Python Basics

Topics

Based on `examples/Python/1-Basics.ipynb`

1. Semantics: constants, variables, literals, expressions, types, input

Constants, Variables, Literals

- **Constants** are fixed (numbers, letters, strings) values that do not change
  - Convention to name them in all capital letters
  - This does not actually prevent reassignment (difference wrt C)

- In `>>> x = 7.16` `x` is a **variable** and `7.16` is a **literal**
  - A literal is raw data given to a variable or constant
  - Numeric literal types: `Integer`, `Float` or `Complex`
  - String literals (surrounded by single, double or triple quotes)
  - Boolean literals (`True` or `False`)
  - Special literal (`None`): specifies that the field has not been created

Rules for Variable Names

- **Reserved words** cannot be used as variable names/identifiers
- Must start with a letter or underscore
- Must consist of letters, numbers, and underscores
- Case sensitive

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

Numeric Expressions

OPERATOR	OPERATION
+	Addition
–	Subtraction
*	Multiplication
/	Division
**	Power
%	Remainder

Order precedence rules (highest to lowest)

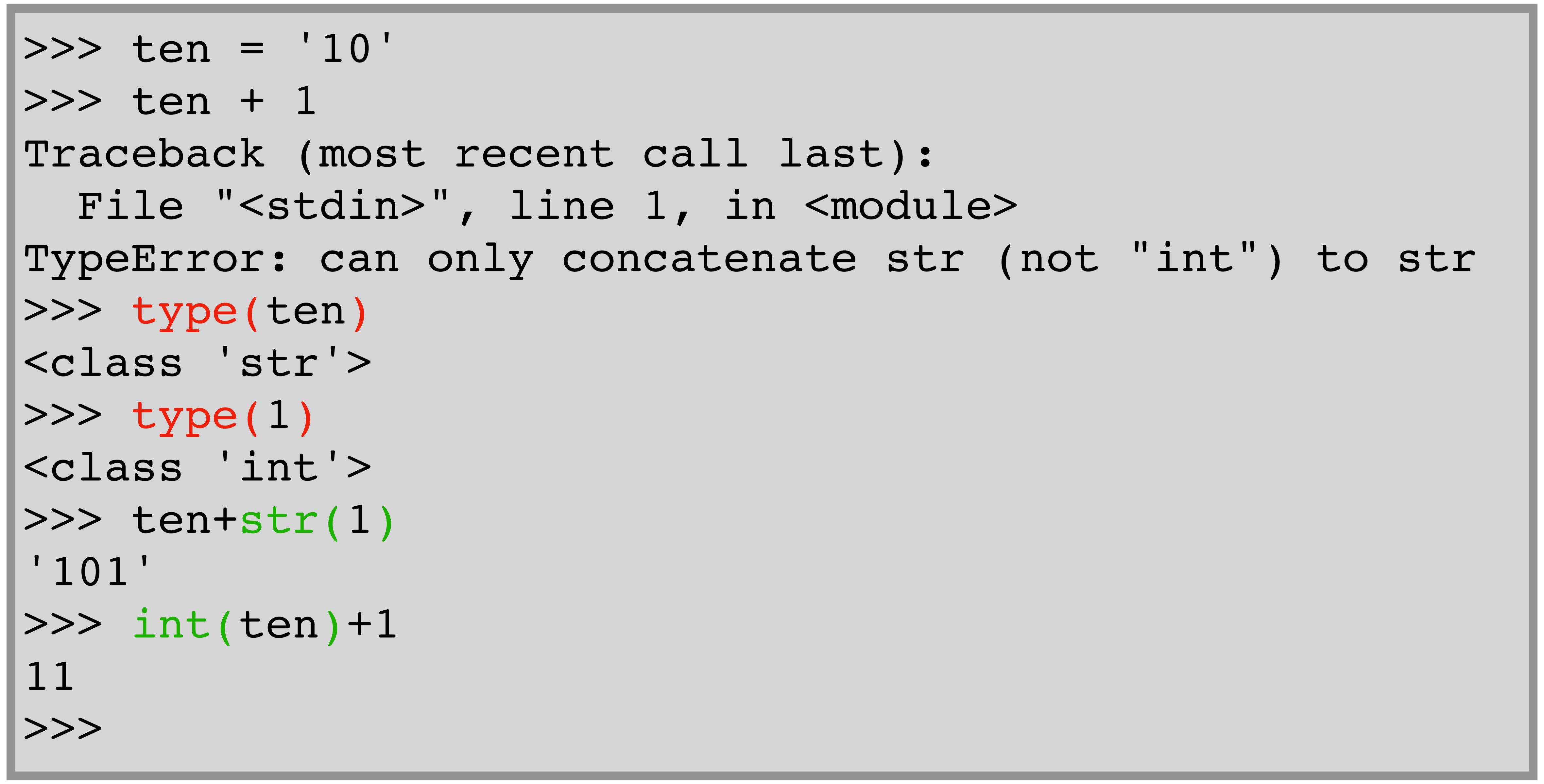
1. Parentheses are always respected
2. Exponentiation (raise to a power)
3. Multiplication, division, and remainder
4. Addition and subtraction
5. Left to right

```
>>> x = 1 + 2 ** 3 / 4 * 5
>>> print(x)
11.0
>>>
```

