python™

Roberto Rossi
University of Edinburgh

Preliminaries

https://goo.gl/xBo9Ju

You should watch these videos.

QR code not marked as above are additional references;
feel free to skip them

Introduction

https://goo.gl/xBo9Ju

Topics

- Computer Programming
- Algorithms
- Integrated Development Environment (IDE)
- Python
- Object-Oriented (OO) Programming
- Test-driven Development
- Assignments

Topics

- Computer Programming
- Algorithms
- Integrated Development Environment (IDE)
- Python
- Object-Oriented (OO) Programming
- Test-driven Development
- Assignments

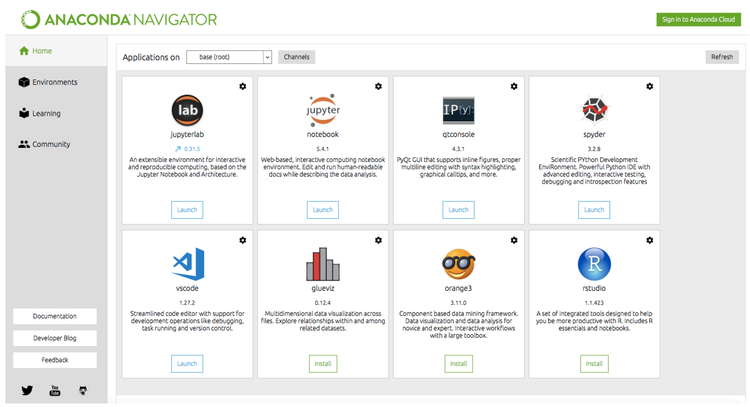
Topics

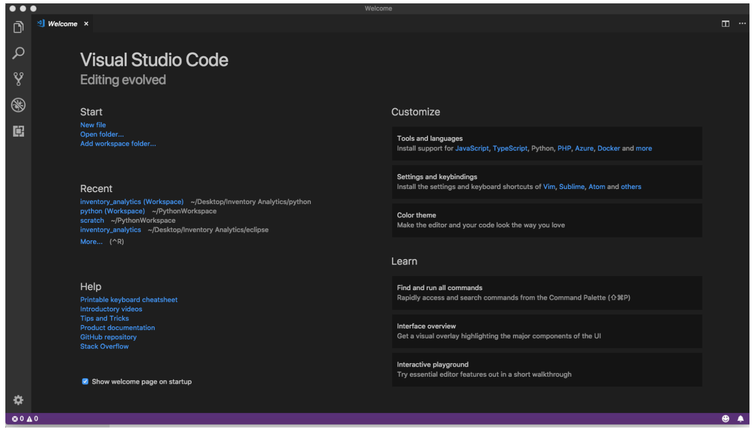
- Computer Programming
- Algorithms
- Integrated Development Environment (IDE)
- Python
- Object-Oriented (OO) Programming
- Test-driven Development
- Assignments

Algebra. from Arabic al-jabr ‘the reunion of broken parts’, ‘bone-setting’, from jabara ‘reunite, restore’. The original sense, ‘the surgical treatment of fractures’, probably came via Spanish, in which it survives; the mathematical sense comes from the title of a book, ‘ilm al-jabr wa'l-muqābala ‘*the science of restoring what is missing and equating like with like*’, by the mathematician al-Ḵwārizmī

Topics

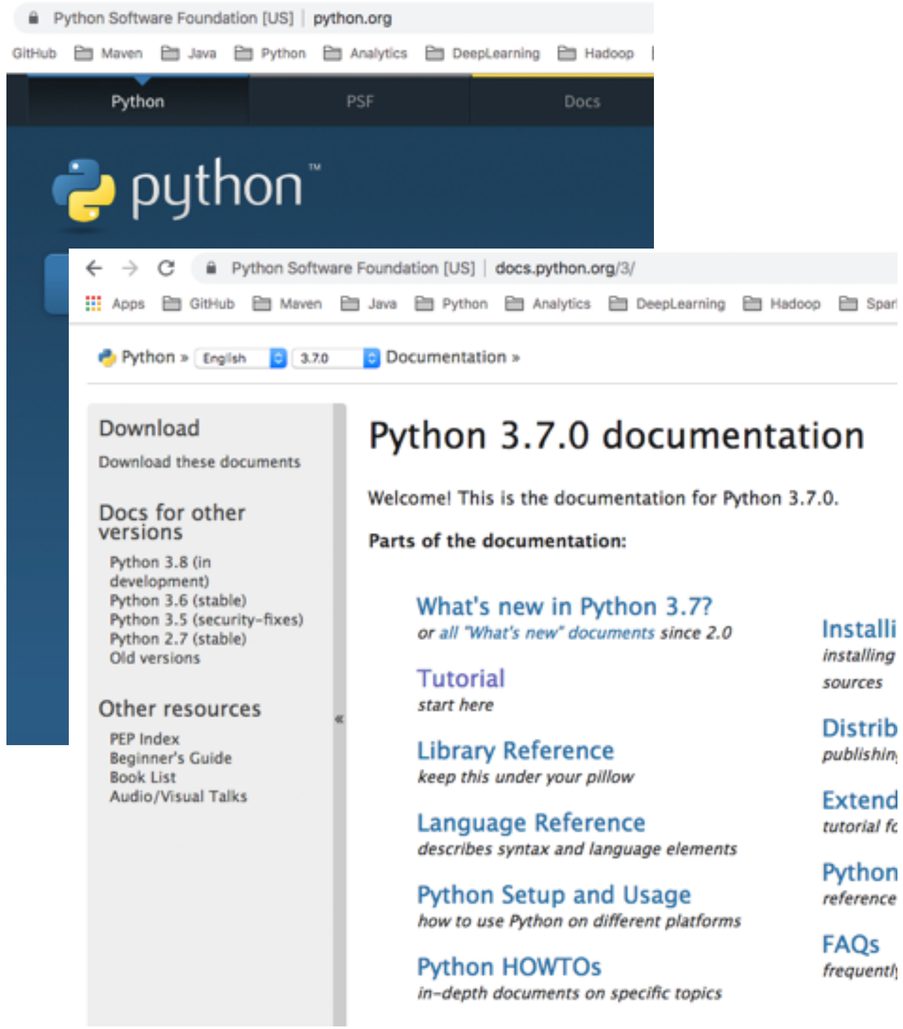
- Computer Programming
- Algorithms
- Integrated Development Environment (IDE)
- Python
- Object-Oriented (OO) Programming
- Test-driven Development
- Assignments





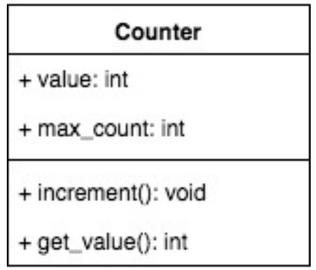
Topics

- Computer Programming
- Algorithms
- Integrated Development Environment (IDE)
- Python
- Object-Oriented (OO) Programming
- Test-driven Development
- Assignments



Topics

- Computer Programming
- Algorithms
- Integrated Development Environment (IDE)
- Python
- Object-Oriented (OO) Programming
- Test-driven Development
- Assignments




```python
class Counter:
    max_count = 500

    def __init__(self):
        self.value = 0

    def increment(self):
        self.value = \
        self.value + 1

    def get_value(self):
        return self.value
```

Topics

- Computer Programming
- Algorithms
- Integrated Development Environment (IDE)
- Python
- Object-Oriented (OO) Programming
- Test-driven Development
- Assignments

```
import unittest
import counter_package.counter_module as cm

Run Test | Debug Test
class TestCounter(unittest.TestCase):

    def setUp(self):
        self.c = cm.Counter()

    def tearDown(self):
        pass

    ✔ Run Test | ✔ Debug Test
    def testCounter(self):
        self.c.increment()
        self.assertEqual(self.c.get_value(), 1)
```

Topics

- Computer Programming
- Algorithms
- Integrated Development Environment (IDE)
- Python
- Object-Oriented (OO) Programming
- Test-driven Development
- Assignments

Assignments

- Euclid's Greatest Common Division (GCD) algorithm
- Bubble Sort
- Eratostene's Sieve
- Treasure Hunt
- Book Catalogue

Computer Programming

https://goo.gl/m5YRiR

“Programming is language…”

–Sarah Mei

“… and there are zillions of languages out there!”

