type** is associated with the **value** and not the variable.  
Therefore:
- A variable can be assigned **different values of different types**, at different places in a program

○ `taxRate = 5`                      `# an int`

*Then later...*

○ `taxRate = 5.5`                      `# a float`

*And then*

○ `taxRate = "Non-taxable"`        `# a string`

- If you use a variable and it has an unexpected type an error will occur in your program



# Example...

- Open PyCharm (or repl.it) and create a new project
- # Testing different types in the same variable  
taxRate = 5 # int  
print(taxRate)  
taxrate = 5.5 # float  
print(taxRate)  
taxRate = "Non-taxable" # string  
print(taxRate)



# But...

- `taxRate = "Non-taxable" # string`  
`print(taxRate)`  
`print(taxRate + 5)`
  - Generates: **TypeError**: can only concatenate str (not "int") to str
- `print(taxRate + "??")`
  - Prints Non-taxable??
- So...
  - Once you have initialized a variable with a value of a particular type you should take great care to keep storing values of the same type in the variable

# Caveats

- You should try the operations that are valid according to the current value of the variable, only
  - Remember which type each variable refers to
- When you use the “+” operator with strings the second argument is concatenated to the end of the first, but both arguments must be strings
  - We’ll cover string operations in more detail later in this chapter

# Table 1: Number Literals in Python






Table 1 Number Literals in Python		
Number	Type	Comment
6	int	An integer has no fractional part.
-6	int	Integers can be negative.
0	int	Zero is an integer.
0.5	float	A number with a fractional part has type float.
1.0	float	An integer with a fractional part .0 has type float.
1E6	float	A number in exponential notation: $1 \times 10^6$ or 1000000. Numbers in exponential notation always have type float.
2.96E-2	float	Negative exponent: $2.96 \times 10^{-2} = 2.96 / 100 = 0.0296$
 100,000		<b>Error:</b> Do not use a comma as a decimal separator.
 3 1/2		<b>Error:</b> Do not use fractions; use decimal notation: 3.5.

# Naming variables

## ■ Use These Simple Rules

- Variable names must start with a letter or the underscore ( \_ ) character
  - Continue with letters (upper or lower case), digits or the underscore
- You cannot use other symbols (? or %, ...) nor spaces
- Separate words with 'camelCase' notation
  - Use upper case letters to signify word boundaries
- Don't use 'reserved' Python words (see Appendix C, pages A6 and A7)

# Table 2: Variable Names in Python

Table 2 Variable Names in Python	
Variable Name	Comment
canVolume1	Variable names consist of letters, numbers, and the underscore character.
x	In mathematics, you use short variable names such as $x$ or $y$ . This is legal in Python, but not very common, because it can make programs harder to understand (see Programming Tip 2.1 on page 36).
 CanVolume	<b>Caution:</b> Variable names are case sensitive. This variable name is different from canVolume, and it violates the convention that variable names should start with a lowercase letter.
 6pack	<b>Error:</b> Variable names cannot start with a number.
 can volume	<b>Error:</b> Variable names cannot contain spaces.
 class	<b>Error:</b> You cannot use a reserved word as a variable name.
 1tr/fl.oz	<b>Error:</b> You cannot use symbols such as / or.

# Tip: Use Descriptive Variable Names

- Choose descriptive variable names
- Which variable name is more self descriptive?
  - `canVolume = 0.35`
  - `cv = 0.355`
- This is particularly important when programs are written by more than one person.

# Constants

- In Python a **constant** is a variable whose value *should not be changed* after it's assigned an *initial* value.
- It is a good practice to **use all caps** when naming constants
  - `BOTTLE_VOLUME = 2.0`
- It is good style to use named constants to explain numerical values to be used in calculations: Which is clearer?
  - `totalVolume = bottles * 2`
  - `totalVolume = bottles * BOTTLE_VOLUME`
- A programmer reading the first statement may not understand the significance of the “2”
- Python will let you change the value of a constant
  - Just because you can do it, doesn't mean you should do it



# Constants: Naming & Style

- It is customary to use all UPPER\_CASE letters for constants to distinguish them from variables.
  - It is a nice visual way cue
- `BOTTLE_VOLUME = 2`      # Constant
- `MAX_SIZE = 100`      # Constant
- `taxRate = 5`      # Variable

# Python comments

- Use comments at the beginning of each program, and to clarify details of the code
- Comments are a courtesy to others and a way to document your thinking
  - Comments to add explanations for humans who read your code
- The compiler ignores comments
  - Other programmers will read them
  - Even yourself, tomorrow

*Documentation is a love letter that you write to your future self. Damian Conway (2005). "Perl Best Practices", p.153, "O'Reilly Media, Inc."*