Types

In Python variables, literals, and constants have a `Type`

- Python knows the difference between an integer and a string (without you having to declare variables) and behaves accordingly
- E.g., + means “addition” for numbers and “concatenate” for strings



```
>>> ten = '10'
>>> ten + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>> type(ten)
<class 'str'>
>>> type(1)
<class 'int'>
>>> ten+str(1)
'101'
>>> int(ten)+1
11
>>>
```

- ‣ Type matters
- ‣ Use `str()`, `int()`, `float()`

What does `float("ten")` do?

User Input

- The `input()` function instructs Python to pause and read data from the user
- `input()` returns a string
- To use as a number a number read in from the user, we must use a type conversion function

```
>>> a = input()
12
>>> type(a)
<class 'str'>
```

✓ Ready for part 1 of `examples/Python/1-Basics.ipynb`

Topics

Based on `examples/Python/1-Basics.ipynb`

1. Semantics: constants, variables, literals, expressions, types, input

2. Flow control: conditional execution and loops

Indentation in Flow Control

- Logical structure is achieved with **indentation** (typically 4 spaces) and **colons** (`:`)
    - Increase indent after an `if` or `for` statement (following the `:`)
    - Maintain indent to indicate the scope of the block
    - Reduce indent back to the level of the `if` or `for` statement to mark the end of the block

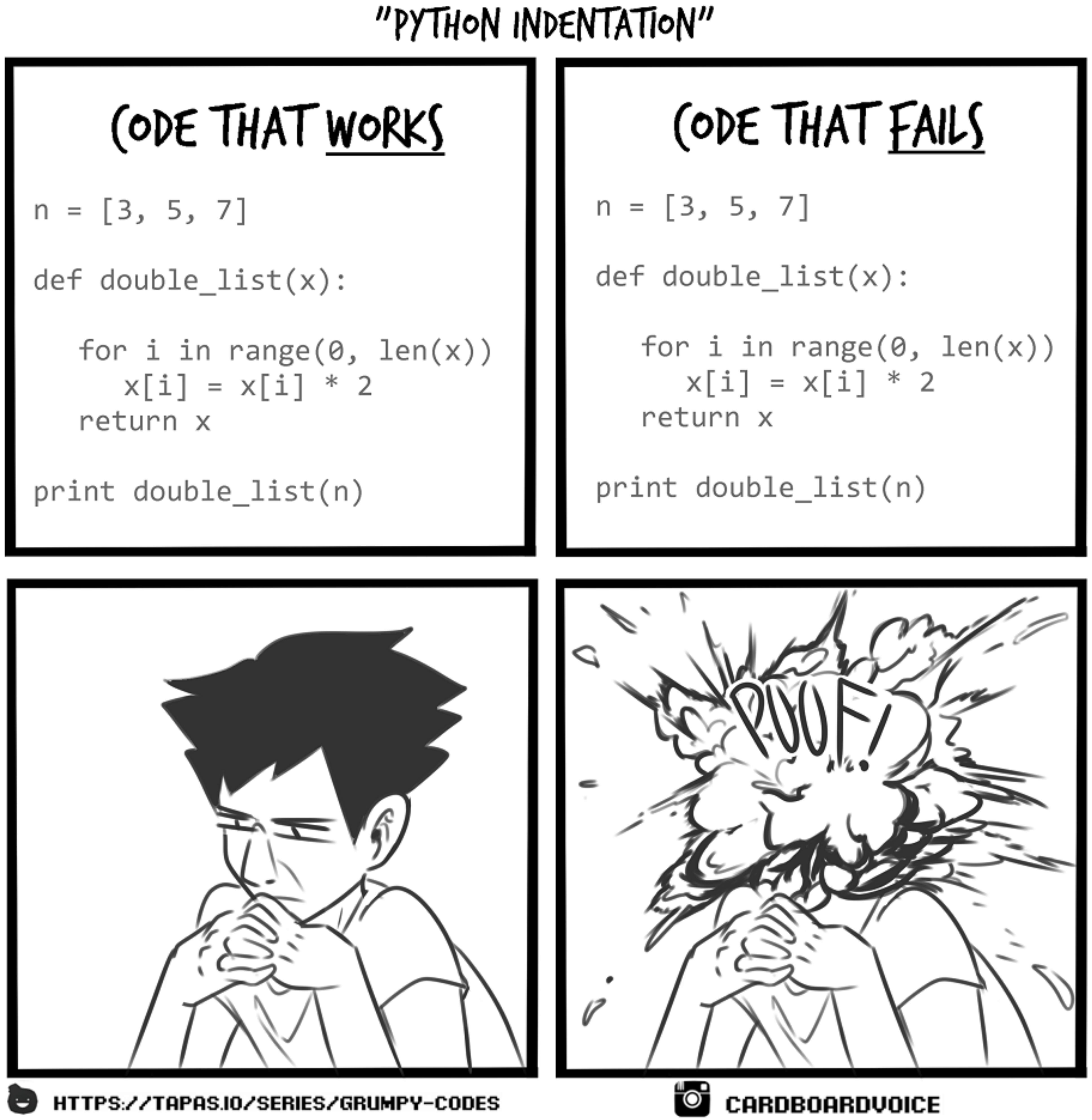
- With regards to indentation, the following are ignored
    - Blank lines
    - Comments on a line by themselves

