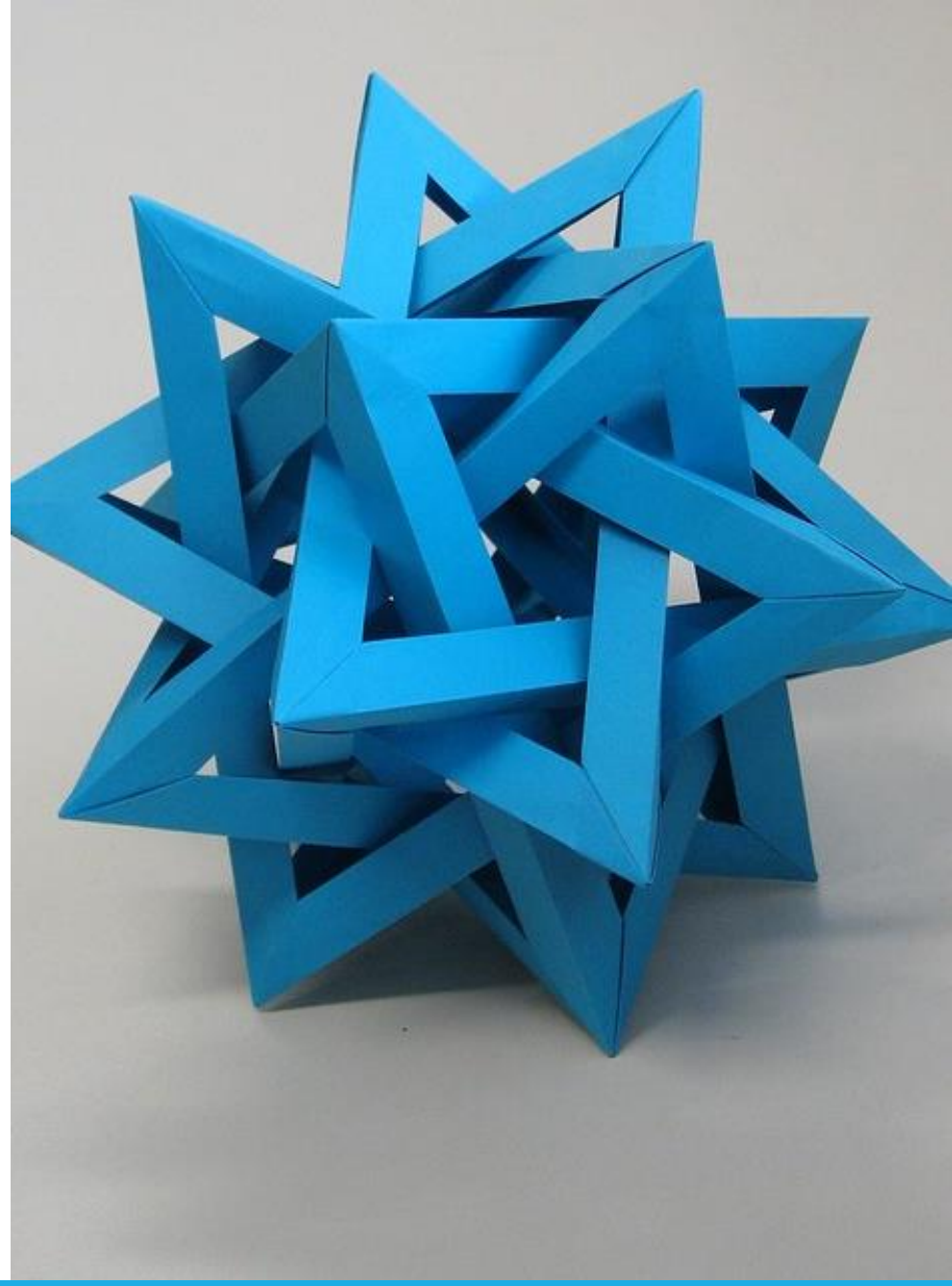
POLITECNICO
DI TORINO
Dipartimento
di Automatica e Informatica