# Commenting Code



```
##  
# This program computes the volume (in liters) of a six-pack of soda  
# cans and the total volume of a six-pack and a two-liter bottle  
#  
  
# Liters in a 12-ounce can  
CAN_VOLUME = 0.355  
  
# Liters in a two-liter bottle.  
BOTTLE_VOLUME = 2  
  
# Number of cans per pack.  
cansPerPack = 6  
  
# Calculate total volume in the cans.  
totalVolume = cansPerPack * CAN_VOLUME  
print("A six-pack of 12-ounce cans contains", totalVolume, "liters.")  
  
# Calculate total volume in the cans and a 2-liter bottle.  
totalVolume = totalVolume + BOTTLE_VOLUME  
print("A six-pack and a two-liter bottle contain", totalVolume, "liters.")
```

# Arithmetics

---



2.2

# Basic Arithmetic Operations

- Python supports all basic arithmetic operations:

- Addition  $+$
- Subtraction  $-$
- Multiplication  $*$
- Division  $/$
- Exponent  $**$

- Use parentheses to write expressions

$$\frac{a+b}{2} \Rightarrow (a + b) / 2$$

$$b \times \left(1 + \frac{r}{100}\right)^n \Rightarrow b * ((1 + r / 100) ** n)$$

- Precedence is similar to Algebra:

- PEMDAS
  - Parenthesis, Exponent, Multiply/Divide, Add/Subtract

# Mixing numeric types

- If you mix **integer** and **floating-point** values in an arithmetic expression, **the result is a floating-point value**.
- `7 + 4.0`      # Yields the floating value `11.0`
- `4` and `4.0` are different data types, for a computer
- Remember:
  - If you mix strings with integer or floating-point values the result is an error

# Floor division

- When you divide two integers with the `/` operator, you get a floating-point value.
  - For example, `7 / 4` yields `1.75`
- We can also perform **floor division** using the `//` operator.
  - The “`//`” operator computes the quotient and discards the fractional part
  - For example, `7 // 4` evaluates to `1`
    - In fact, 7 divided by 4 is 1.75 with a fractional part of 0.75, which is discarded.
  - Only for integer numbers

# Calculating a remainder

- If you are interested in the **remainder** of dividing two integers, use the “**%**” operator (called **modulus**):
  - `remainder = 7 % 4`
- The value of remainder will be 3
- Sometimes called “modulo division”
- Only for integer numbers



# Example

```
# Convert pennies to dollars and cents
pennies = 1729

dollars = pennies // 100  # Calculates the number
of dollars

cents = pennies % 100     # Calculates the number
of pennies

print("I have", dollars, "and", cents, "cents")
```



# Integer Division and Remainder Examples

Table 3 Floor Division and Remainder		
Expression (where $n = 1729$ )	Value	Comment
$n \% 10$	9	For any positive integer $n$ , $n \% 10$ is the last digit of $n$ .
$n // 10$	172	This is $n$ without the last digit.
$n \% 100$	29	The last two digits of $n$ .
$n \% 2$	1	$n \% 2$ is 0 if $n$ is even, 1 if $n$ is odd (provided $n$ is not negative)
$-n // 10$	-173	-173 is the largest integer $\leq -172.9$ . We will not use floor division for negative numbers in this book.

# Calling functions

- Recall that a **function** is a collection of programming instructions that carry out a particular task.
- The `print()` function can display information, but there are **many other functions available** in Python
  - You will learn to call the available functions, and to create your own
- When calling a function you must provide the correct number of **arguments** (also called **parameters**)
  - The program will generate an error message if you don't

# Calling functions that return a value

- **Most functions return a value.** That is, when the function completes its task, it passes a value back to the point where the function was called.
- For example:
  - The call `abs(-173)` returns the value 173.
  - The value **returned** by a function can be **used in an expression** or can be **stored in a variable**:
    - `distance = abs(x)`
- You can use a function call as an argument to the print function
  - `print(abs(-173))`

# Built-in Mathematical Functions

Table 4 Built-in Mathematical Functions

Function	Returns
<code>abs(<math>x</math>)</code>	The absolute value of $x$ .
<code>round(<math>x</math>)</code> <code>round(<math>x</math>, <math>n</math>)</code>	The floating-point value $x$ rounded to a whole number or to $n$ decimal places.
<code>max(<math>x_1</math>, <math>x_2</math>, ..., <math>x_n</math>)</code>	The largest value from among the arguments.
<code>min(<math>x_1</math>, <math>x_2</math>, ..., <math>x_n</math>)</code>	The smallest value from among the arguments.

