CS303E: Elements of Computers and Programming

Simple Python

Mike Scott
Department of Computer Science
University of Texas at Austin

Adapted from
Professor Bill Young's Slides

Last updated: June 5, 2023

"Once a person has understood the way variables are used in programming, they have understood the quintessence of programming."

-Professor Edsger W. Dijkstra

Simple Program: Body Mass Index

- **B**ody **M**ass **I**ndex or **BMI** is a quick calculation based on height and mass (weight) used by medical professionals to broadly categorize people .
- Formula:

$$\boxed{\mathrm{BMI} = \frac{\mathrm{mass}_{\mathrm{kg}}}{\mathrm{height}_{\mathrm{m}}^{2}} = \frac{\mathrm{mass}_{\mathrm{lb}}}{\mathrm{height}_{\mathrm{in}}^{2}} \times 703}$$

- Quick tool to get a rough estimate if someone is underweight, normal weight, overweight, or obese
- Write an interactive program that gets the name, height, and weight of a user and calculates BMI.

Assignment Statements

An assignment in Python has form:

<variable> = <expression>

This means that variable is *assigned* **value.** i.e., after the assignment, **variable** "contains" **value.**

The equals sign is NOT algebraic equality.
It causes an action! The *expression* on the right is evaluated and the result is assigned to the variable on the left.

```
>>> x = 17.2
>>> y = -39
>>> z = x * y - 2
>>> print( z )
-672.8
```

Variables

A **variable** is a named memory location (in the RAM typically) used to store values. We'll explain shortly how to name variables.

Unlike some programming languages, Python variables do not have fixed data types.

```
// Ccode
int x = 17;        // variable x has type int
x = 5.3;           // illegal
```

```
# Python code
x = 17             # x gets int value 17
x = 5.3            # x gets float value 5.3
```

A variable in Python actually holds a *pointer* to a class object, rather than the object itself.
A variable exists at a particular *address.* Each memory location (4 or 8 bytes typically circa 2021) has an address or location. A number that specifies that location in memory

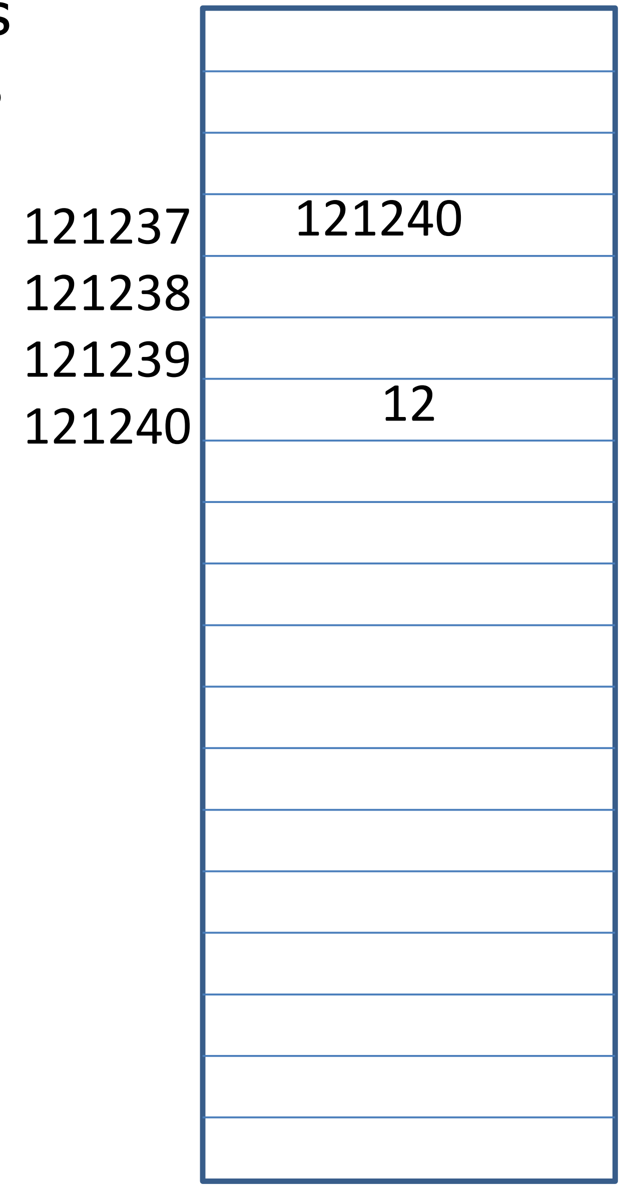
What's a Pointer?