- Python cares a lot about how far a line is indented
    - Mixing tabs and spaces can lead to “indentation errors”
    - **Stay away from tabs!**

```
x = 5
if x > 2:
    print('Bigger than 2')
    print('Still bigger')
print('Done with 2')

for i in range(5):
    print(i)
    if i > 2:
        print('Bigger than 2')
    print('Done with i', i)
print('All Done')
```

Increase
Maintain
Decrease

From C++ to Python


"PYTHON INDENTATION"

CODE THAT WORKS

```
n = [3, 5, 7]

def double_list(x):

  for i in range(0, len(x))
    x[i] = x[i] * 2
  return x

print double_list(n)
```

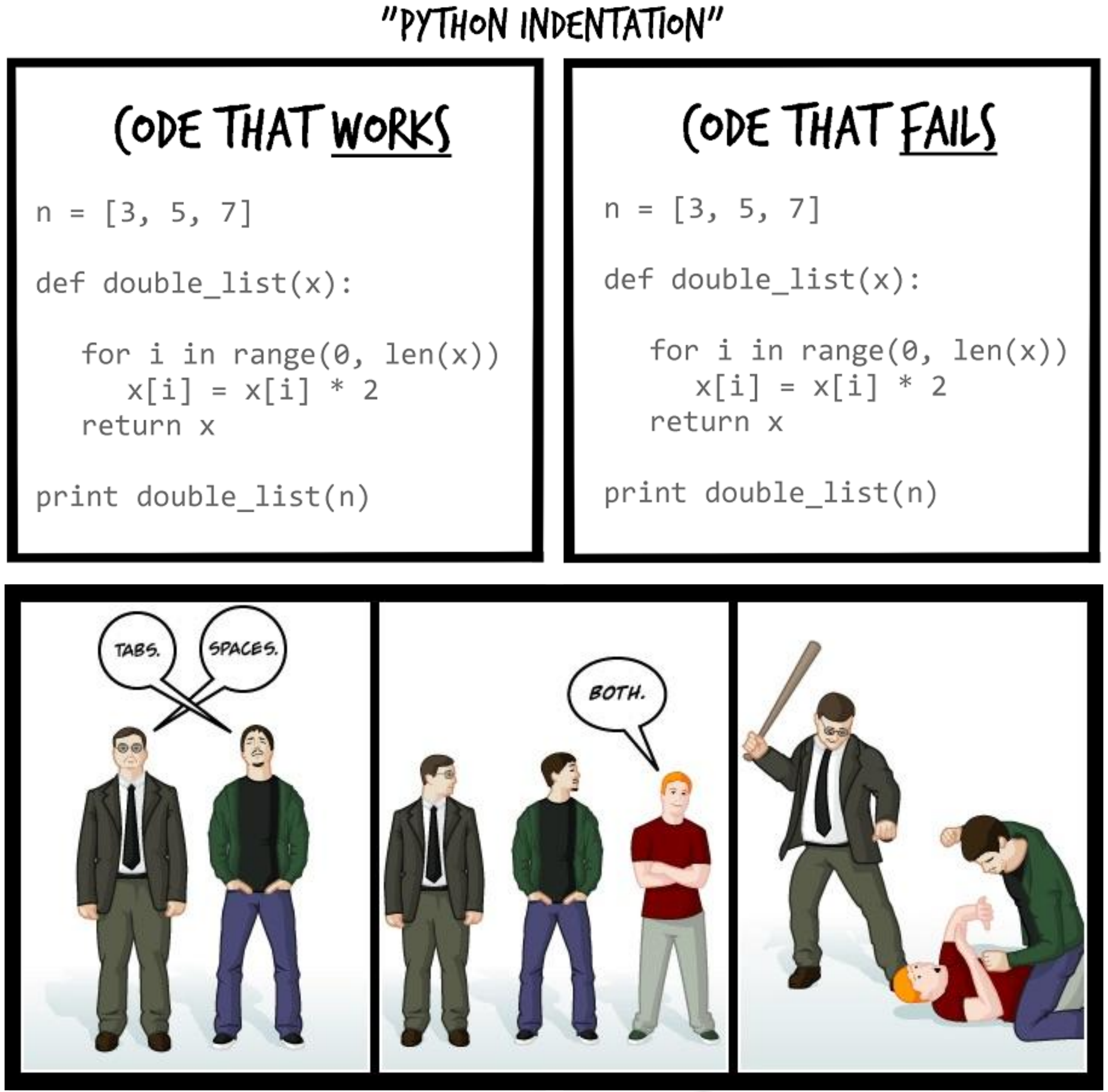
CODE THAT FAILS

```
n = [3, 5, 7]

def double_list(x):

  for i in range(0, len(x))
    x[i] = x[i] * 2
  return x

print double_list(n)
```