# Python libraries (modules)

- A **library** is a collection of code (e.g., functions, constants, data types), written and compiled by someone else, that is ready for you to use in your program
  - Always check the documentation before using
- The **standard library** is a library that is considered part of the language and is **included** with any Python system
  - <https://docs.python.org/3/library/index.html>
- Libraries (including the standard library) are organized in **modules**
  - Related functions and data types are grouped into the same module.
  - Functions defined in a module must be explicitly loaded into your program before they can be used

# Libraries, modules, functions

- Built-in functions
  - Always available (such as `print()`, `abs()`, ...)
- Standard library
  - Part of every Python installation
  - Organized in *modules*
  - Each module contains many functions
  - Before using the functions, you must *import the module*
- Additional libraries
  - Not part of Python
  - Must be downloaded/installed in your project (using the IDE or with command line tools)
  - After being installed, they can be imported and functions may be used

# Built-in functions (always available)

Built-in Functions				
<code>abs()</code>	<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>
<code>all()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>any()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>ascii()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>bin()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bool()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>breakpoint()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	

<https://docs.python.org/3/library/functions.html>



# Standard Library functions

- The `sqrt()` function, which computes the square root of its argument, is in the 'math' module of the standard library
  - <https://docs.python.org/3/library/math.html>

```
# First include this statement at the top of your  
# program file.
```

```
from math import sqrt
```

```
# Then you can simply call the function as  
y = sqrt(x)
```

# Some Functions from the Math Module

Table 5 Selected Functions in the math Module

Function	Returns
<code>sqrt(x)</code>	The square root of $x$ . ( $x \geq 0$ )
<code>trunc(x)</code>	Truncates floating-point value $x$ to an integer.
<code>cos(x)</code>	The cosine of $x$ in radians.
<code>sin(x)</code>	The sine of $x$ in radians.
<code>tan(x)</code>	The tangent of $x$ in radians.
<code>exp(x)</code>	$e^x$
<code>degrees(x)</code>	Convert $x$ radians to degrees (i.e., returns $x \cdot 180/\pi$ )
<code>radians(x)</code>	Convert $x$ degrees to radians (i.e., returns $x \cdot \pi/180$ )
<code>log(x)</code> <code>log(x, base)</code>	The natural logarithm of $x$ (to base $e$ ) or the logarithm of $x$ to the given <i>base</i> .

```
from math import xxxxx
```

# Importing modules

- Three ways to import functions from modules:
  - `from math import sqrt, sin, cos`  
# imports listed functions
  - `from math import *`  
# imports all functions from the module
  - `import math`  
# imports the module and gives access to all functions
- If you use the last style you have to add the module name and a “.” before each function call
  - `import math`
  - `y = math.sqrt(x)`

# Floating-point to integer conversion

- You can use the function `int()` and `float()` to **convert** between integer and floating-point values:  

```
balance = total + tax    # balance: float  
dollars = int(balance)  # dollars: integer
```
- You lose the fractional part of the floating-point value (no rounding occurs)

# Floating-point to integer conversion

Function	Definition
<code>math.floor(x)</code>	Return the floor of $x$ , the largest integer less than or equal to $x$
<code>math.ceil(x)</code>	Return the ceiling of $x$ , the smallest integer greater than or equal to $x$ .
<code>math.trunc(x)</code>	Return the real value $x$ truncated to an integer ( <i>not</i> the same as floor, for negative numbers!)
<code>round(x, d)</code>	Return $x$ rounded to $d$ digits precision after the decimal point. If $d$ is omitted or is None, it returns the nearest integer to its input.
<code>int(x)</code>	Return an integer object constructed from a number or string $x$ . For floating point numbers, this truncates towards zero. For strings, it converts the string to an integer

# Roundoff Errors / Rounding Errors

- Floating point values are not exact
  - This is a limitation of binary values; not all floating-point numbers have an exact representation
- Try this:

```
price = 4.35
quantity = 100
total = price * quantity
# Should be 100 * 4.35 = 435.00
print(total) # prints 434.999999999999994
```
- You can deal with roundoff errors by
  - Rounding to the nearest integer (see Section 2.2.4)
  - Displaying a fixed number of digits after the decimal separator (see Section 2.5.3)
  - Defining a “tolerance” EPSILON and do only approximate comparisons (see Section 3.2, and see `math.isclose()`)

# Arithmetic Expressions

Table 6 Arithmetic Expression Examples		
Mathematical Expression	Python Expression	Comments
$\frac{x + y}{2}$	<code>(x + y) / 2</code>	The parentheses are required; <code>x + y / 2</code> computes $x + \frac{y}{2}$ .
$\frac{xy}{2}$	<code>x * y / 2</code>	Parentheses are not required; operators with the same precedence are evaluated left to right.
$\left(1 + \frac{r}{100}\right)^n$	<code>(1 + r / 100) ** n</code>	The parentheses are required.
$\sqrt{a^2 + b^2}$	<code>sqrt(a ** 2 + b ** 2)</code>	You must import the <code>sqrt</code> function from the <code>math</code> module.
$\pi$	<code>pi</code>	<code>pi</code> is a constant declared in the <code>math</code> module.