- Also called references, but pointers and references have differences that are beyond the scope of this class.
- A variable exists at a particular *address.* Each memory location (4 or 8 bytes typically circa 2021) has an address or location. A number that specifies that location in memory.
    - Just like the address of a house or building on a street
- So a variable is just a name in our program for a spot in the RAM that stores a value.
- But Python (for reasons we don't want to talk about now) has a bit of " bureaucracy" when a **variable** is bound to a **value**

x = 12

\# let's assume the variable x is at memory

\# location 121237



Types in Python

Is it correct to say that there are no types in Python?

Yes and no. It is best to say that Python is "dynamically typed." Variables in Python are untyped, but values have associated data types (actually classes). In some cases, you can convert one type to another.

Most programming languages assign types to both variables and values. This has its advantages and disadvantages.

What do you think the advantages are of requiring variables to declare the data type of a variable?

Variables and Assignments

You can create a new variable in Python by assigning it a value. *You don't have to declare variables' types, as in many other programming languages.*

```
>>> x = 3                # creates x, assigns int
>>> print(x)
3
>>> x = "abc"            # re-assigns x a string
>>> print(x)
abc
>>> x = 3.14             # re-assigns x a float
>>> print(x)
3.14
>>> y = 6                # creates y, assigns int
>>> x * y                # uses x and y
18.84
```

```
x = 17        # Defines and initializes x
y = x + 3     # Defines y and initializes y
z = w         # Runtime error if w undefined
```

This code defines three variables x, y and z. Notice that on the *left hand side* of an assignment the variable is created (if it doesn't already exist), and given a value.

On the *right hand side* of an assignment is an expression. When the assignment statement is run the expression shall be evaluated and the resulting value will be bound to the variable on the left hand side.

Naming Variables

Below are (most of) the rules for naming variables:

- Variable names must begin with a letter or underscore (_) character.
- After that, use any number of letters, underscores, or digits.
- Case matters: "score" is a different variable than "Score."
- You can't use *reserved words*; these have a special meaning to Python and cannot be variable names.

Python Reserved Words.

[Also known as Keywords].

and, as, assert, break, class, continue, ***def****, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, nonlocal, None, not, or, pass, raise, return, True, try, while, with, yield*

IDLE, PyCharm, and other IDEs display reserved words in a different color to help you recognize them.

Not Reserved, but avoid using names of common functions

- A function is a subprogram.
- Python has many built in functions we will use.
- Function names like `print` are *not* reserved words. But using them as variable names is *a very bad idea* because it redefines them.

```
>>> x = 12
>>> print(x)
12
>>> print = 37
>>> print(x)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print(x)
TypeError: 'int' object is not callable
>>>
```