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

We'll see later where the name comes from

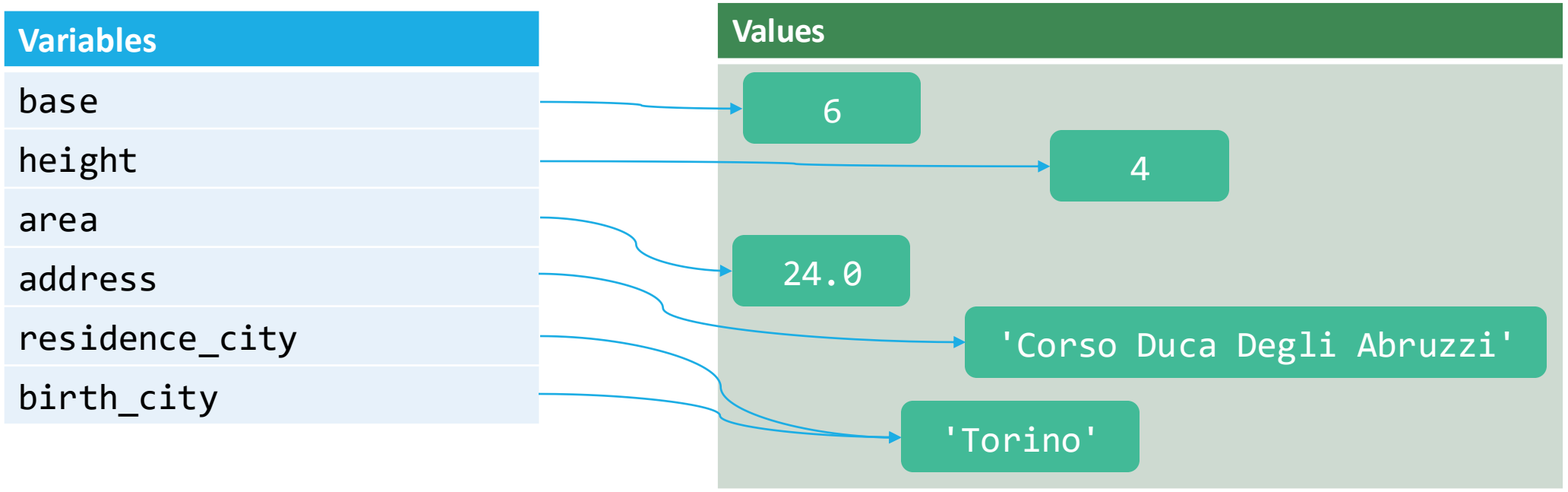
- To **declare** and **initialize** variables and constants
- To understand the properties and limitations of **integers and floating-point numbers**
- To appreciate the importance of comments and good code layout
- To write arithmetic expressions and assignment statements
- To create programs that read,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

Variables

2.1

Variables

- A variable is a **name** that refers to specific data in your program
  - You can think of it as a "**memory location with a name**" (not 100% accurate in Python, but a good learning metaphor)
- There are many different **types** of data: the type defines the possible values the data can assume, and the operations that can be performed



Variables

- **Name** (i.e., a label)

- **Object** (i.e., the data)
  - Names are bound to objects
  - **Different names can be bound to the same object**

- A **variable** is an object with a name

Variables and Names (more formally)

- Name: A sequence of characters
- Type (Class): The definition of possible values and operations that can be performed
- Object: Data interpreted according to the *class*
- Variable: An object that was bound to a name
- In Python there are many different types, that can be grouped in *families*, each type used to store different things
- The Python interpreter always knows the type of all objects
- Names have **no** type — they may be bound to different objects in different moments