# Unbalanced Parentheses

- Consider the expression

$((a + b) * t / 2 * (1 - t)$

- What is wrong with the expression?

- Now consider this expression.

$(a + b) * t) / (2 * (1 - t)$

- This expression has three “(” and three “)”, but it still is not correct

- At any point in an expression the count of “(” must be greater than or equal to the count of “)”
- At the end of the expression the two counts must be the same



# Programming Tips

- Use Spaces in expressions

`totalCans = fullCans + emptyCans`

- Is easier to read than

`totalCans=fullCans+emptyCans`

# Problem Solving Exercise



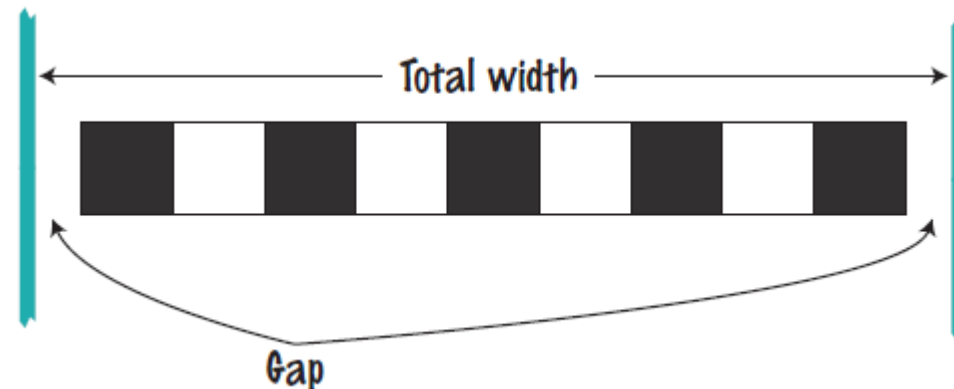
2.3

---

DEVELOP THE ALGORITHM FIRST, THEN WRITE PYTHON CODE

# Problem Solving: First by Hand

- A very important step for developing an algorithm is to first carry out the computations by hand.
  - If you can't compute a solution by hand, how do you write the program?
- Example Problem:
  - A row of black and white tiles needs to be placed along a wall. For aesthetic reasons, the architect has specified that the first and last tile shall be **black**.
  - Your task is to compute the **number of tiles** needed and **the gap** at each end, given the **space** available and the **width** of each tile.

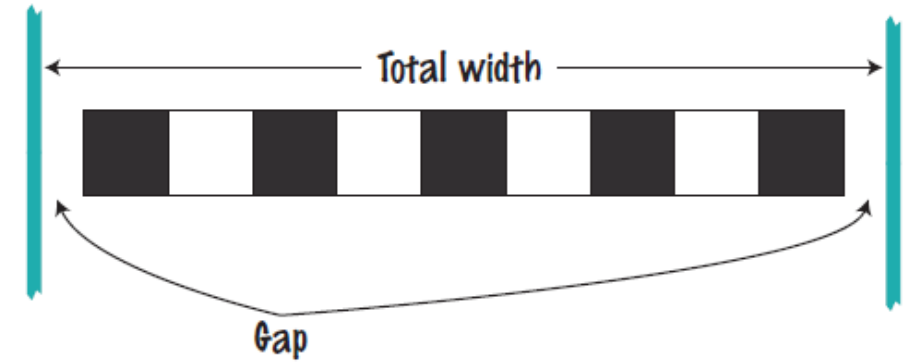


# Start with example values

- Givens
  - Total width: 100 inches
  - Tile width: 5 inches
- Test your values
- Let's see...  $100/5 = 20$ , perfect! 20 tiles. No gap.
  - But wait... BW...BW "...first and last tile shall be black."
- Look more carefully at the problem....
  - Start with one black, then some number of WB pairs



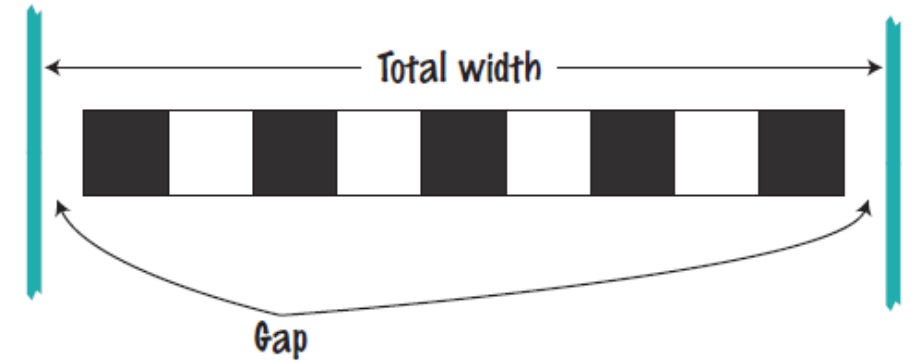
- Observation: each pair is 2x width of 1 tile
  - In our example,  $2 \times 5 = 10$  inches