HTTPS://TAPAS.IO/SERIES/GRUMPY-CODES

CARDBOARDVOICE

From C++ to Python



CODE THAT WORKS

```
n = [3, 5, 7]

def double_list(x):

   for i in range(0, len(x))
      x[i] = x[i] * 2
   return x

print double_list(n)
```

CODE THAT FAILS

```
n = [3, 5, 7]

def double_list(x):

   for i in range(0, len(x))
      x[i] = x[i] * 2
   return x

print double_list(n)
```

HTTPS://TAPAS.IO/SERIES/GRUMPY-CODES

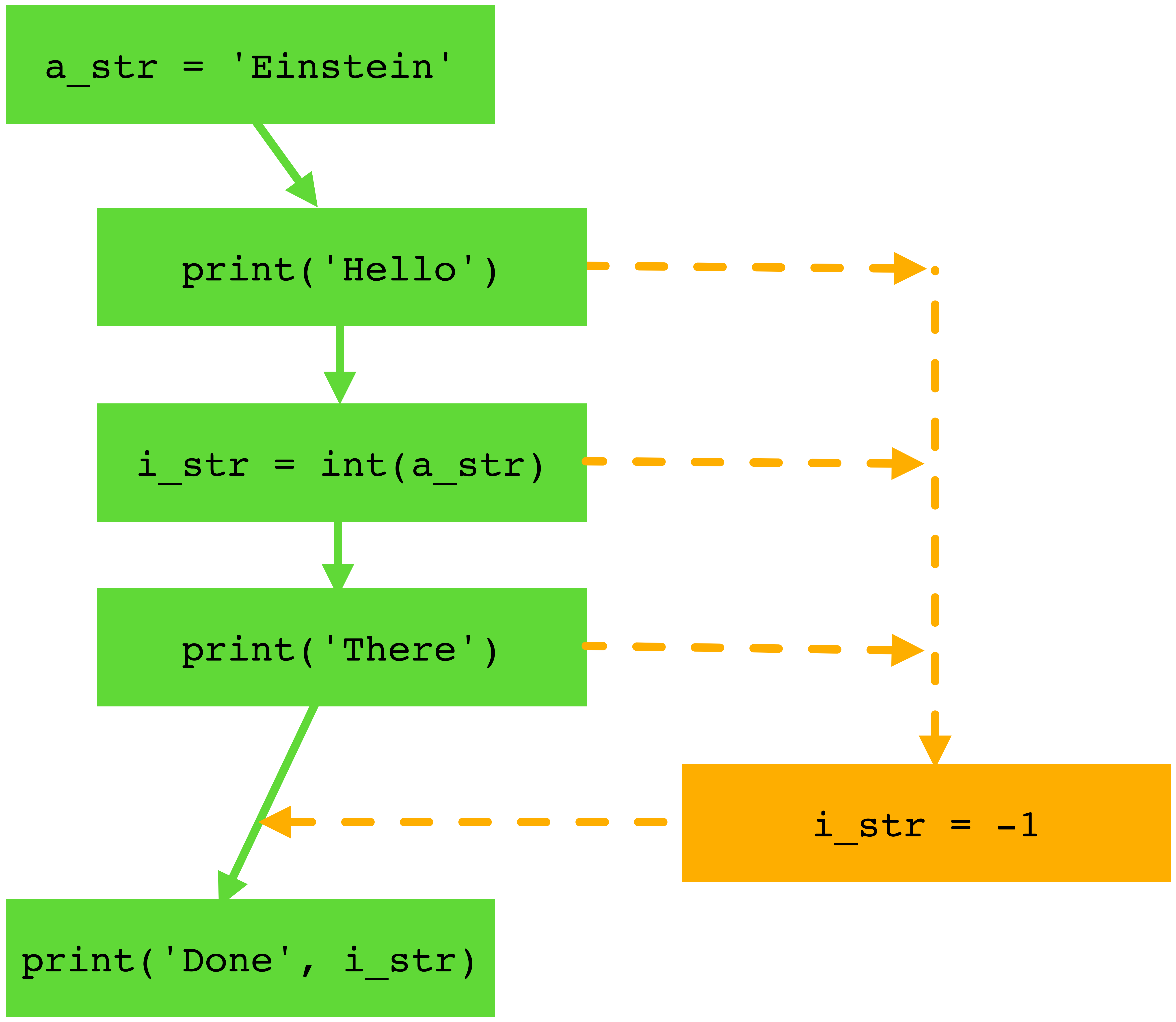
CARDBOARDVOICE

Flow Control Tools

- `while` loop: basically the same as C
- `for` loop: requires a sequence of objects to iterate on [e.g., `range(0,10,1)`]
- `if/elif` conditional: `elif` stands for “else if” and does not exist in C
- `try/except`: is also new, it provides a safety net
    - Surround a dangerous section of code with `try` and `except`
    - If the code in the `try` works, the `except` is skipped