–me

Programming Paradigms

Programming paradigms

- Action
- Agent-oriented
- Array-oriented
- Automata-based
- Concurrent computing
  - Relativistic programming
- Data-driven
- Declarative (contrast: Imperative)
  - Functional
    - Functional logic
    - Purely functional
  - Logic
    - Abductive logic
    - Answer set
    - Concurrent logic
    - Functional logic
    - Inductive logic
  - Constraint
    - Constraint logic
      - Concurrent constraint logic
  - Dataflow
    - Flow-based
    - Cell-oriented (spreadsheets)
    - Reactive
- Dynamic/scripting
- Event-driven
  - Service-oriented
  - Time-driven
- Function-level (contrast: Value-level)
  - Point-free style
    - Concatenative
- Generic
- Imperative (contrast: Declarative)
  - Procedural
  - Object-oriented
- Literate
- Language-oriented
  - Natural-language programming
  - Discipline-specific
  - Domain-specific
  - Grammar-oriented
  - Intentional
- Metaprogramming
  - Automatic
    - Inductive programming
  - Reflective
    - Attribute-oriented
  - Homoiconic
  - Macro
  - Template
- Non-structured (contrast: Structured)
  - Array
- Nondeterministic
- Parallel computing
  - Process-oriented
- Probabilistic
- Stack-based
- Structured (contrast: Non-structured)
  - Block-structured
  - Modular (contrast: Monolithic)
  - Object-oriented
    - Actor-based
    - Class-based
    - Concurrent
    - Prototype-based
    - By separation of concerns:
      - Aspect-oriented
      - Role-oriented
      - Subject-oriented
  - Recursive
- Symbolic
- Value-level (contrast: Function-level)
- Quantum programming

Programming Paradigms

Programming paradigms

- Action
- Agent-oriented
- Array-oriented
- Automata-based
- Concurrent computing
  - Relativistic programming
- Data-driven
- Declarative (contrast: Imperative)
  - Functional
    - Functional logic
    - Purely functional
  - Logic
    - Abductive logic
    - Answer set
    - Concurrent logic
    - Functional logic
    - Inductive logic
  - Constraint
    - Constraint logic
      - Concurrent constraint logic
  - Dataflow
    - Flow-based
    - Cell-oriented (spreadsheets)
    - Reactive
- Dynamic/scripting
- Event-driven
  - Service-oriented
  - Time-driven
- Function-level (contrast: Value-level)
  - Point-free style
    - Concatenative
- Generic
- Imperative (contrast: Declarative)
  - Procedural
  - Object-oriented
- Literate
- Language-oriented
  - Natural-language programming
  - Discipline-specific
  - Domain-specific
  - Grammar-oriented
  - Intentional
- Metaprogramming
  - Automatic
    - Inductive programming
  - Reflective
    - Attribute-oriented
  - Homoiconic
  - Macro
  - Template
- Non-structured (contrast: Structured)
  - Array
- Nondeterministic
- Parallel computing
  - Process-oriented
- Probabilistic
- Stack-based
- Structured (contrast: Non-structured)
  - Block-structured
  - Modular (contrast: Monolithic)
  - Object-oriented
    - Actor-based
    - Class-based
    - Concurrent
    - Prototype-based
    - By separation of concerns:
      - Aspect-oriented
      - Role-oriented
      - Subject-oriented
  - Recursive
- Symbolic
- Value-level (contrast: Function-level)
- Quantum programming

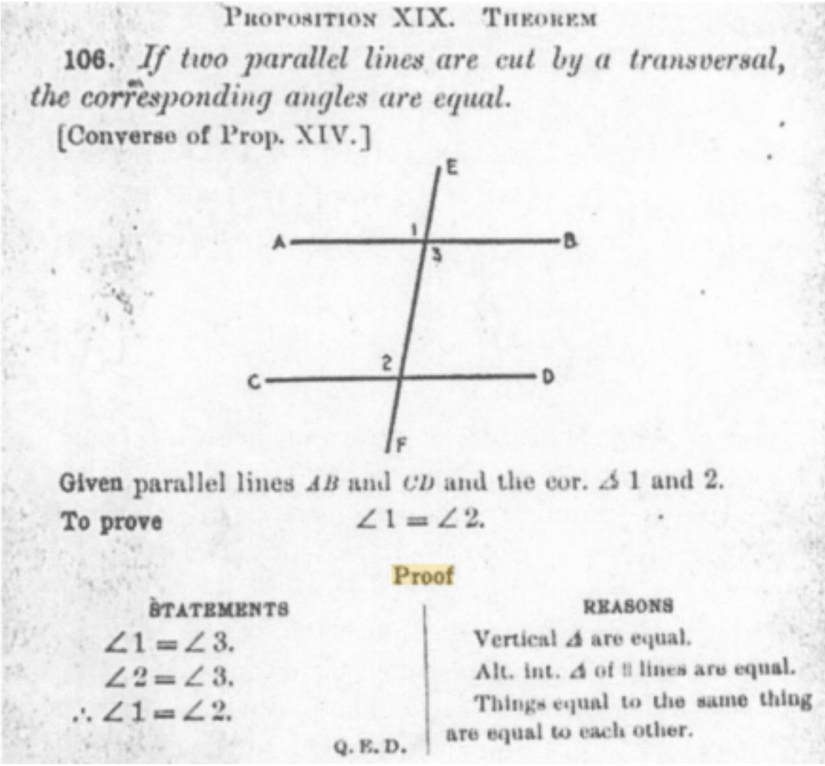
Python is a blend of these two styles!

Algorithms

https://goo.gl/NLYk72

What is an algorithm?

Are mathematical proofs
“algorithms?”



PROPOSITION XIX. THEOREM

106. *If two parallel lines are cut by a transversal, the corresponding angles are equal.*

[Converse of Prop. XIV.]

Given parallel lines AB and CD and the cor. $\angle$s 1 and 2.

To prove $\angle 1 = \angle 2$.

Proof

STATEMENTS	REASONS
$\angle 1 = \angle 3$.	Vertical $\angle$s are equal.
$\angle 2 = \angle 3$.	Alt. int. $\angle$s of $\parallel$ lines are equal.
$\therefore \angle 1 = \angle 2$.	Things equal to the same thing are equal to each other.

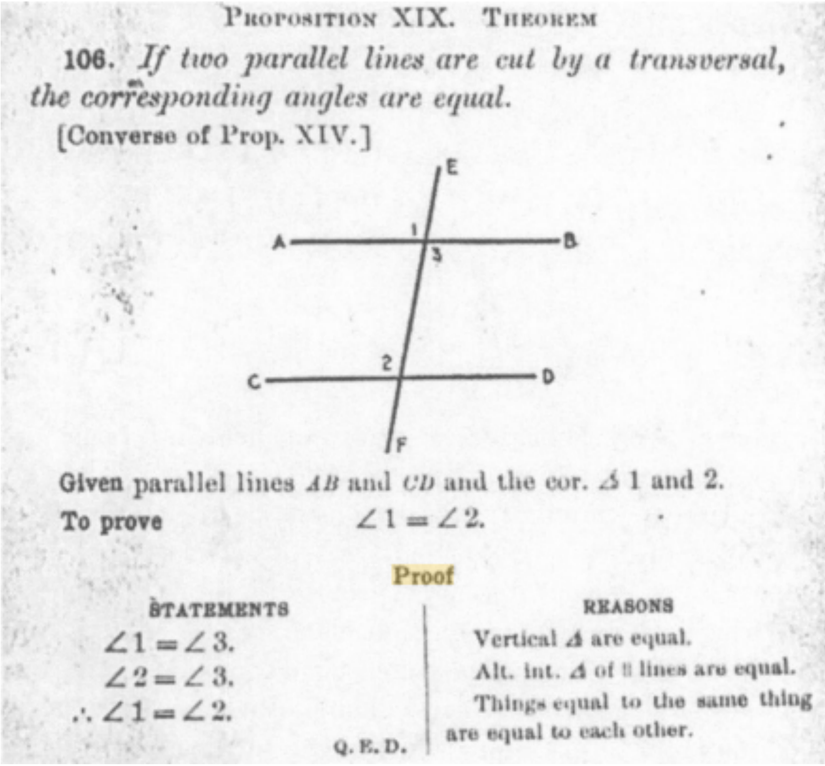
Q. E. D.

A two-column proof

What is an algorithm?

Proofs employ logic but usually include *some amount of natural language* which usually admits some **ambiguity**.

In fact, the vast majority of proofs in written mathematics can be considered as applications of **rigorous informal logic**.



PROPOSITION XIX. THEOREM

106. *If two parallel lines are cut by a transversal, the corresponding angles are equal.*

[Converse of Prop. XIV.]

Given parallel lines AB and CD and the cor. $\angle$s 1 and 2.

To prove $\angle 1 = \angle 2$.

Proof

STATEMENTS	REASONS
$\angle 1 = \angle 3$.	Vertical $\angle$s are equal.
$\angle 2 = \angle 3$.	Alt. int. $\angle$s of $\parallel$ lines are equal.
$\therefore \angle 1 = \angle 2$.	Things equal to the same thing are equal to each other.

Q. E. D.

A two-column proof

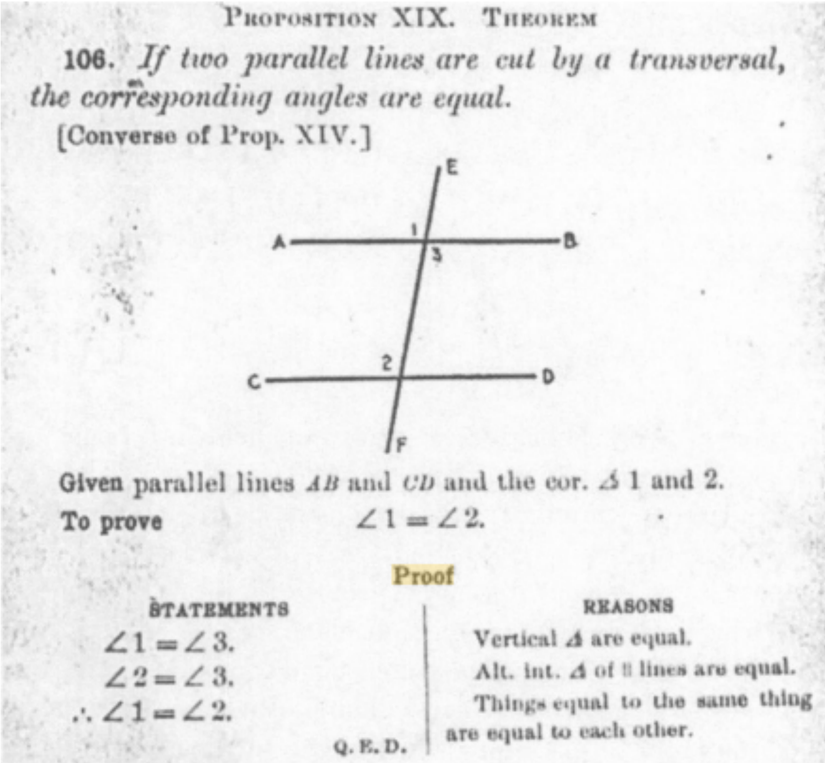
What is an algorithm?