Naming Variables

```
>>> ___ = 10                        # not standard but legal
>>> _123 = 11                       # also not standard
>>> ab_cd = 12                      # fine
>>> ab|c = 13                       # illegal character
  File "<stdin>", line 1
SyntaxError: can't assign to operator
>>> assert = 14                     # assert is reserved
  File "<stdin>", line 1
    assert = 14
           ^
SyntaxError: invalid syntax
>>> max_value = 100                 # good
>>> print = 8                       # legal but ill-advised
>>> print( "abc" )                  # we've redefined print
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

Naming Variables

In addition to the rules, there are also some conventions that programmers follow and we expect you to follow in CS303e:

- Variable names shall begin with a lowercase letter.
- Choose meaningful names that describe how the variable is used. This helps with program readibility.

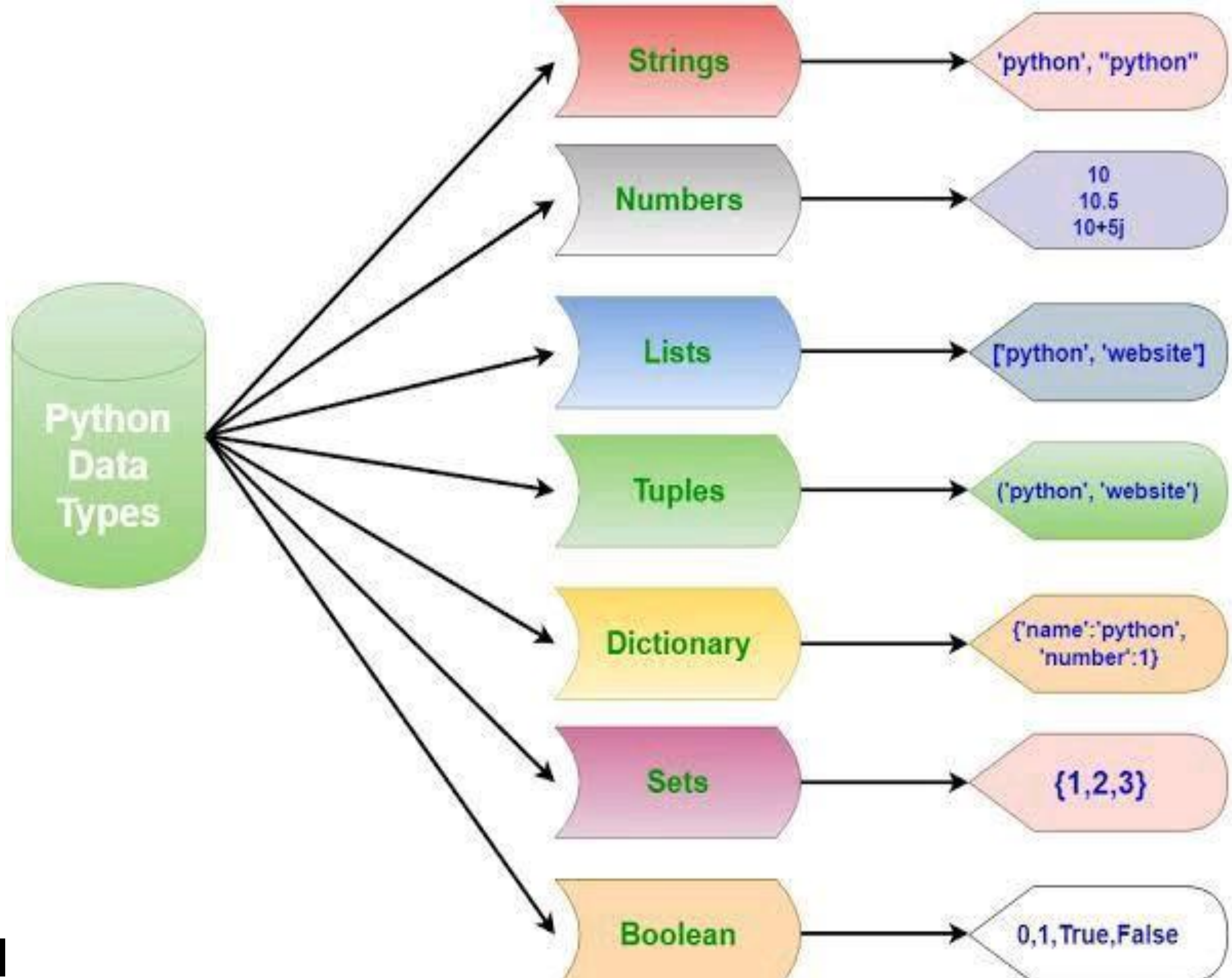
  Use max rather than m.
  Use num_columns rather than c.
- Use underscores to separate multiple words
- loop variables are often i, j, etc.