    - If the code in the `try` fails, it jumps to the `except` section

try/except

```
a_str = 'Einstein'
try:
    print('Hello')
    i_str = int(a_str)
    print('There')
except:
    i_str = -1

print('Done', i_str)
```



a_str = 'Einstein'

print('Hello')

i_str = int(a_str)

print('There')

i_str = -1

print('Done', i_str)

try/except/else/finally

- The else clause can be used with if/elif conditional statements (as in C) and with:
  - for loops – it is executed when the loop terminates
  - while loops – it is executed when the condition becomes false
  - try/except try statements – it is executed with the following logic

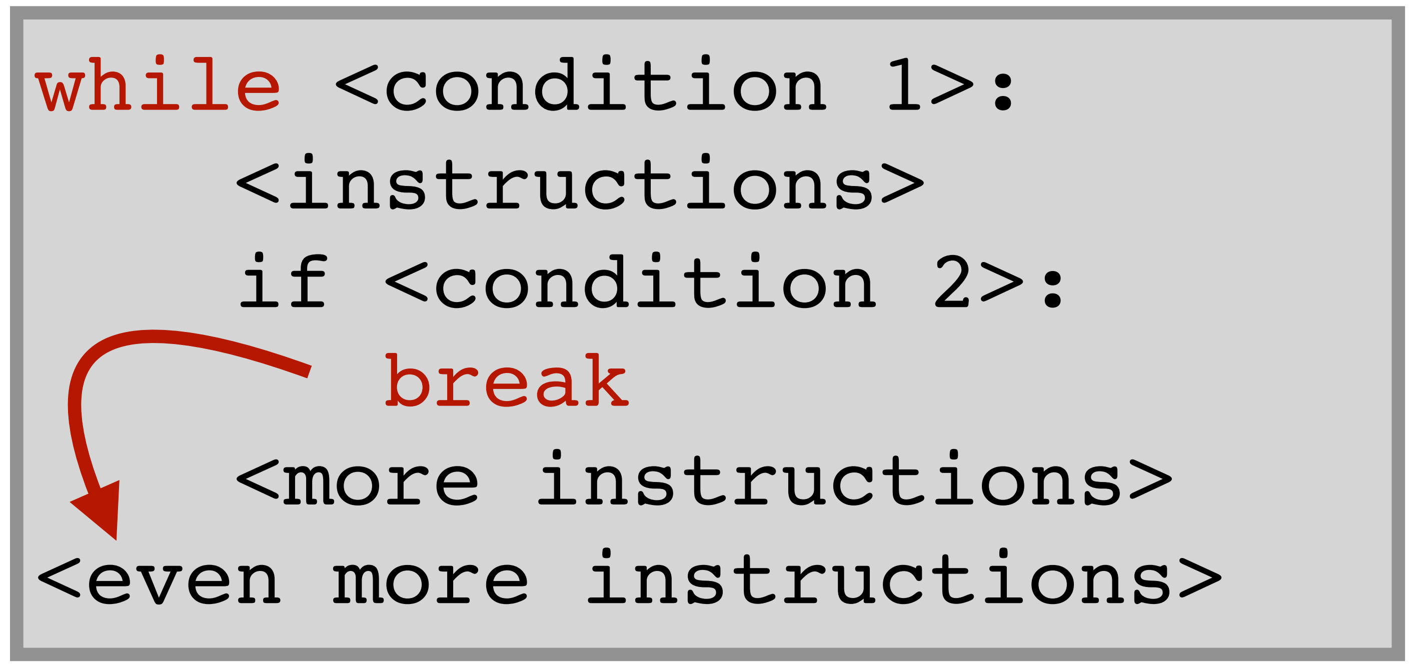
```
try:
    # Some code
except:
    # Optional block
    # Handling of exception (if required)
else:
    # Execute if no exception
finally:
    # Some code (always executed)
```