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
```

The assignment statement

- Beware: the **=** sign is **NOT** used for comparison:
  - It associates a name to an object!
  - You will learn about the comparison operator in the next chapter

- Read the instruction as follows: first the **right side of the equals**, then the left side
  - **Right Hand Side (RHS):** The expression whose result value will be associated to the variable
  - **Left Hand Side (LHS):** the variable (label) that we will bind to this result.

Visualizing variables: 0.in

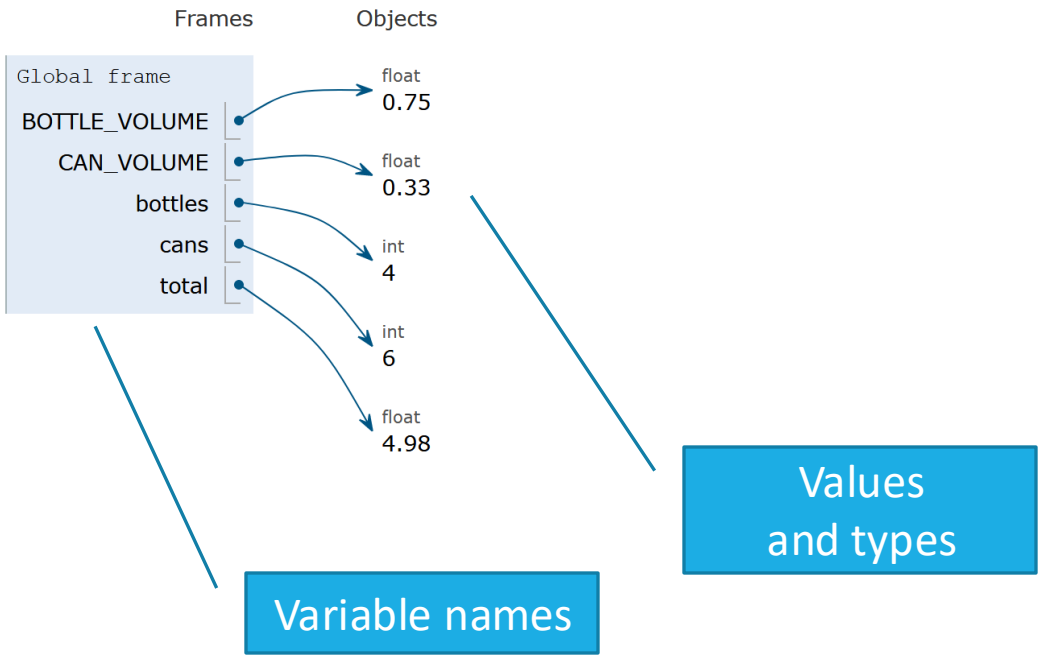
Write code in Python 3.6 (drag lower right corner to resize code editor)