Algorithms only represent **a small subset** of mathematics

(e.g. Euclidean algorithm for the greatest common divisor)

To show that an algorithm "works" one typically has to produce a (mathematical) proof.

Problem: how do we "prove" that complex software systems (e.g. your bank website) work?



PROPOSITION XIX. THEOREM

106. *If two parallel lines are cut by a transversal, the corresponding angles are equal.*

[Converse of Prop. XIV.]

Given parallel lines AB and CD and the cor. $\angle$s 1 and 2.

To prove $\angle 1 = \angle 2$.

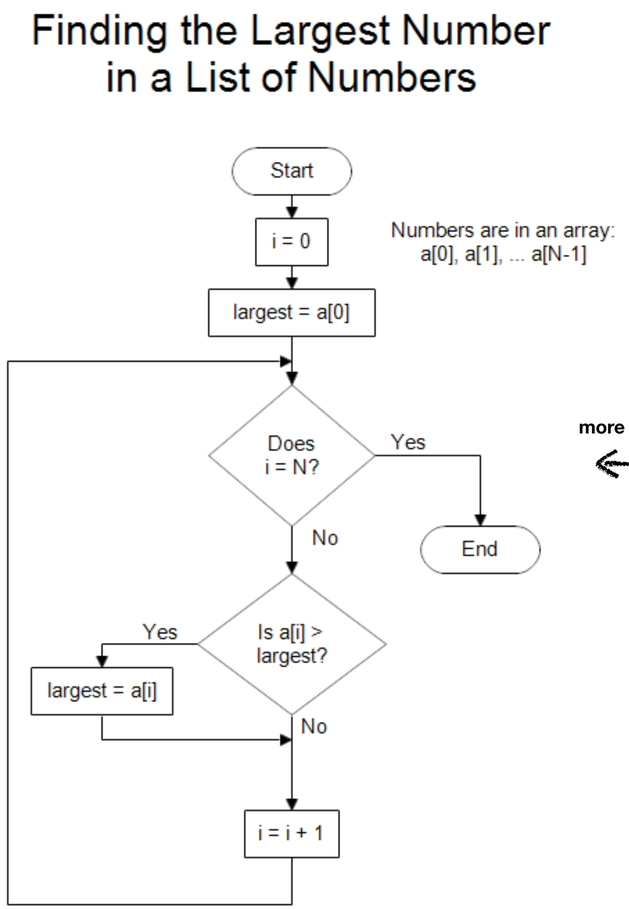
Proof

STATEMENTS	REASONS
$\angle 1 = \angle 3$.	Vertical $\angle$s are equal.
$\angle 2 = \angle 3$.	Alt. int. $\angle$s of $\parallel$ lines are equal.
$\therefore \angle 1 = \angle 2$.	Things equal to the same thing are equal to each other.

Q. E. D.

A two-column proof

What is an algorithm?



Flowchart

more

Ambiguity

less

```
Algorithm LargestNumber
  Input: A list of numbers L.
  Output: The largest number in the list L.

  if L.size = 0 return null
  largest ← L[0]
  for each item in L, do
    if item > largest, then
      largest ← item
  return largest
```

- "←" denotes assignment. For instance, "*largest* ← *item*" means that the value of *largest* changes to the value of *item*.
- "**return**" terminates the algorithm and outputs the following value.

Algorithm: “Find largest number in a list” pseudocode

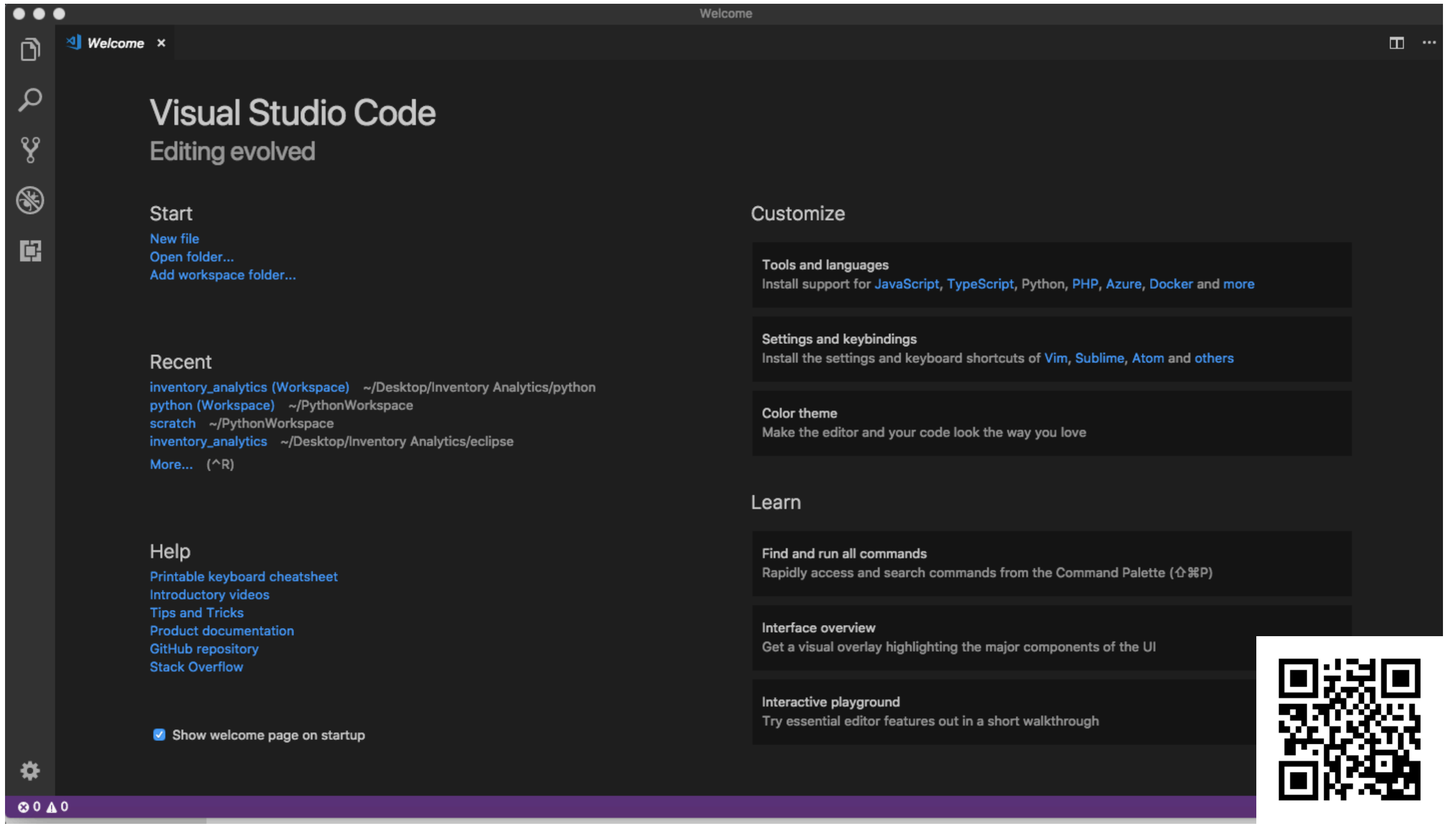
```python
a = [1, 2, 3, 4, 6, 7, 99, 88, 999]
max = 0
for i in a:
    if i > max:
        max = i
print(max)
```

Algorithm: “Find largest number in a list” in Python

Integrated Development Environment (IDE)

https://goo.gl/cwEGFH

Integrated Development Environment (IDE)



Assignment: set up a project folder as shown in

https://goo.gl/cwEGFH

Python

https://goo.gl/ErZZSt

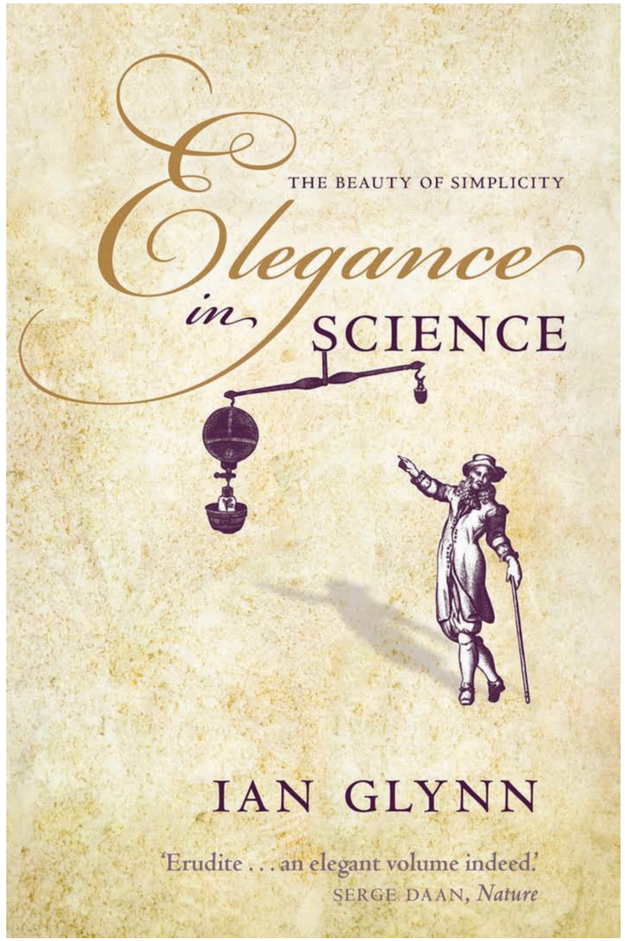
Why Python?