# Keep applying your solution

- Total width: 100 inches
- Tile width: 5 inches

- Calculate total width of all tiles
  - One black tile: 5 in
  - 9 pairs of BWs: 90 in
  - Total tile width: 95 in
- Calculate gaps (one on each end)
  - $100 - 95 = 5$  in = total gap
  - $5$  in gap /  $2 = 2.5$  in, at each end



# Now devise an algorithm

- Use your example to see how you calculated values
- How many pairs?
  - Note: must be a whole number
  - Integer part of:  $(\text{total width} - \text{tile width}) / (2 \times \text{tile width})$
- How many tiles?
  - $1 + 2 \times \text{the number of pairs}$
- Gap at each end
  - $(\text{total width} - \text{number of tiles} \times \text{tile width}) / 2$

# The algorithm

- Calculate the number of pairs of tiles
  - Number of pairs = integer part of  $(\text{total width} - \text{tile width}) / (2 * \text{tile width})$
- Calculate the number of tiles
  - Number of tiles =  $1 + (2 * \text{number of pairs})$
- Calculate the gap
  - Gap at each end =  $(\text{total width} - \text{number of tiles} * \text{tile width} / 2)$
- Print the number of pairs of tiles
- Print the total number of tiles in the row
- Print the gap

# Strings

---



2.4



# Strings

- Basic definitions:
  - Text consists of **characters**
  - Characters are letters, numbers, punctuation marks, spaces, ....
  - A **string** is a **sequence of characters**
- In Python, string literals are specified by enclosing a sequence of characters within a matching pair of either single or double quotes.  
`print("This is a string.", 'So is this.')`
- By allowing both types of delimiters, Python makes it easy to include an apostrophe or quotation mark within a string.
  - `message = 'He said "Hello"'`
  - Remember to use *matching* pairs of quotes: single with single, double with double

# String Length

- The number of characters in a string is called the **length** of the string.
  - Example, the length of "Harry" is 5
- You can compute the length of a string using Python's `len()` function:  
`length = len("World!") # length is 6`
- A string of length **0** is called the *empty string*. It contains no characters and is written as `""` or `' '`.

# String Concatenation (“+”)

- You can ‘add’ one String onto the end of another

```
firstName = "Harry"
```

```
lastName = "Morgan"
```

```
name = firstName + lastName # HarryMorgan
```

```
print("my name is:", name)
```

- You wanted a space in between the two names?

```
name = firstName + " " + lastName # Harry Morgan
```

# Note

- Using “+” to concatenate strings is an example of a concept called **operator overloading**.
- The “+” operator performs **different** functions, depending on the **types** of the involved **values**:
  - integer + integer → integer addition
  - float + float, float + integer → float addition
  - string + string → string concatenation
  - list + list → list concatenation
  - ...
- But...
  - string + integer → error

## String repetition ("\*")

- You can also produce a string that is the result of repeating a string multiple times
- Suppose you need to print a dashed line
- Instead of specifying a literal string with 50 dashes, you can use the `*` operator to create a string that is comprised of the string `" - "` repeated 50 times

```
dashes = " - " * 50
```

  - results in the string  
" - - - - -"
- The `"*"` operator is also overloaded

# Converting Numbers to Strings

- Use the **str()** function to convert between numbers and strings.

```
balance = 888.88
```

```
dollars = 888
```

```
balanceAsString = str(balance)
```

```
dollarsAsString = str(dollars)
```

```
print(balanceAsString)
```

```
print(dollarsAsString)
```

# Converting Strings to Numbers

- To turn a string containing a number into a numerical value, we use the **int()** and **float()** functions:

```
id = int("1729")  
price = float("17.29")  
print(id)  
print(price)
```

- This conversion is **important** when the strings come from user input

# Strings and Characters

- Strings are sequences of **characters**
- Strings are immutable: they cannot be modified after their creation
  - The same variable can be updated to refer to another string, however



# Extracting a character from a String

- Each character inside a String has an index number:

0	1	2	3	4	5	6	7	8	9
c	h	a	r	s		h	e	r	e

index

character

- The first character has index zero (0)
  - The last character has index  $\text{len}(\text{name}) - 1$
- The `[]` operator returns a character at a given index inside a String:

```
name = "Harry"  
first = name[0]  
last = name[4]
```