```
for i in range(1, 20):
    print(i)
```

rather than:

```
for some_value in range(1, 20):
    print(some_value)
```

Common Python Data Types



What is a Data Type?

A **data type** is a categorization of values.

Data Type	Description	Example
int	integer. An immutable number of unlimited magnitude	42
float	A real number. An immutable floating point number, system defined precision	3.1415927
str	string. An immutable sequence of characters	'Wikipedia'
bool	boolean. An immutable truth value	True, False
tuple	Immutable sequence of mixed types.	(4.0, 'UT', True)
list	Mutable sequence of mixed types.	[12, 3, 12, 7, 6]
set	Mutable, unordered collection, no duplicates	{12, 6, 3}
dict	dictionary a.k.a. maps, A mutable group of (key, value pairs)	{'k1': 2.5, 'k2': 5}

Others we likely won't use in 303e:
complex, bytes, frozenset

The `type` Function

```
>>> x = 17
>>> type(x)
<class 'int'>
>>> y = -20.9
>>> type(y)
<class 'float'>
>>> type(w)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'w' is not defined
>>> lst = [1, 2, 3]
>>> type(lst)
<class 'list'>
>>> type(20)
<class 'int'>
>>> type( (2, 2.3 ) )
<class 'tuple'>
>>> type('abc')
<class 'str'>
>>> type( {1, 2, 3} )
<class 'set'>
>>> type(print)
<class 'builtin_function_or_method'>
```

- *Class* is another name for data type.
- Data type is a categorization or classification
- "What kind of thing is the value this variable refers to?"

Three Common Data Types

Three data types we will use in many of our early Python programs are:

`int`: signed integers (whole numbers)

- Computations are exact and *of unlimited size*
- Examples: 4, -17, 0

`float`: signed real numbers (numbers with decimal points) Large

- range, but fixed precision
- Computations are approximate, not exact Examples:
- 3.2, -9.0, 3.5e7

`str`: represents text (a string)

- We use it for input and output We'll see
- more uses later Examples: "Hello, World!",
- 'abc'

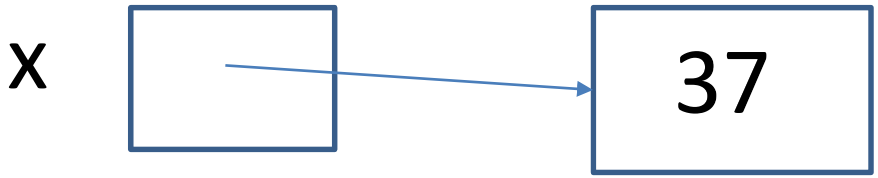
These are all *immutable.* The value cannot be altered.

Immutable

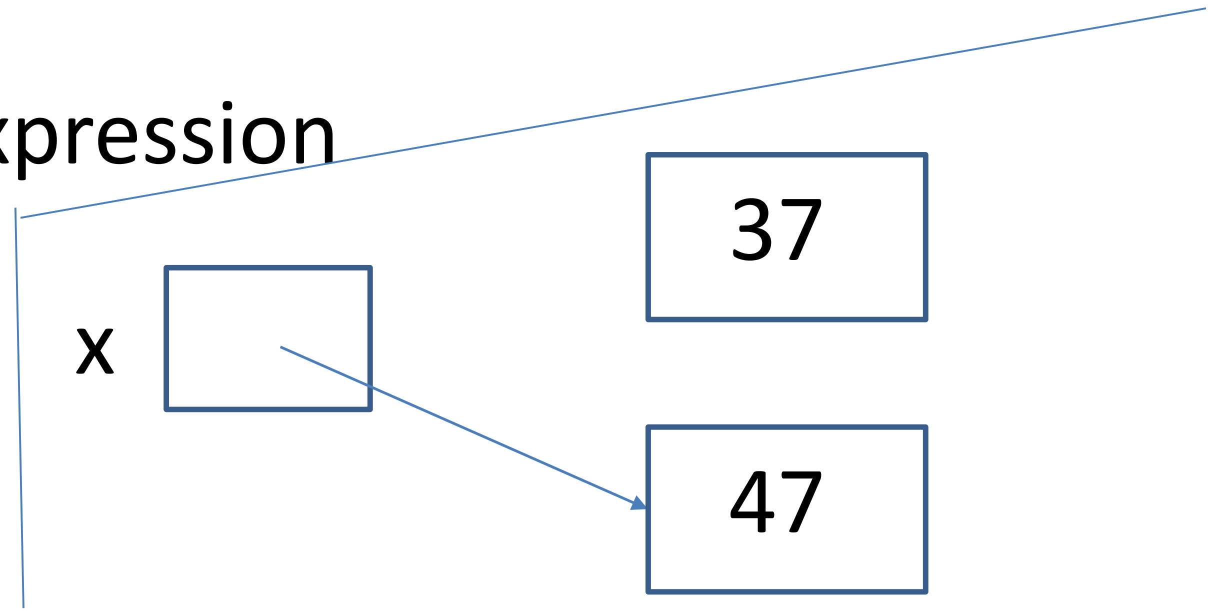
- It may appear some values are mutable
  - they are not
  - rather variables are mutable and can be bound (refer to) different values
- Note, how the id of x (*similar to its address*) has changed