```
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

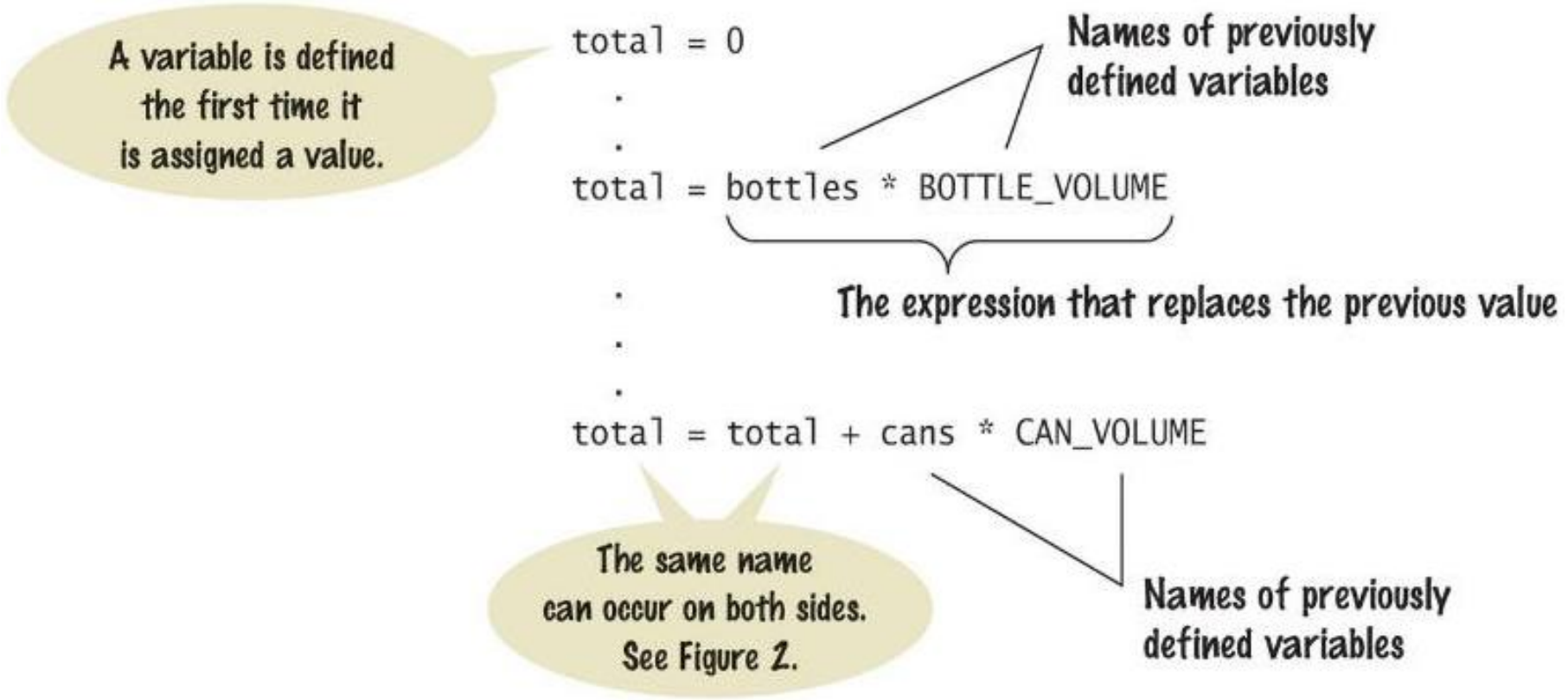
<< First | < Prev | Next > | Last >>

Done running (7 steps)



Variable Definition

- To define a variable, you must specify a name and an initial value.



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:	Type of Number
Number of cans per pack	Whole number
Litres per can	Number with fraction
Litres per bottle	Number with fraction

Why different types?

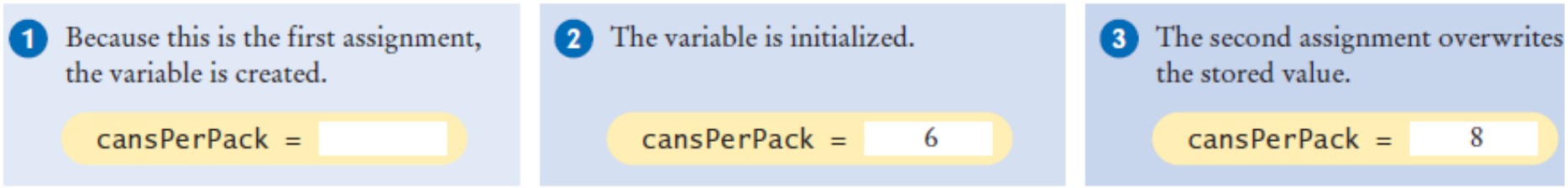
- We saw two different types of data:
  - A whole number (no fractional part) `7` (integer or int)
  - A number with a fraction part `8.88` (float)

- Another useful one…:
  - A sequence of characters `"Bob"` (string)

- The type is associated with the value, not the variable
  - `cansPerPack = 6  # int`
  - `canVolume = 12.0  # float`