0	1	2	3	4
H	a	r	r	y

↑ name[0]

↑ name[4]

# Valid index

- The only valid indexes are from 0 to `len(string)-1`
- Invalid indexes will generate an error
  - `IndexError: string index out of range.`

```
name = 'Bob'  
print(name, 'has length', len(name))  
print(name[0])  
print(name[1])  
print(name[2])  
print(name[3])
```

```
Bob has length 3  
B  
o  
b  
Traceback (most recent call last):  
  File "main.py", line 6, in <module>  
    print(name[3])  
IndexError: string index out of range
```

# Immutability

- Strings are immutable in python
- You can't change the value of a character
  - A **TypeError** occurs

```
name = 'Bob'

print(name[0])

name[0] = 'G'
```

```
B
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    name[0] = 'G'
TypeError: 'str' object does not support
item assignment
```

- **Workaround**: we will learn how to build a new 'updated' string instead of modifying the current one

# String Operations

Try it with  
PythonTutor!

Table 7 String Operations

Statement	Result	Comment
<pre>string = "Py" string = string + "thon"</pre>	string is set to "Python"	When applied to strings, + denotes concatenation.
<pre>print("Please" +       " enter your name: ")</pre>	Prints Please enter your name:	Use concatenation to break up strings that don't fit into one line.
<pre>team = str(49) + "ers"</pre>	team is set to "49ers"	Because 49 is an integer, it must be converted to a string.
<pre>greeting = "H &amp; S" n = len(greeting)</pre>	n is set to 5	Each space counts as one character.
<pre>string = "Sally" ch = string[1]</pre>	ch is set to "a"	Note that the initial position is 0.
<pre>last = string[len(string) - 1]</pre>	last is set to the string containing the last character in string	The last character has position len(string) - 1.

# Characters

- Characters are stored as integer values
  - See the ASCII subset on Unicode chart in Appendix A
  - For example, the letter 'H' has an ASCII value of 72
- Python uses **Unicode** characters
  - Unicode defines over 100,000 characters
  - Unicode was designed to be able to encode text in essentially all written languages
  - <https://home.unicode.org/> and <http://www.unicode.org/charts/>