```
>>> x = 37
>>> x
37
>>> id(x)
140711339416352
>>> x = x + 10
>>> x
47
>>> id(x)
140711339416672
```

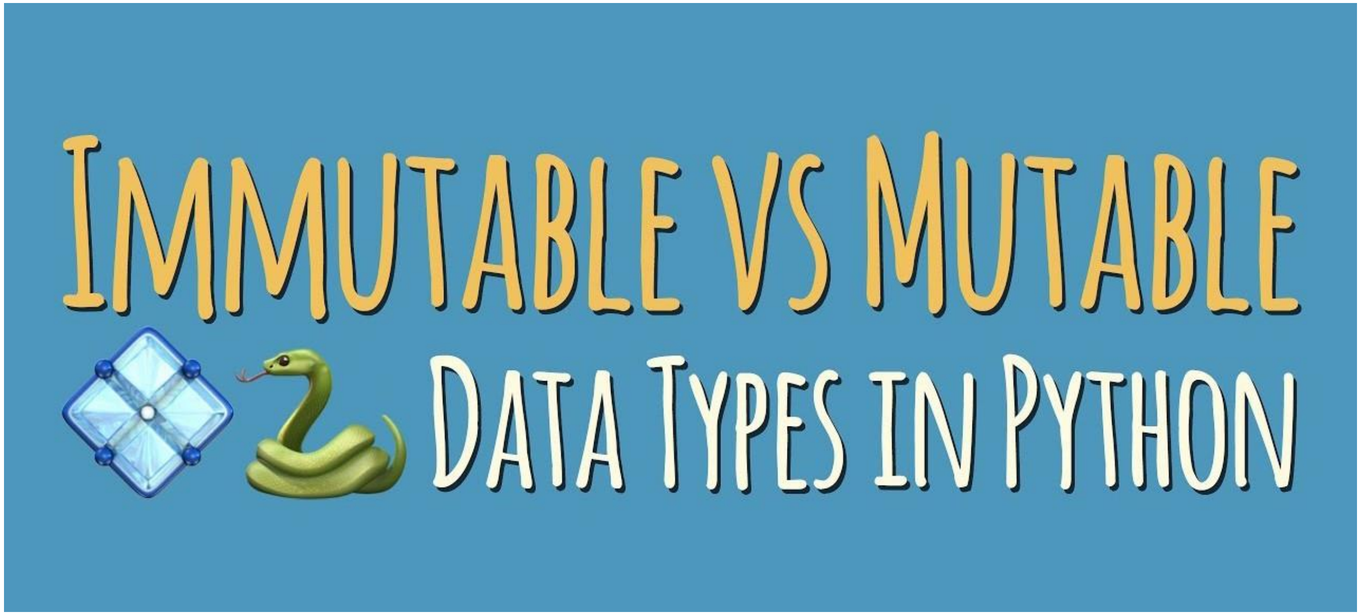
```
x = 37
```