`break`, `continue`, and `pass`

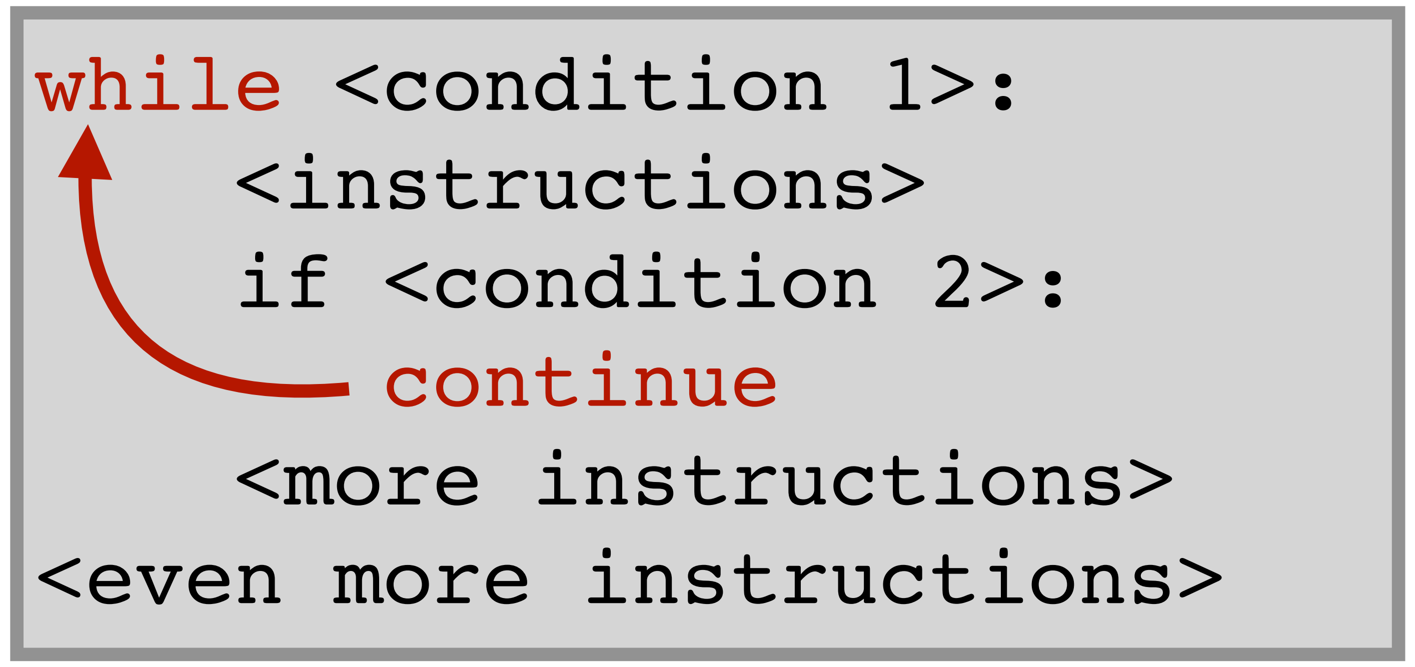
- Like in C, the `break` statement takes you out of the innermost enclosing `for` or `while` loop

- Like in C, the `continue` statement continues with the next loop iteration

- The `pass` statement does nothing: it is used when a statement is required syntactically but the program requires no action



```
while <condition 1>:
    <instructions>
    if <condition 2>:
       break
    <more instructions>
<even more instructions>
```



```
while <condition 1>:
    <instructions>
    if <condition 2>:
       continue
    <more instructions>
<even more instructions>
```