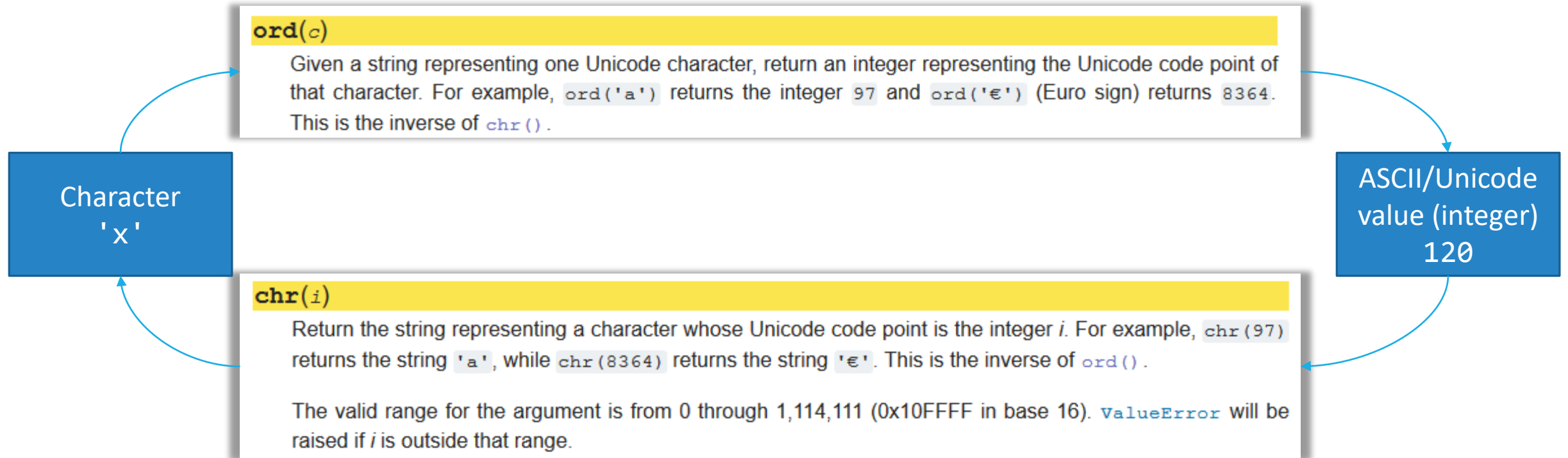
# ASCII codes

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	<b>NUL</b> (null)	32	20	040	&#32;	<b>Space</b>	64	40	100	&#64;	<b>@</b>	96	60	140	&#96;	<b>`</b>
1	1	001	<b>SOH</b> (start of heading)	33	21	041	&#33;	<b>!</b>	65	41	101	&#65;	<b>A</b>	97	61	141	&#97;	<b>a</b>
2	2	002	<b>STX</b> (start of text)	34	22	042	&#34;	<b>"</b>	66	42	102	&#66;	<b>B</b>	98	62	142	&#98;	<b>b</b>
3	3	003	<b>ETX</b> (end of text)	35	23	043	&#35;	<b>#</b>	67	43	103	&#67;	<b>C</b>	99	63	143	&#99;	<b>c</b>
4	4	004	<b>EOT</b> (end of transmission)	36	24	044	&#36;	<b>\$</b>	68	44	104	&#68;	<b>D</b>	100	64	144	&#100;	<b>d</b>
5	5	005	<b>ENQ</b> (enquiry)	37	25	045	&#37;	<b>%</b>	69	45	105	&#69;	<b>E</b>	101	65	145	&#101;	<b>e</b>
6	6	006	<b>ACK</b> (acknowledge)	38	26	046	&#38;	<b>&amp;</b>	70	46	106	&#70;	<b>F</b>	102	66	146	&#102;	<b>f</b>
7	7	007	<b>BEL</b> (bell)	39	27	047	&#39;	<b>'</b>	71	47	107	&#71;	<b>G</b>	103	67	147	&#103;	<b>g</b>
8	8	010	<b>BS</b> (backspace)	40	28	050	&#40;	<b>(</b>	72	48	110	&#72;	<b>H</b>	104	68	150	&#104;	<b>h</b>
9	9	011	<b>TAB</b> (horizontal tab)	41	29	051	&#41;	<b>)</b>	73	49	111	&#73;	<b>I</b>	105	69	151	&#105;	<b>i</b>
10	A	012	<b>LF</b> (NL line feed, new line)	42	2A	052	&#42;	<b>*</b>	74	4A	112	&#74;	<b>J</b>	106	6A	152	&#106;	<b>j</b>
11	B	013	<b>VT</b> (vertical tab)	43	2B	053	&#43;	<b>+</b>	75	4B	113	&#75;	<b>K</b>	107	6B	153	&#107;	<b>k</b>
12	C	014	<b>FF</b> (NP form feed, new page)	44	2C	054	&#44;	<b>,</b>	76	4C	114	&#76;	<b>L</b>	108	6C	154	&#108;	<b>l</b>
13	D	015	<b>CR</b> (carriage return)	45	2D	055	&#45;	<b>-</b>	77	4D	115	&#77;	<b>M</b>	109	6D	155	&#109;	<b>m</b>
14	E	016	<b>SO</b> (shift out)	46	2E	056	&#46;	<b>.</b>	78	4E	116	&#78;	<b>N</b>	110	6E	156	&#110;	<b>n</b>
15	F	017	<b>SI</b> (shift in)	47	2F	057	&#47;	<b>/</b>	79	4F	117	&#79;	<b>O</b>	111	6F	157	&#111;	<b>o</b>
16	10	020	<b>DLE</b> (data link escape)	48	30	060	&#48;	<b>0</b>	80	50	120	&#80;	<b>P</b>	112	70	160	&#112;	<b>p</b>
17	11	021	<b>DC1</b> (device control 1)	49	31	061	&#49;	<b>1</b>	81	51	121	&#81;	<b>Q</b>	113	71	161	&#113;	<b>q</b>
18	12	022	<b>DC2</b> (device control 2)	50	32	062	&#50;	<b>2</b>	82	52	122	&#82;	<b>R</b>	114	72	162	&#114;	<b>r</b>
19	13	023	<b>DC3</b> (device control 3)	51	33	063	&#51;	<b>3</b>	83	53	123	&#83;	<b>S</b>	115	73	163	&#115;	<b>s</b>
20	14	024	<b>DC4</b> (device control 4)	52	34	064	&#52;	<b>4</b>	84	54	124	&#84;	<b>T</b>	116	74	164	&#116;	<b>t</b>
21	15	025	<b>NAK</b> (negative acknowledge)	53	35	065	&#53;	<b>5</b>	85	55	125	&#85;	<b>U</b>	117	75	165	&#117;	<b>u</b>
22	16	026	<b>SYN</b> (synchronous idle)	54	36	066	&#54;	<b>6</b>	86	56	126	&#86;	<b>V</b>	118	76	166	&#118;	<b>v</b>
23	17	027	<b>ETB</b> (end of trans. block)	55	37	067	&#55;	<b>7</b>	87	57	127	&#87;	<b>W</b>	119	77	167	&#119;	<b>w</b>
24	18	030	<b>CAN</b> (cancel)	56	38	070	&#56;	<b>8</b>	88	58	130	&#88;	<b>X</b>	120	78	170	&#120;	<b>x</b>
25	19	031	<b>EM</b> (end of medium)	57	39	071	&#57;	<b>9</b>	89	59	131	&#89;	<b>Y</b>	121	79	171	&#121;	<b>y</b>
26	1A	032	<b>SUB</b> (substitute)	58	3A	072	&#58;	<b>:</b>	90	5A	132	&#90;	<b>Z</b>	122	7A	172	&#122;	<b>z</b>
27	1B	033	<b>ESC</b> (escape)	59	3B	073	&#59;	<b>;</b>	91	5B	133	&#91;	<b>[</b>	123	7B	173	&#123;	<b>{</b>
28	1C	034	<b>FS</b> (file separator)	60	3C	074	&#60;	<b>&lt;</b>	92	5C	134	&#92;	<b>\</b>	124	7C	174	&#124;	<b> </b>
29	1D	035	<b>GS</b> (group separator)	61	3D	075	&#61;	<b>=</b>	93	5D	135	&#93;	<b>]</b>	125	7D	175	&#125;	<b>}</b>
30	1E	036	<b>RS</b> (record separator)	62	3E	076	&#62;	<b>&gt;</b>	94	5E	136	&#94;	<b>^</b>	126	7E	176	&#126;	<b>~</b>
31	1F	037	<b>US</b> (unit separator)	63	3F	077	&#63;	<b>?</b>	95	5F	137	&#95;	<b>_</b>	127	7F	177	&#127;	<b>DEL</b>