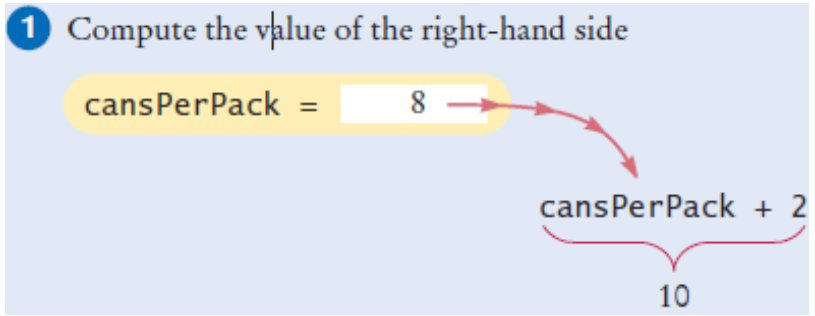
Updating a Variable (assigning a value)

- If an existing variable is assigned a new value, that value replaces the association with the previous value.
- For example:
  - cansPerPack = 6
  - cansPerPack = 8

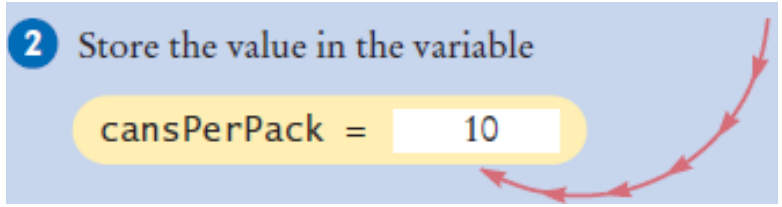


Updating a Variable (computed)

- Executing the Assignment:
  - cansPerPack = 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 **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 VSCode, pythontutor (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)
  ```
  ```

Careful!

- The following program prints `5 , 5 , Non-Taxable`. Why? What's the error?

- ```
  # 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 to use **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: 1×10^6 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`		**Error:** Do not use a comma as a decimal separator.
`3 1/2`		**Error:** 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`	**Caution:** 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`	**Error:** Variable names cannot start with a number.
🚫 `can volume`	**Error:** Variable names cannot contain spaces.
🚫 `class`	**Error:** You cannot use a reserved word as a variable name.
🚫 `ltr/fl.oz`	**Error:** 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 interpreter 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.")
```

Arithmetic

2.2

Basic Arithmetic Operations

- Python supports all basic arithmetic operations:
  - Addition `+`
  - Subtraction `-`
  - Multiplication `*`
  - Division `/`
  - Exponent `**`
- Use parentheses to write expressions

$\frac{a+b}{2}$ ➡ `(a + b) / 2`

$b \times \left(1 + \frac{r}{100}\right)^n$ ➡ `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 or integer division** using the **//** operator.
  - The "**//**" operator computes the quotient and discards the fractional part
  - For example, `7 // 4` evaluates to `1`
  - Only for integer numbers

Calculating a remainder

- If you are interested in the remainder of dividing two integers, use the "%" operator (called modulo or 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 ≤ –172.9. We will not use floor division for negative numbers in this book.

Calling functions

- 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 another function
  - `print(abs(-173))`

Built-in Mathematical Functions

Table 4 Built-in Mathematical Functions

Function	Returns
`abs(`x`)`	The absolute value of x.
`round(`x`)` `round(`x`, `n`)`	The floating-point value x rounded to a whole number or to n decimal places.
`max(`x_1`, `x_2`, ..., `x_n`)`	The largest value from among the arguments.
`min(`x_1`, `x_2`, ..., `x_n`)`	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		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

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

```
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
`math.floor(x)`	Return the floor of *x*, the largest integer less than or equal to *x*
`math.ceil(x)`	Return the ceiling of *x*, the smallest integer greater than or equal to *x*.
`math.trunc(x)`	Return the real value *x* truncated to an integer (*not* the same as int(), for negative numbers!)
`round(x,d)`	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.
`int(x)`	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 434.99999999999994
```