It is “**elegant**” (for simple codes).

It follows the philosophy of “**batteries included**:” a rich standard library is immediately available, without making the user download separate packages.

It allows “**rapid prototyping**” of small/medium projects.

You need to write much less compared to other languages, e.g. Java, to obtain the same result.



Why Python?

Java

```java
public class HelloWorld
{
    public static void main (String[] args)
    {
        System.out.println("Hello, world!");
    }
}
```

94 characters

```java
// print the integers from 1 to 9
public class PrintIntegers{
    public static void main (String[] args)
    {
        for (int i = 1; i < 10; i++)
        {
            System.out.println(i);
        }
    }
}
```

132 characters

Python

−77%

```python
print(“Hello, World!”)
```

21 characters

−80%

```python
print(*[x for x in range(11)])
```

26 characters

Why Python?

Java

```java
public class Employee
{
    private String myEmployeeName;
    private int    myTaxDeductions = 1;
    private String myMaritalStatus = "single";

    //--------- constructor #1 -------------
    public Employee(String EmployeName)
    {
        this(employeeName, 1);
    }

    //--------- constructor #2 -------------
    public Employee(String EmployeName, int taxDeductions)
    {
       this(employeeName, taxDeductions, "single");
    }

    //--------- constructor #3 -------------
    public Employee(String EmployeName,
           int taxDeductions,
           String maritalStatus)
    {
       this.employeeName    = employeeName;
       this.taxDeductions   = taxDeductions;
       this.maritalStatus   = maritalStatus;
    }
}
```

545 characters

Python

-66%

```python
class Employee():

    def __init__(self,
        employeeName,
        taxDeductions=1,
        maritalStatus="single"):

        self.employeeName    = employeeName
        self.taxDeductions   = taxDeductions
        self.maritalStatus   = maritalStatus
```

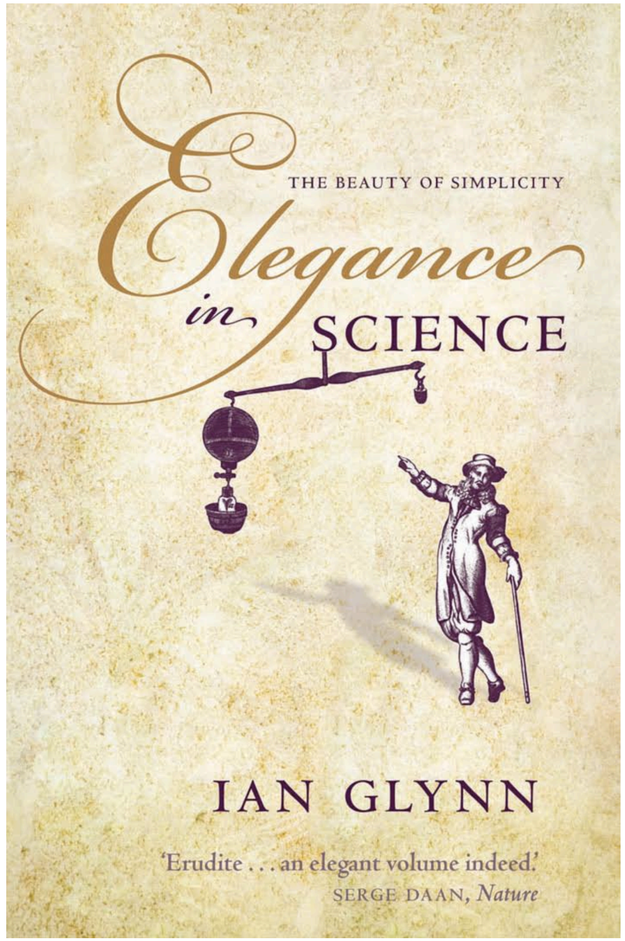
180 characters

Why not to use Python?

Code becomes hard to read and maintain when complexity of the project increases.

“**Duck typing**” (we will see this later) quickly becomes a problem as complexity of the project increases.

Debugging is a complex matter.

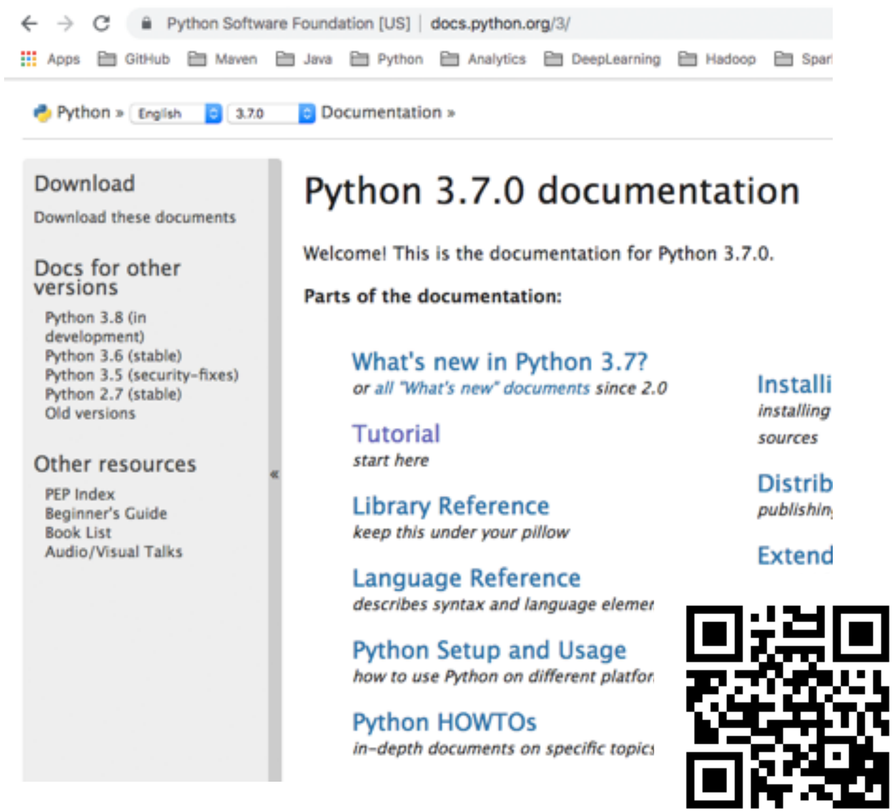


What’s the secret then?