```
x = x + 10
# substitute in the value x is referring to
x = 37 + 10
# evaluate the expression
x = 47
# so now …
```



Mutable vs. Immutable



An **immutable** value is one that cannot be changed by the programmer after you create it; e.g., numbers, strings, etc.

A **mutable** values is one that can be changed; e.g., sets, lists, etc.

What Immutable Means

- An **immutable** object is one that cannot be changed by the programmer after you create it; e.g., numbers, strings, etc.
- It also means that *there is typically only one copy of the object in memory.*
- Whenever the system encounters a new reference to 17, say, it creates a pointer (references) to the already stored value 17.
- Every reference to 17 is actually a pointer to the *only* copy of 17 in memory. Ditto for `"abc"`.
- If you do something to the object that yields a new value (e.g., uppercase a string), you're actually creating a new object, not changing the existing one.

Immutability

```
>>> x = 17              # x holds a pointer to the object 17
>>> y = 17              # so does y
>>> x is y              # x and y point to the same object
True
>>> id(x)               # the unique id associated with 17
10915008
>>> id(y)
10915008
>>> s1 = "abc"          # creates a new string
>>> s2 = "ab" + "c"     # creates a new string (?)
>>> s1 is s2            # actually it doesn't!
True
>>> id(s1)
140197430946704
>>> id(s2)
140197430946704
>>> s3 = s2.upper()        # uppercase s2
>>> print(s3)
ABC
>>> id(s3)              # this is a new string
140197408294088
```

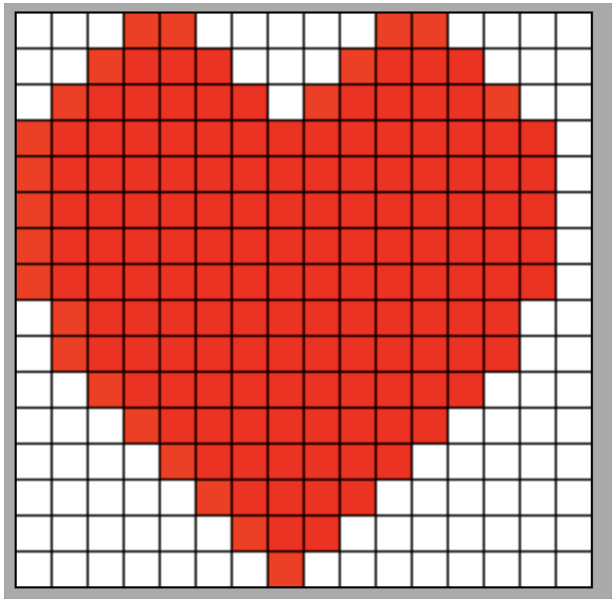
Let's Take a Break

Review from chapter 1

Fundamental fact: *all data* in the computer is stored as a series of bits (0s and 1s) in the memory.

That's true whether you're storing numbers, letters, documents, pictures, movies, sounds, programs, etc. *Everything!*

A key problem in designing any computing system or application is deciding how to *represent* the data we care about as a sequence of bits.



How is Data Stored: Digital Images

For example, images can be stored digitally in any of the following formats (among others):

- JPEG: Joint Photographic Experts Group
- PNG: Portable Network Graphics
- GIF: Graphics Interchange Format
- TIFF: Tagged Image File
- PDF: Portable Document Format
- EPS: Encapsulated Postscript

Most of the time, we won't need to know how data is stored in the memory. The computer will take care of that for us.

Standards?

How is Data Stored?

The memory can be thought of as a big array of **bytes**, where a byte is a sequence of 8 bits. Each memory address has an **address** (0..maximum address) and **contents** (8 bits).

5

Address	Contents	
...		
...		
10000	00110011	Encoding for character '3'
10001	00110000	Encoding for character '0'
10002	00110011	Encoding for character '3'
10003	01000101	Encoding for character 'E'
...		
...		

A byte is the smallest unit of storage a programmer can address. We say that the memory is *byte-addressable*.

Contemporary computer systems may have addressability of 4 or 8 bytes instead of single bytes,

Representation Example: ASCII

The standard way to represent *characters* in memory is ASCII. The following is part of the ASCII (American Standard Code for Information Interchange) representation:

032	sp	048	0	064	@	080	P	096	`	112	p
033	!	049	1	065	A	081	Q	097	a	113	q
034	"	050	2	066	B	082	R	098	b	114	r
035	#	051	3	067	C	083	S	099	c	115	s
036	$	052	4	068	D	084	T	100	d	116	t
037	%	053	5	069	E	085	U	101	e	117	u
038	&	054	6	070	F	086	V	102	f	118	v
039	'	055	7	071	G	087	W	103	g	119	w
040	(	056	8	072	H	088	X	104	h	120	x
041	)	057	9	073	I	089	Y	105	i	121	y
042	*	058	:	074	J	090	Z	106	j	122	z
043	+	059	;	075	K	091	[	107	k	123	{
044	,	060	<	076	L	092	\	108	l	124	|
045	-	061	=	077	M	093	]	109	m	125	}
046	.	062	>	078	N	094	^	110	n	126	~
047	/	063	?	079	O	095	_	111	o	127	⌂

The standard ASCII table defines 128 character codes (from 0 to 127), of which, the first 32 are control codes (non-printable), and the remaining 96 character codes are printing characters.

How is Data Stored

- Characters or small numbers can be stored in one byte. If data can't be stored in a single byte (e.g., a large number), it must be split across a number of adjacent bytes in memory.
- The way data is encoded in bytes varies
  - depending on: the data type
  - the specifics of the computer
- *Most of the time, we won't need to know how data is stored in the memory.* The computer will take care of that for us.

Formats of Data Types

- It would be nice to look at the character string "25" and do arithmetic with it.
- However, the `int` 25 (a number) is represented in binary in the computer by: 00011001. Why?
- And the string "25" (two characters) is represented by: $00110010\ 00110101$. Why?
- `float` numbers are represented in an even more complicated way, since you have to account for an exponent. (Think "scientific notation.") So the number 25.0 (or $2.5 * 10^1$) is represented in yet a third way.

Data Type Conversion - Using Built in Functions

- Python provides functions to *explicitly* convert numbers from one type to another:

  `float` (< number, variable, string >)
  `int` (<number, variable, string >)
  `str` (<number, variable >)

- Note: `int` *truncates*, meaning it throws away the decimal point and anything that comes after it. If you need to *round* to the nearest whole number, use:

  `round` (<number or variable >)

Conversion Examples