✓ Ready for part 2 of `examples/Python/1-Basics.ipynb`

Topics

Based on `examples/Python/1-Basics.ipynb`

1. Semantics: constants, variables, literals, expressions, types, input

2. Flow control: conditional execution and loops

3. Functions and modules

Functions

- The keyword `def` is used to create a new function
- It is followed by the name of the function which becomes a reserved word
- In parenthesis we list the parameters of the function
  - A parameter is a “handle” that allows the code in the function to access the arguments for a particular function invocation
- The body of the function needs to be indented
- This defines the function, without executing its body
- The `return` statement ends the function definition and “sends back” the result
  - A void function is one that does not return a value (a `None` is returned automatically)
    - In this case, the end of the indentation ends the function definition

✓ Ready for part 3 of `examples/Python/1-Basics.ipynb`

Topics

Based on `examples/Python/1-Basics.ipynb`

1. Semantics: constants, variables, literals, expressions, types, input

2. Flow control: conditional execution and loops

3. Functions and modules

4. Collections: tuples, lists, dictionaries, and sets; comprehensions

What is a Collection?

- Allows you to put more than one value in it and carry them all around in one convenient package (think of arrays and `std::vector` class in C++)
  - We have a bunch of values in a single “variable”
  - We do this by having more than one place “in” the variable
  - We have ways of finding the different places in the variable

- Python collections can be
  - mutable/immutable
  - ordered/unordered

1. tuples
2. lists
3. strings

1. dictionaries
2. sets

Tuples

- Tuples are sequences: they have elements which are indexed starting at 0
  - They can store any object, and objects of different types (or be empty)

- They are **immutable**: once you create a tuple, you cannot alter its contents
  - This makes them similar to strings
  - You **cannot** `sort()`, `append()`, `reverse()` a tuple

```
>>> x = (9, 8, 7)
>>> x[1] = 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not
support item assignment
>>>
```

```
>>> y = 'ABC'
>>> y[2] = 'D'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not
support item assignment
>>>
```

Lists

- In short, lists are like tuples that can be modified: they are **mutable**
  - Tuples are more efficient (because they are less flexible)
- List **slicing**
  - `a[start:stop:step]`
  - `start` through not past `stop`, by `step`

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[0:6:2]
['a', 'c', 'e']
```

- Concatenating is achieved with the + operator
- `in` and `not in` operators let you check if an item is in a list
  - These are logical operators that return `True` or `False`

Dictionaries

- Python’s most powerful data collection
- **Mutable**
- **Unordered**: index the dictionary **items** with “lookup tags” known as **keys**
  - Similar to lists but these use numbers
- This is like `map` & `vector` in C++, but with lighter syntax

```
>>> purse = {'money':12, 'candy':3}
>>> purse['keys'] = ['office', 'house']
>>> print(purse)
{'money': 12, 'candy': 3, 'keys': ['office', 'house']}
>>> print(purse['candy'])
3
>>> purse['candy'] = purse['candy'] + 2
>>> print(purse)
{'money': 12, 'candy': 5, 'keys': ['office', ‘house']}
```

Sets

- A set is an **unordered** and **mutable** collection with no duplicate elements
  - Basic uses include membership testing and eliminating duplicate entries
  - Sets support mathematical operations such as union, intersection, difference, and symmetric difference

```
>>> x = {9, 8, 7, 7, 8, 9}
>>> print(x)
{8, 9, 7}
>>> x[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

✓ Ready for part 4 of `examples/Python/1-Basics.ipynb`

Topics

Based on `examples/Python/1-Basics.ipynb`

1. Semantics: constants, variables, literals, expressions, types, input

2. Flow control: conditional execution and loops

3. Functions and modules

4. Collections: tuples, lists, dictionaries, and sets; comprehensions

5. File processing: simple text I/O, `pickle`, `json`

Files

- A text file can be thought of as a sequence of lines
- Before we can read the contents of the file, use the `open()` function to specify the file to work with and what will be done with it
  - This returns a “file handle,” a variable used to perform operations on the file

```
handle = open(filename, mode)
```

  - `mode` is optional: `'r'` for reading the file and `'w'` for writing to the file

pickle

- Implements binary protocols for (de-)serializing a Python object structure
  - "**Pickling**:" converting a Python object hierarchy into a byte stream
  - "**Unpickling**:" the inverse operation
- `import pickle` or `import cpickle` : the latter is written in C and is faster