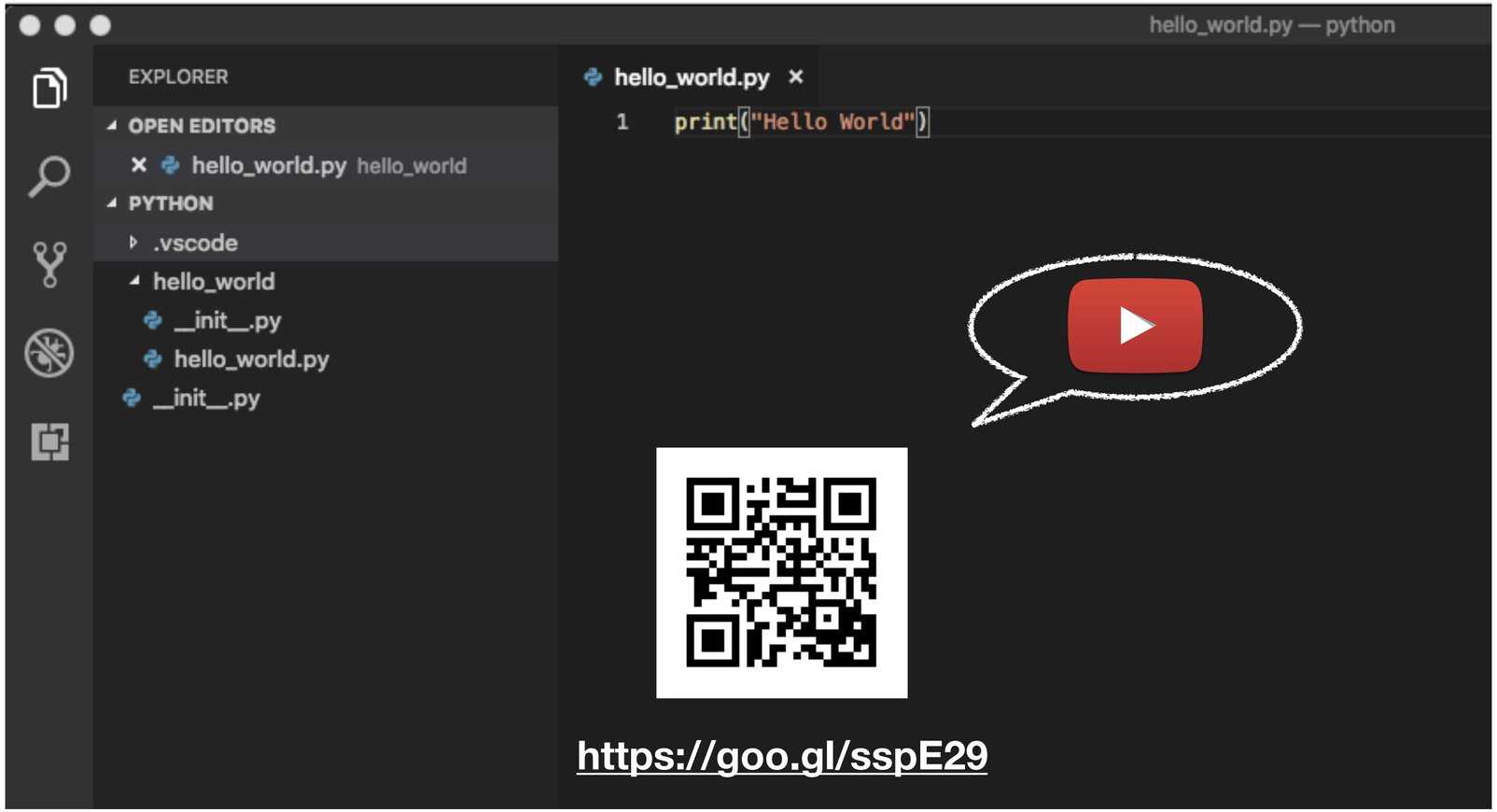
Be methodic and organised. If your code is messy and inconsistent, it will not work and you will never find why!

The aim of this lecture is to provide you with a **principled approach** and a **toolkit** to **achieve consistency** while you are coding.

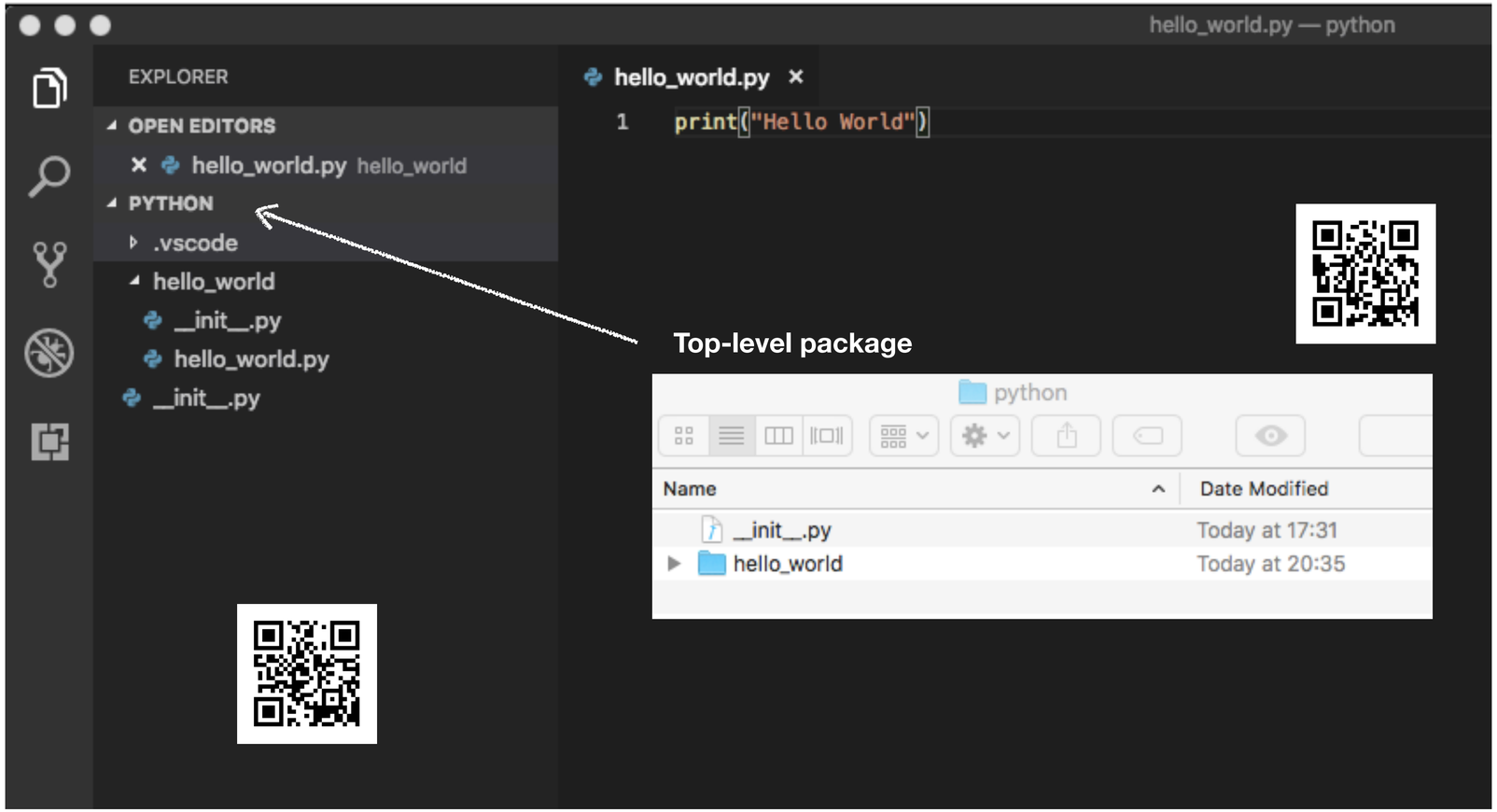
You will not be a code master in a few hours, but hopefully you will be **pointed to the right direction**.



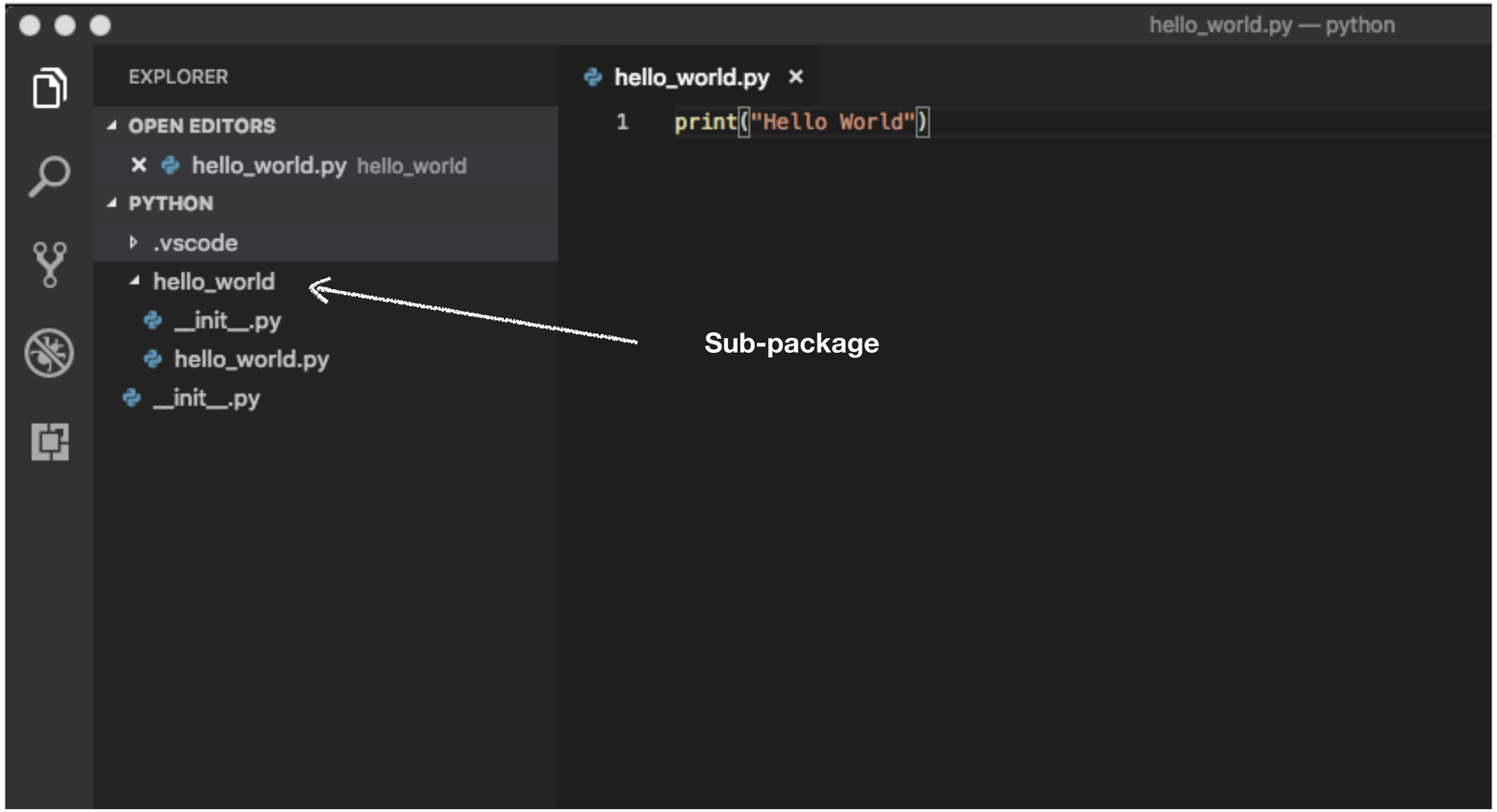
Hello World


hello_world.py — python
EXPLORER
OPEN EDITORS
hello_world.py hello_world
PYTHON
.vscode
hello_world
__init__.py
hello_world.py
__init__.py
hello_world.py
1 print("Hello World")
https://goo.gl/sspE29