- 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
$\frac{x + y}{2}$	`(x + y) / 2`	The parentheses are required; `x + y / 2` computes $x + \frac{y}{2}$.
$\frac{xy}{2}$	`x * y / 2`	Parentheses are not required; operators with the same precedence are evaluated left to right.
$\left(1 + \frac{r}{100}\right)^n$	`(1 + r / 100) ** n`	The parentheses are required.
$\sqrt{a^2 + b^2}$	`sqrt(a ** 2 + b ** 2)`	You must import the `sqrt` function from the `math` module.
π	`pi`	`pi` is a constant declared in the `math` 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
```

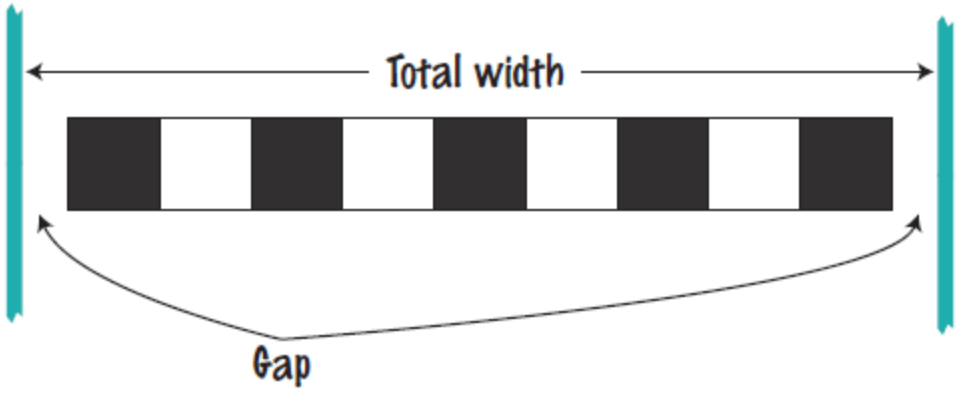
Problem Solving Exercise

2.3

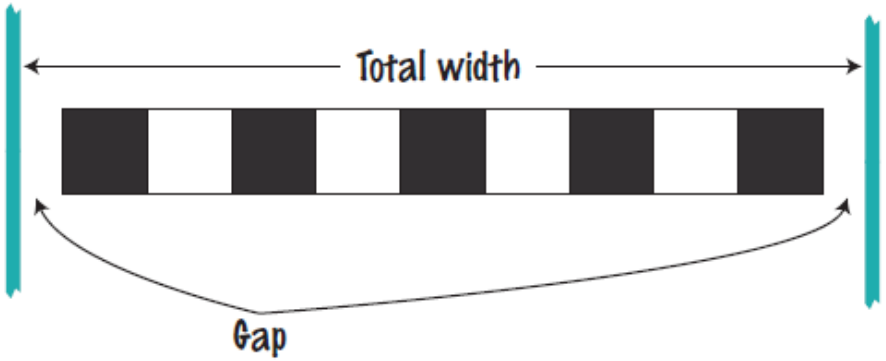
DEVELOP THE ALGORITHM FIRST, THEN WRITE PYTHON CODE

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

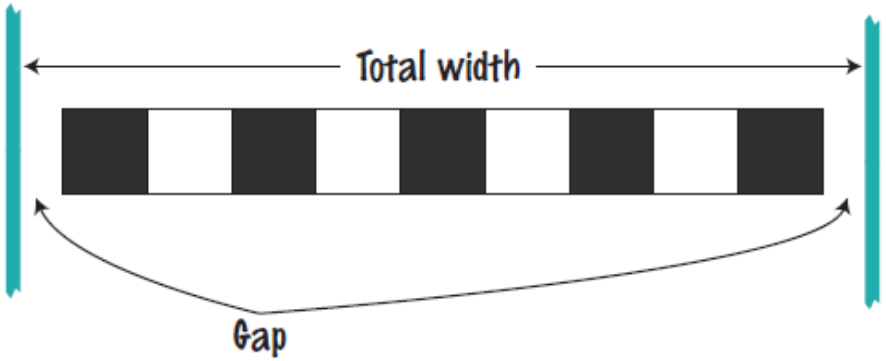


- Givens
  - Total width: 100 inches
  - Tile width: 5 inches
- Test your values
- Let's see… 100/5 = 20, perfect! 20 tiles. No gap.
  - But wait… BW…BW "…first and last tile shall be black.""
- Look more carefully at the problem….
  - Start with one black, then some number of WB pairs

- Observation: each pair is 2x width of 1 tile
  - In our example, 2 x 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: (total width – tile width) / (2 x tile width)
- How many tiles?
  - 1 + 2 x the number of pairs
- Gap at each end
  - (total width – number of tiles x tile width) / 2

The algorithm

- Calculate the number of pairs of tiles
  - Number of pairs = integer part of (total width – tile width) / (2 * tile width)
- Calculate the number of tiles
  - Number of tiles = 1 + (2 * number of pairs)
- Calculate the gap
  - Gap at each end = (total width – number of tiles * tile width / 2)

- Print the total number of tiles in the row
- Print the gap

Python Solution