```
float(17)
17.0
>>> str(17)
'17'
>>> int(17.75)                          # truncates
17
>>> str(17.75)
'17.75'
>>> int("17")
17
>>> float("17")
17.0
>>> round(17.1)
17
>>> round(17.6)
18
round(17.5)                             # round to even
18
>>> round(18.5)                         # round to even
18
```

Conversion Examples

If you have a string that you want to (try to) interpret as a number, you can use eval.

```
>>> eval("17")
17
>>> eval("17 + 3")
20
>>> eval(17 + 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: eval() arg 1 must be a string,
    bytes or code object
```

What was wrong with the last example?

Be Cautious Using `eval`

- Using the function `eval` is considered dangerous, especially when applied to user input.
- `eval` passes its argument to the Python interpreter, and a malicious (or careless) user could input a command string that could:
  - delete all of your files,
  - take over your machine, or
  - some other horrible thing.
- **Use `int()` or `float()` is you want to convert a string input into one of these types.**

Arithmetic Operations

Here are some useful operations you can perform on numeric data types.

Name	Meaning	Example	Result
+	Addition	34 + 1	35
-	Subtraction	34.0 - 0.1	33.9
*	Multiplication	300 * 30	9000
/	Float division	1 / 2	0.5
//	floor division	1 // 2	0
**	Exponentiation	4 ** 0.5	2.0
%	Remainder	20 % 3	2

(x % y) is often referred to as "x mod y"

Integer Division

- Floor Division specified with the // operator
- … goes to the *floor* on a number line
- Discards the remainder from the division operation.