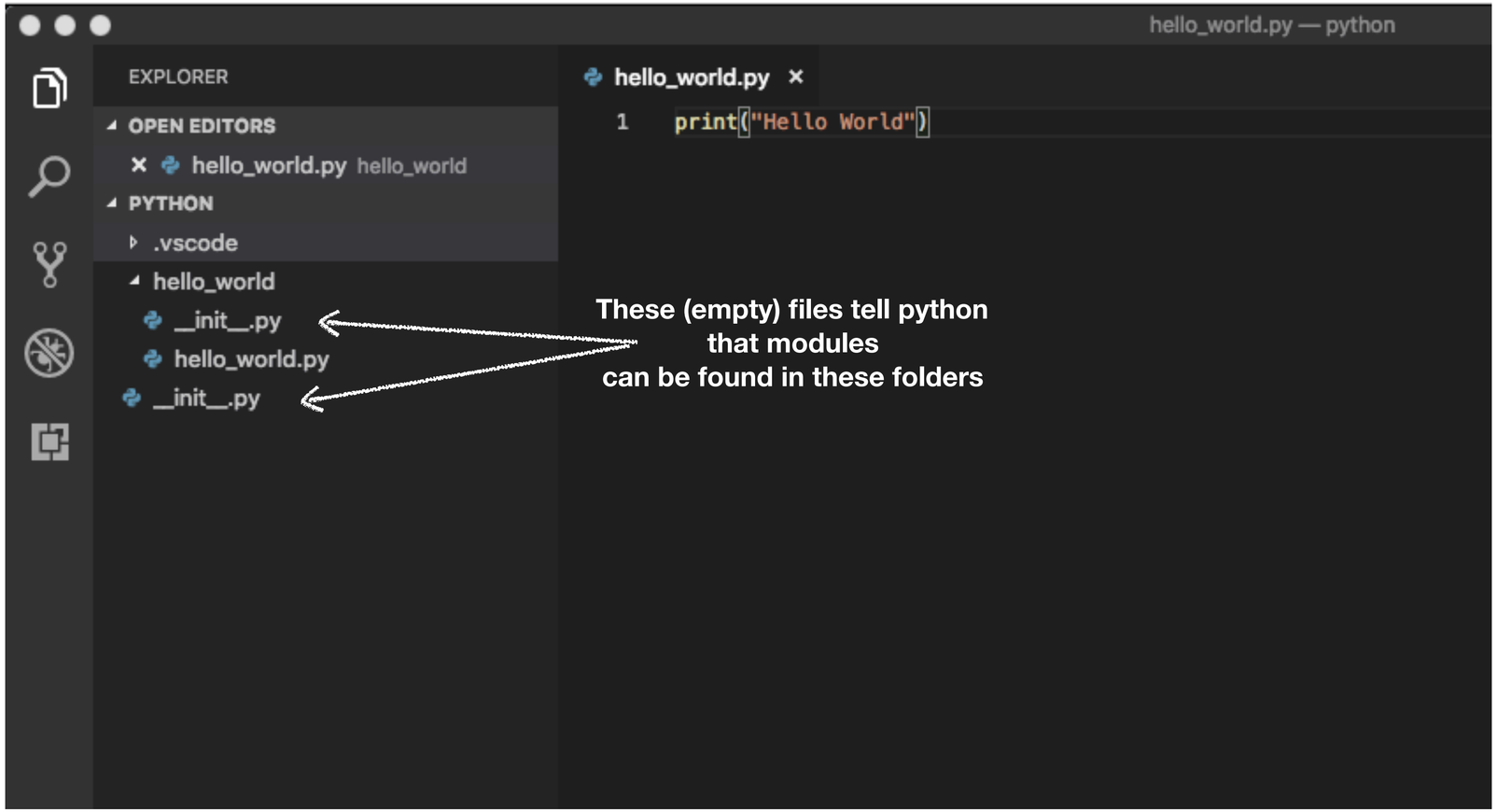
Python Packages



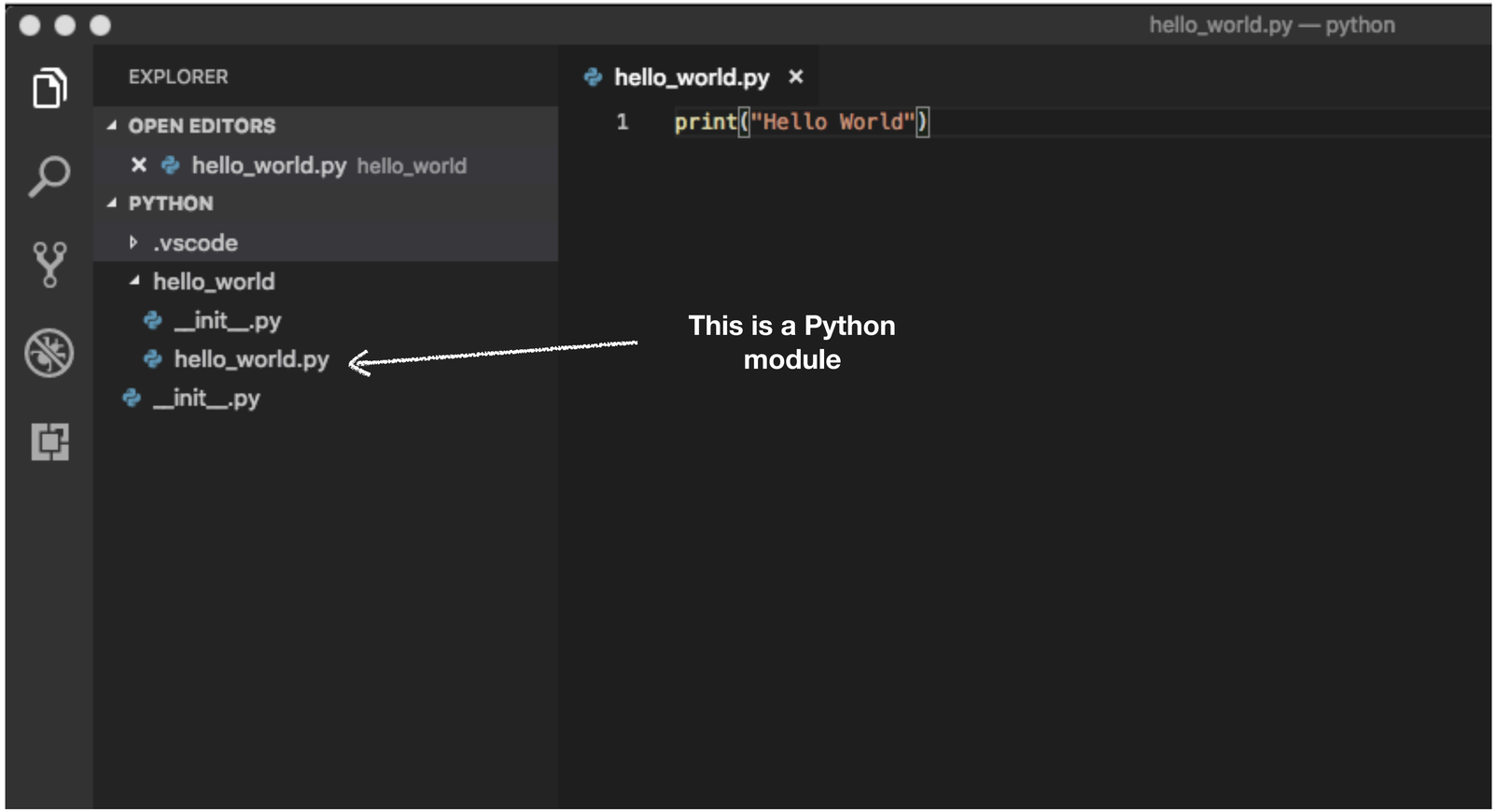
Python Packages



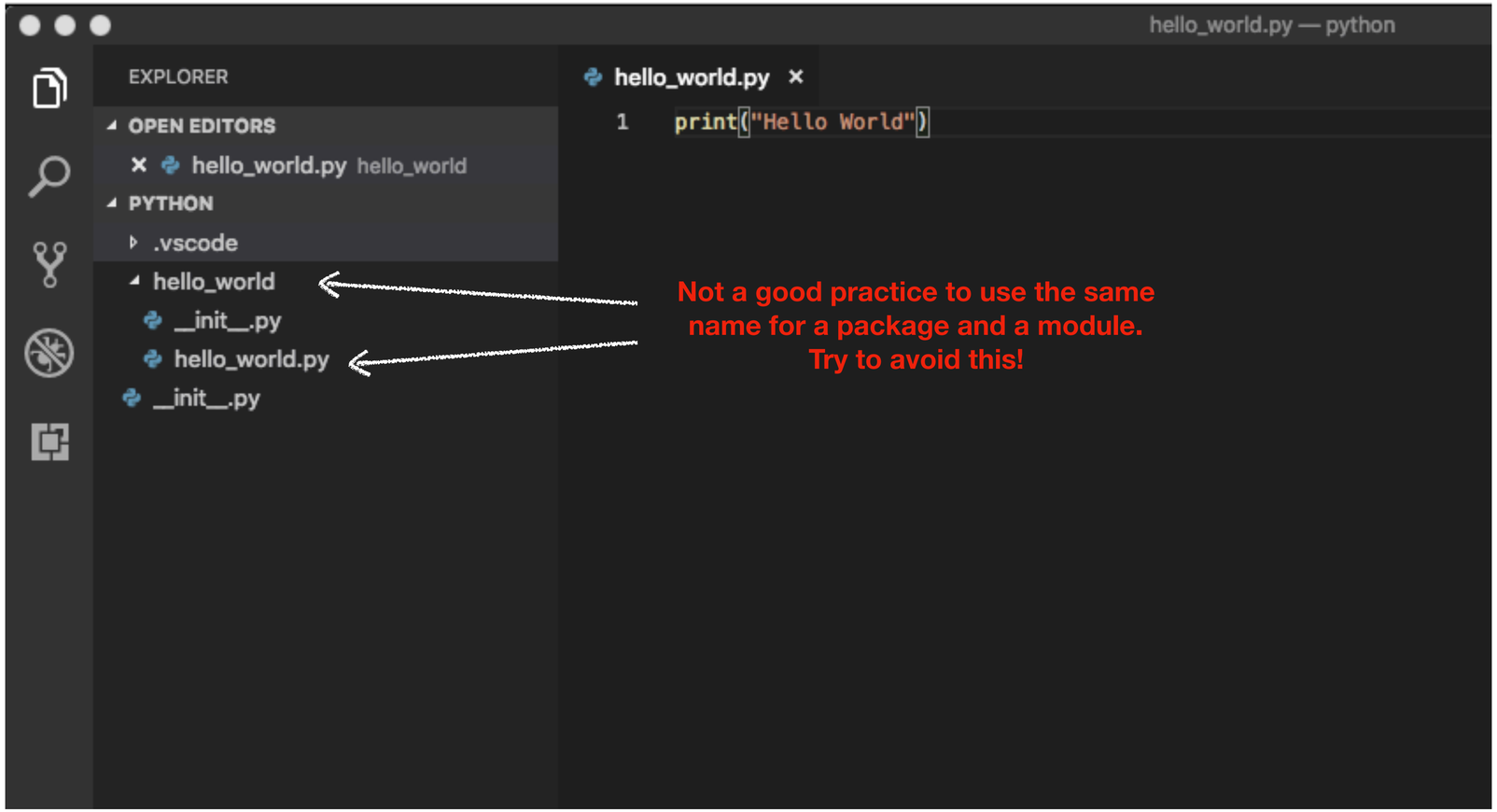
Python Packages



Python Packages



Python Packages



Hello World (better)

hello_world.py — python

EXPLORER

OPEN EDITORS

hello_world.py hello_world

PYTHON

.vscode

hello_world

__init__.py

hello_world.py

__init__.py

OUTLINE

Filter

hello_msg

hello_world.py

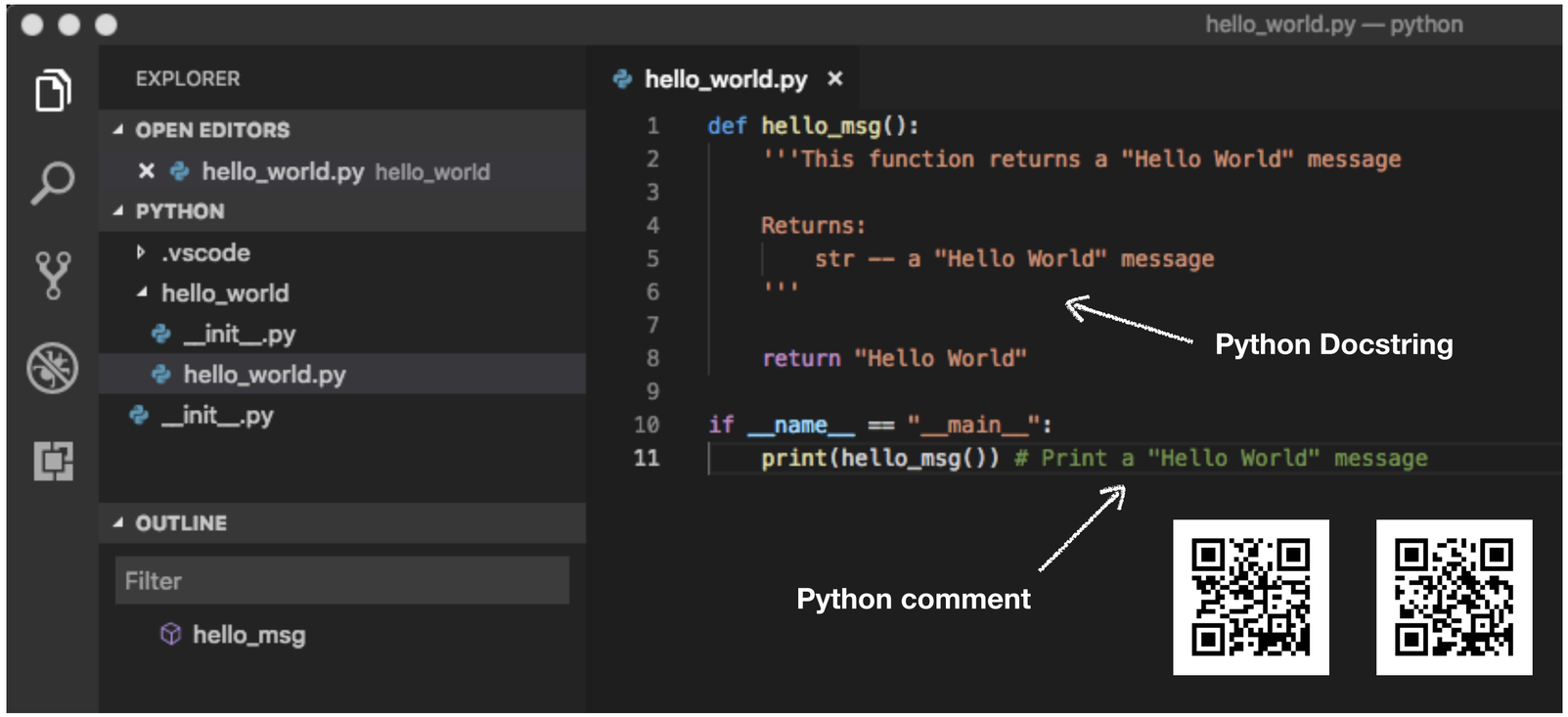
```
def hello_msg():
    return "Hello World"

if __name__ == "__main__":
    print(hello_msg())
```

Function definition

This code executes only if this specific module is executed

Remember: a function always returns a value, a procedure never.

Hello World (even better!)



Fibonacci series

fibonacci.py — intro_to_python (Workspace)

EXPLORER

OPEN EDITORS

fibonacci.py mathematics

INTRO_TO_PYTHON (WORKSPACE)

- python
  - .vscode
  - counter_package
  - hello_world
  - mathematics
    - __init__.py
    - fibonacci.py
  - __init__.py

fibonacci.py

```python
def fibonacci(n):
    '''This procedure prints the first n Fibonacci series elements
    '''
    a, b = 0, 1
    while a < n:
        print(a)
        a, b = b, a+b

if __name__ == "__main__":
    fibonacci(5) # Print the first five Fibonacci series elements
```

Remember: a function always returns a value, a procedure never.

Assignment: develop the Fibonacci module `fibonacci.py` in package `mathematics`

Strings

https://goo.gl/tPoQcg

Strings

```
# strings can be defined as follows
'this is a string'  # single quotes