```
##
#  Computes the number of tiles needed and the gap at each end when
#  placing tiles along a wall.
#

# Define the dimensions.
totalWidth = 100
tileWidth = 5

# Calculate the tiles and gaps.
numberOfPairs = (totalWidth - tileWidth) // (2 * tileWidth)
numberOfTiles = 1 + 2 * numberOfPairs
gap = (totalWidth - numberOfTiles * tileWidth) / 2.0

print("Number of tiles:", numberOfTiles)
print("Gap at each end:", gap)
```

tiles.py

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)
# nothing changed? Try:
print(dollarsAsString + balanceAsString)
```

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 (we'll see it next week)

Conversion errors

- You cannot convert a character's string into a number, it is necessary to have a string made of digits to do so:
- Example:
  - `val = int("Hola")`

```
ValueError: invalid literal for int() with
base 10: 'Hola'
```

Strings and Characters

- Strings are sequences of characters
  - A character is itself a string (of length = 1)
- Strings are immutable: they cannot be modified after their creation
  - The same variable can be updated to refer to another string, however

- **How to access/operate on single characters?**

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 `len(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 slices

- It is possible to extract a part of a string or *slice*
- Given a string:
  - **`name = "never odd or even"`**
- If we want the string slice from character 4 (index 3→'**`e`**') included, to 8 (index 7 → '**`d`**') not included
- **`stringSlice = name[3:7]`**
- The string slice contains characters in indexes 3, 4, 5 and 6.
  - **`"er o"`**

String slices - 2

- Examples
- **`name[ : 6]` → `"never odd or even"`**
  - All the characters included from the first one (index 0) to the 6° one, (index 6 not included)
- **`name[6 : ]` → `"never odd or even"`**
  - Characters from the 7° to the last one included
- **`name[ : ]`**
  - All the characters included, it makes a copy of the string

String portions

- A string portion can be extracted providing *start*, *stop* and *step* indexes
- **`name[3:8:2]` → `"never_odd or even"`**
  - Include all the characters from 3 to 8 using an increment of 2 of the index steps.
- **`name[::2]` → `"never odd or even"`**
  - Include all the characters in even indexes
- **`name[1::2]` → `"never_odd_or even"`**
  - Include all the characters in odd indexes