Source: [www.LookupTables.com](http://www.LookupTables.com)

For Unicode characters see: <https://unicode-table.com/>

# Character conversions



# Functions vs Methods

- Python is an object-oriented language, and all values are *objects*.
  - Object-oriented programming **is out of the scope** of the course, we will only see some very practical aspect
- Every object may have *methods*, i.e., functions that can be called on *that specific object*, using a syntax **object.method()**
- Example: all strings have an **upper()** method that returns a new string, converted to upper case

```
name = "John Smith"
# Sets uppercaseName to "JOHN SMITH"
uppercaseName = name.upper()
```



# Functions vs Methods – what to Remember

## FUNCTION

- Functions are **general** and can accept arguments of different types
- Functions are called directly, with a list of parameters
  - `func(param)`
- Functions may return a result, that may be stored in a variable
  - `result = func(param)`

## METHOD

- Methods are **specific** to a type of object
  - All strings have a group of methods
  - All integers have a group of methods
  - ...
- Methods are called with the **dot-syntax**
  - `object.method()`
- Methods may return a result, that may be stored in a variable
  - `result = obj.method()`

# Some Useful String Methods

Table 8 Useful String Methods

Method	Returns
<code>s.lower()</code>	A lowercase version of string <i>s</i> .
<code>s.upper()</code>	An uppercase version of <i>s</i> .
<code>s.replace(<i>old</i>, <i>new</i>)</code>	A new version of string <i>s</i> in which every occurrence of the substring <i>old</i> is replaced by the string <i>new</i> .

# String *Escape* Sequences

- How would you print a double quote?
  - Preface the " with a "\" inside the double quoted String  
`print("He said \"Hello\"")`
- OK, then how do you print a backslash?
  - Preface the \ with another \  
`print("C:\\Temp\\Secret.txt")`
- Special characters inside Strings
  - Output a newline with a '\n'  
`print("*\n**\n***")`

```
*  
**  
***
```

# Summary

---

# Summary: variables

- A variable is a storage location with a name.
- When defining a variable, you must specify an initial value.
- By convention, variable names should start with a lower case letter.
- An assignment statement stores a new value in a variable, replacing the previously stored value.

# Summary: operators

- The assignment operator = does not denote mathematical equality.
- Variables whose initial value should not change are typically capitalized by convention.
- The / operator performs a division yielding a value that may have a fractional value.
- The // operator performs an integer division, the remainder is discarded.
- The % operator computes the remainder of a floor division.

# Summary: python overview

- The Python library declares many mathematical functions, such as `sqrt()` and `abs()`
- You can convert between integers, floats and strings using the respective functions: `int()`, `float()`, `str()`
- Python libraries are grouped into modules. Use the `import` statement to use methods from a module.

# Summary: Strings

- Strings are sequences of characters.
- The `len()` function yields the number of characters in a String.
- Use the `+` operator to concatenate Strings; that is, to put them together to yield a longer String.
- In order to perform a concatenation, the `+` operator requires both arguments to be strings. Numbers must be converted to strings using the `str()` function.
- String index numbers are counted starting with 0.



# Summary: Strings

- Use the `[ ]` operator to extract the elements (single characters) of a String.