"this is a string"  # double quotes
```

```
'doesn\'t' # use \' to escape single quote
"doesn't"  # or use double quotes instead
```

```
'"Yes," they said.'   # nested quotes
"\"Yes,\" they said." # or escaped quotes
```

```
'First line.\nSecond line.'  # \n means newline
r'First line.\nSecond line.' # r means raw string
```


```
'First line.\nSecond line.'  # \n means newline
r'First line.\nSecond line.' # r means raw string
```


```
"""String
spanning \
multiple
lines
""" # character \ prevents automatic end of line
```

```
"a" + "string" # string concatenation
'Py' 'thon'    # automatic concatenation
3 * "a"        # string repeat, produces "aaa"
```

```
('Put several strings within parentheses ' # use brackets
 'to have them joined together.') # to break long strings
```

```
# in Python strings are immutable lists of characters
string = 'a string' # create a string
string[0]           # returns 'a'
string[0] = 'b'     # TypeError: 'str' object does
                    # not support item assignment
len('a string')     # returns 8

# we will see later on how to manipulate lists
```

```
'Total = ' + 3      # TypeError: must be str, not int
str(3)              # converts number to string
'Total = ' + str(3) # ok
```

```
'Pi = ' + str(3.141592)          # number too long?
'Pi = ' + str(round(3.141592,2)) # round to 2 decimals
```

Variables & Operators

https://goo.gl/t9jY9C

Variables in Python

Variables are used to store information to be referenced and manipulated in a computer program.