Characters

- Characters are stored as **integer values**

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

Source: www.LookupTables.com

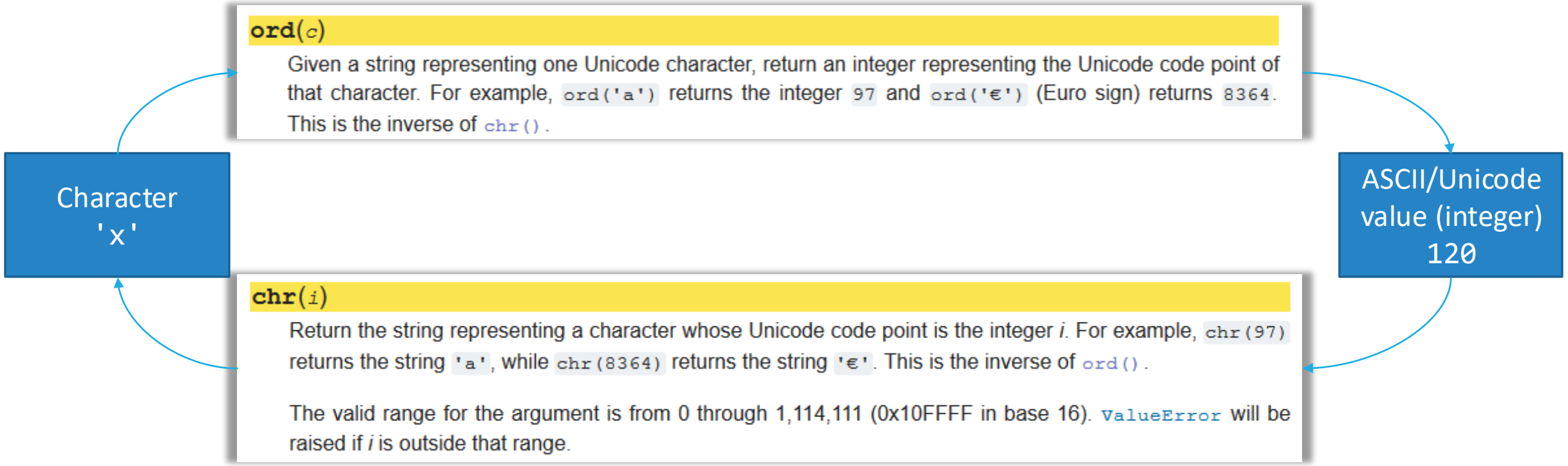
For Unicode characters see: https://unicode-table.com/

Unicode codes (Sample)

```
0061    'a'; LATIN SMALL LETTER A
0062    'b'; LATIN SMALL LETTER B
0063    'c'; LATIN SMALL LETTER C
...
007B    '{'; LEFT CURLY BRACKET
...
2167    'Ⅷ'; ROMAN NUMERAL EIGHT
2168    'Ⅸ'; ROMAN NUMERAL NINE
...
265E    '♞'; BLACK CHESS KNIGHT
265F    '♟'; BLACK CHESS PAWN
...
1F600   '😀'; GRINNING FACE
1F609   '😉'; WINKING FACE
...
```

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 aspects
- 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
  - `Exmp: func(param), print(a)`
- 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
`s.lower()`	A lowercase version of string *s*.
`s.upper()`	An uppercase version of *s*.
`s.replace(old, new)`	A new version of string *s* in which every occurrence of the substring *old* is replaced by the string *new*.

We'll see more methods later ….

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***")
```

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

Escape Sequences

Escape Sequence	Description
\newline	Backslash and newline ignored
\\	Backslash
\'	Single quote
\"	Double quote
\a	ASCII Bell
\b	ASCII Backspace
\f	ASCII Formfeed
\n	ASCII Linefeed
\r	ASCII Carriage Return
\t	ASCII Horizontal Tab
\v	ASCII Vertical Tab
\ooo	Character with octal value ooo
\xHH	Character with hexadecimal value HH

https://www.programiz.com/python-programming/string

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 (**CONSTANTS**) 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
- Use the operator [a:b] to extract slices of the string or [a:b:c] to use a different index step during the extraction.

Thanks

- Part of these slides are [edited versions] of those originally made by **Prof Giovanni Squillero** (Teacher of Course 1)

License

- These slides are distributed under the Creative Commons license “Attribution - Noncommercial - ShareAlike 2.5 (CC BY-NC-SA 2.5)”
- You are free:
  - to reproduce, distribute, communicate to the public, exhibit in public, represent, perform and recite this work
  - to modify this work
- Under the following conditions:
  - **Attribution** — You must attribute the work to the original authors, and you must do so in a way that does not suggest that they endorse you or your use of the work.
  - **Non-Commercial** — You may not use this work for commercial purposes.
  - **Share Alike** — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or equivalent license as this one.
- http://creativecommons.org/licenses/by-nc-sa/2.5/it/