```
>>> 37 // 10
3
>>> 17 // 20
0
>>> 2.5 // 2.0
1.0
>>> -22 // 7
-4
>>> -22 // -7
3
```

Modulo Operator

- % is the Modulo operator
- x % y evaluates to the remainder of x // y
- "The floor division and modulo operators are connected by the following identity:"

```
>>> 37 % 10
7
>>> 17 % 20
17
>>> -22 % 7
6
>>> -22 % -7
-1
```

```
x == (x // y) * y + (x % y)
```

- **B**ody **M**ass **I**ndex or **BMI** is a quick calculation based on height and mass (weight) used by medical professionals to broadly categorize people .
- Formula:

$$\boxed{\mathrm{BMI} = \frac{\mathrm{mass}_{\mathrm{kg}}}{\mathrm{height}_{\mathrm{m}}^2} = \frac{\mathrm{mass}_{\mathrm{lb}}}{\mathrm{height}_{\mathrm{in}}^2} \times 703}$$

- Quick tool to get a rough estimate if someone is underweight, normal weight, overweight, or obese
- Write an interactive program that gets the name, height, and weight of a user and calculates BMI.

Simple Input

- Obtain input from the user by calling a built in Python function named **`input`**.
- Just like we can send information (arguments) to **`print`**, we can send information (again, arguments) to **`input.`**
  - The argument is a prompt that will be displayed.
- Trying reading a height and weight from the user and calculating BMI.
- What happens?

- More built in functions to convert from String data type to int or float data type. **`int(), float()`**

Simple Program: Pythagorean Triples

In file pythagoreanTriple.py:

```
""" The sides of a right triangle satisfy the relation:
       a**2 + b**2 = c**2.
    Test whether the three integers in variables a, b, c
    form a pythagorean triple, i.e., satisfy this relation.
       """


a = 3
b = 4
c = 5
ans = ( a**2 + b**2 == c**2 )
print("a:", a, "b:", b, "c:", c, \
      "is" if ans else "is not", \
      "a pythagorean triple" )
```

```
> python pythagoreanTriple.py
a: 3 b: 4 c: 5 is a pythagorean triple
```

Note, print can take multiple values.

Default separator is a space,

default end is a newline

Augmented Assignment Operators

Python (like C, Java, C++...) provides a shorthand syntax for some common assignments:

`i += j`	functionally the same as	`i = i + j`
`i -= j`	functionally the same as	`i = i - j`
`i *= j`	functionally the same as	`i = i * j`
`i /= j`	functionally the same as	`i = i / j`
`i //= j`	functionally the same as	`i = i // j`
`i %= j`	functionally the same as	`i = i % j`
`i **= j`	functionally the same as	`i = i ** j`

```
>>> x = 2.4
>>> x *= 3.7   # functionally same as x = x * 3.7
>>> print(x)
8.88
```

Mixed-Type Expressions

Most arithmetic operations behave as you would expect for numeric data types.

- Combining two floats results in a float.
- Combining two ints results in an int (except for /). Use // for integer division.
- Dividing two ints gives a float. E.g., `2 / 5` yields 2.5.
- Combining a float with an int usually yields a float.

Python will figure out what the result will be and return a value of the appropriate data type.

Mixed Type Expressions

```
>>> 5 * 3 - 4 * 6                   # (5 * 3) - (4 * 6)
-9
>>> 4.2 * 3 - 1.2
11.400000000000002                  # approximate result
>>> 5 // 2 + 4                      # integer division
6
>>> 5 / 2 + 4                       # float division
6.5
```

Special Assignment Statements

Simultaneous assignments:

```
m, n = 2, 3
```

means the same as:

```
m = 2
n = 3
```

With the caveat that these happen *at the same time*.

What does the following do?

```
i, j = j, i
```

Multiple assignments:

```
i = j = k = 1
```

means the same as:

```
k = 1
j = k
i = j
```

Note that these happen right to left.

Advice on Programming

Think before you code!
Think before you code!
Think before you code!

- Don't jump right into writing code.
- Think about the overall process of solving your problem; write it down.
- Refine each part into subtasks. Subtasks may require further refinement.
- Code and test each subtask before you proceed.
- Add `print` statements to view intermediate results.

Advice on Programming

Software development is typically done via an iterative process. You'll do well to follow it, except on the simplest programs.