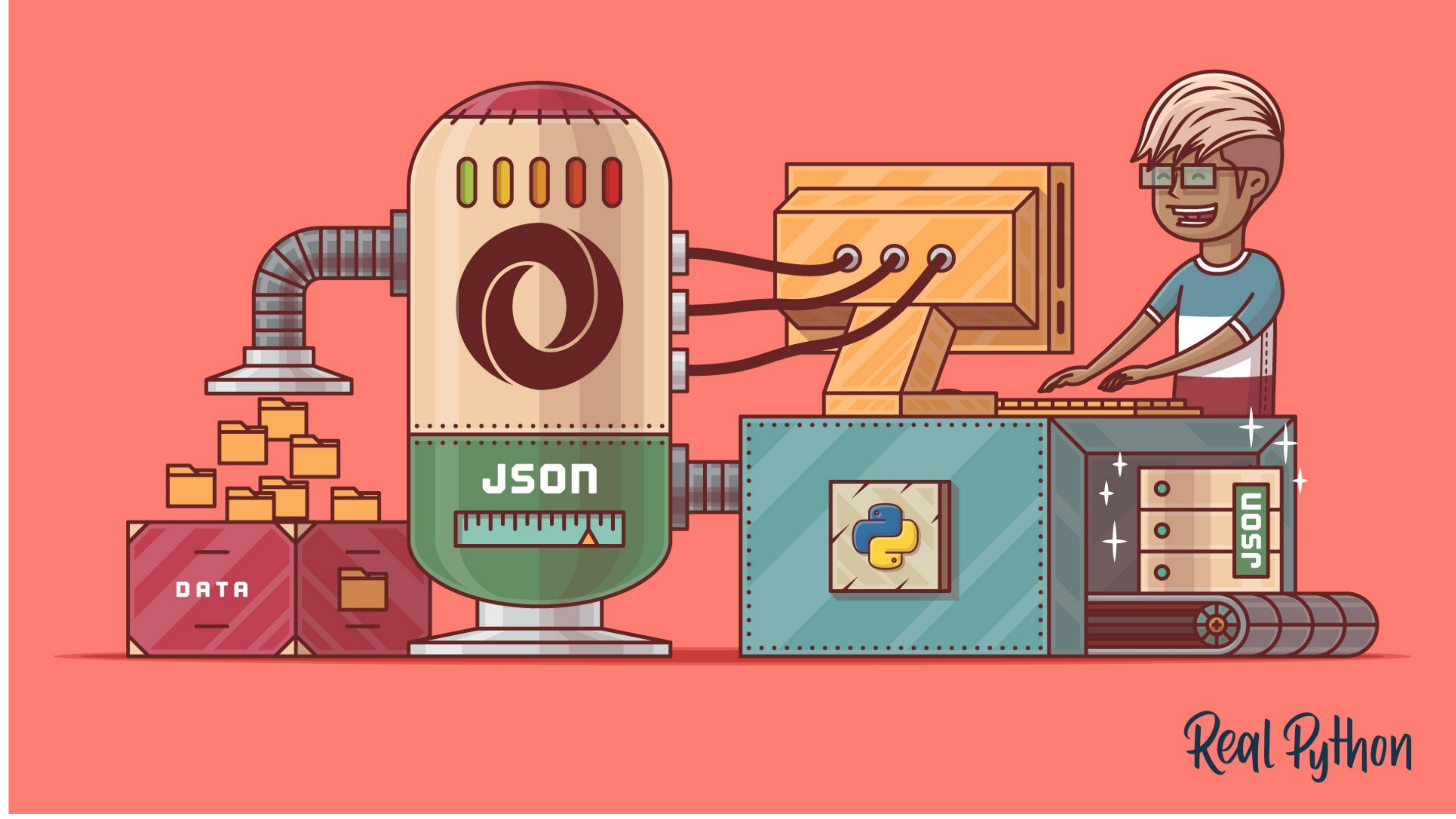
```
>>> import pickle
>>> a = ['John', 'Paul', 'George', 'Ringo']
>>> fileHandle = open("theBeatles", 'wb')
>>> pickle.dump(a, fileHandle)
>>> fileHandle.close()
```

b is for binary

```
>>> import pickle
>>> fileHandle = open("theBeatles",'rb')
>>> b = pickle.load(fileHandle)
>>> print(b)
>>> ['John', 'Paul', 'George', 'Ringo']
>>> fileHandle.close()
```

b is for binary

`json`



- JavaScript Object Notation (JSON) is another common storage format
    - Cross platform and cross language: in Python it is handled by the `json` module
- **Serialization** is the equivalent of pickling
    - `dump()`: convert an object into JSON and possibly write to file
    - `dumps()`: convert to JSON string but cannot interact with file
- **Deserialization** is the equivalent of unpickling
    - `load()`

✓ Ready for part 5 of `examples/Python/1-Basics.ipynb`