Unlike other programming languages, Python has no command for declaring a variable.

A variable is created when you first assign a value to it.

The equal sign (=) is used to assign a value to a variable.

Note: `x = 5` really means `x ← 5`, but unfortunately there is no sign ← on your keyboard, so we use = for convenience.

Assignment Operators

Assignment operators in Python

Operator	Example	Equivatent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

Mathematics in Python

Arithmetic operators in Python

Operator	Meaning	Example
+	Add two operands or unary plus	x + y +2
-	Subtract right operand from the left or unary minus	x - y -2
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	x % y (remainder of x/y)
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x**y (x to the power y)

Mathematics in Python

Comparision operators in Python

Operator	Meaning	Example
>	Greater that - True if left operand is greater than the right	x > y
<	Less that - True if left operand is less than the right	x < y
==	Equal to - True if both operands are equal	x == y
!=	Not equal to - True if operands are not equal	x != y
>=	Greater than or equal to - True if left operand is greater than or equal to the right	x >= y
<=	Less than or equal to - True if left operand is less than or equal to the right	x <= y

Logical operators in Python

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Control Structures

https://goo.gl/SNx3n9

Control Structures

if

```python
# Read from standard input
x = int(input("Enter an integer: "))
if x < 0:
    print('Negative')
elif x == 0:
    print('Zero')
else:
    print('Positive')
```

while

```python
# prints "1 2 3 4 5"
a = 0
while a < 5:
    a = a + 1
    print(a, end= ' ' if a < 5 else '\n')
```

for

```python
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))
```

```python
# range(5) := [1, 2, 3, 4, 5]
# := means "is defined as"
for i in range(5):
    print(i)
# range(5, 10) := [5, 6, 7, 8, 9]
# range(0, 10, 3) := [0, 3, 6, 9]
```

continue

```python
for num in range(2, 10):
    if num % 2 == 0:
        print("Found an even number", num)
        continue # proceed to the next iteration by ignoring remaining statements
    print("Found a number", num)
```

break

```python
for num in range(1, 10):
    if num % 2 == 0:
        print("Found an even number", num)
        break # terminates the loop
    print("Found a number", num)
```

pass

```python
def initlog(*args):
    pass   # Remember to implement this!

# *args means a non-predefined number of
# arguments; args[i] is the i-th argument
```

List and Tuples

https://goo.gl/TTNZhC

Lists & Tuples

```python
# create a new list
squares = [1, 4, 9, 16, 25]
```

```python
# Removes first element
del squares[0]

# Removes a range
del squares[2:4]
```

```python
# Nested lists
a = [1, 2]
b = [3, 4]
c = [a, b]
print(c[0][0]) # prints 1
```

```python
# Create a tuple
t = 12345, 54321, 'hello!'

# Tuples are immutable:
t[0] = 88888
# TypeError: 'tuple' object
does not support item
assignment

# but they can contain
mutable objects:
v = ([1, 2, 3], [3, 2, 1])
v[0][0] = 2 # now the tuple
is ([2, 2, 3], [3, 2, 1])
```

```python
# List properties
len(squares)  # returns the length of the list

# List indexing
squares[0]    # returns the first item
squares[-1]   # first item starting from the end

# List slicing
squares[-3:]  # "slicing" returns a new list [9, 16, 25]

# List concatenation
squares + [36, 49] # returns [1, 4, 9, 16, 25, 39, 49]

# List update
squares[0] = 100    # updates list to [100, 4, 9, 16, 25]
```

```python
# Traditional list creation
squares = []
for x in range(5):
    squares = squares + [x**2]
print(squares) # prints [0, 1, 4, 9, 16]

# List comprehension (we will use this a lot!)
squares = [x**2 for x in range(5)]
print(squares) # prints [0, 1, 4, 9, 16]

# Lambda calculus (equivalent, but less elegant)
squares = list(map(lambda x: x**2, range(5)))
print(squares) # prints [0, 1, 4, 9, 16]
```

List comprehension

```
# List comprehension is incredibly expressive
[(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
# Returns [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]

# ...and is equivalent to
combs = []
for x in [1,2,3]:
    for y in [3,1,4]:
        if x != y:
            combs.append((x, y))
# ...which is of course much less effective!
```

Sets and Dictionaries

https://goo.gl/KcLDQE

Sets

```
# Sets (collections that do not allow duplicates)
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)      # duplicates have been removed: {'orange', 'banana', 'pear', 'apple'}
'orange' in basket # membership testing: True
'crabgrass' in basket

# Demonstrate set operations on unique letters from two words
a = set('abracadabra')
b = set('alacazam')
a                                  # unique letters in a
a - b                              # letters in a but not in b
a | b                              # letters in a or b or both
a & b                              # letters in both a and b
a ^ b                              # letters in a or b but not both

# Set comprehension
{x for x in 'abracadabra' if x not in 'abc'} # returns {'r', 'd'}
```

Dictionaries

```
# Dictionaries
telephone = {'jack': 4098, 'sape': 4139}
telephone['guido'] = 4127 # add entry {'jack': 4098, 'sape': 4139, 'guido': 4127}

telephone['jack']         # 4098
del telephone['sape']     # remove entry{'jack': 4098, 'guido': 4127}

list(telephone)           # return ['jack', 'guido']
sorted(telephone)         # sort dictionary

'guido' in tel            # test membership: True
'guido' not in tel        # test membership: False

# The dict() constructor builds dictionaries directly from sequences of key-value pairs
dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])        # {'sape': 4139, 'guido': 4127, 'jack': 4098}

# Dictionary comprehension
{x: x**2 for x in (2, 4, 6)}                                   # Returns {2: 4, 4: 16, 6: 36}
```

Iterators and Generators

https://goo.gl/FCf79h

Iterators & Generators

```python
# Iterators
for element in [1, 2, 3]:
    print(element) # prints 1 2 3

for element in (1, 2, 3):
    print(element) # prints 1 2 3

for key in {'one':1, 'two':2}:
    print(key)     # prints one two

for char in "123":
    print(char)    # prints 1 2 3
```

```python
# Generators
def return_odd_elements(n):     # create a function that returns odd
numbers
    for index in range(n):      # iterate from 0 to n-1
        if index % 2 > 0:       # if an odd index is found
            yield index         # add index to list to be returned

# the function returns a collection
print([x for x in return_odd_elements(10)])
```

Functions

https://goo.gl/yDEgHf

Functions

```
def sample_function(param1, param2):
    '''Docstring
    '''
    pass    # an empty function
```

4 sp.

List of parameters separated by comma.

A Docstring (optional)

Body of the function: statements that will be executed when the function is called.

```
# There are no procedures as such in Python; all functions return some
value. If no return statement is given, function returns None
```

```
def sample_function():
    '''This function sets a = 0 and then increments a.
    '''
    a = 0    # a new local variable is created in
    a += 1   # the local symbol table of the function
    b = 3

a = 5        # create a variable a in global symbol table
sample_function()
print(sample_function())    # prints 'None'
print(a)     # a is 5
print(b)     # NameError: name 'b' is not defined
             # b does not exists outside sample_function
```

```
def print_table(header, *persons):    # Arbitrary Argument Lists
    print(header+'\n---')             # print a header
    for p in persons:
        print(p)               # print all items in tuple persons

print_table('Name', 'John', 'Mike', 'Mark')  # as many names you like
```

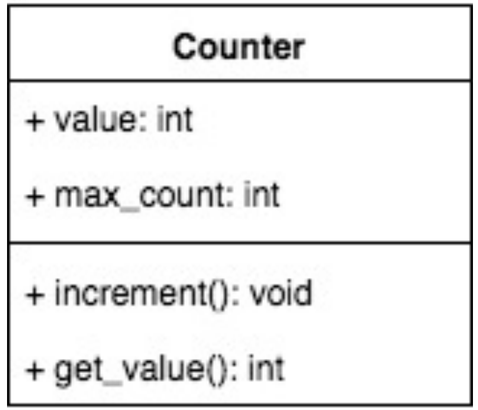
```
def swap(a):
    '''This function swaps elements in a.
    '''
    a[0], a[1] = a[1], a[0]    # typical Python assignment:
                               # read all on the right, assign to the left in order

l = [1,2]    # create a list in global symbol table
swap(l)      # call by value, where value is a reference to the object
print(l)     # list a has been swapped
```

```
def add_person(pb, name, surname, phone='not set'):
    '''This function adds a person to a phonebook
    '''
    pb[(name, surname)] = phone

phonebook = {}      # creates an empty phonebook
add_person(phonebook, 'John', 'Doe', '07823472222')   # standard call
add_person(phonebook, 'Foo', 'Bar')                   # default value used for phone
print(phonebook)    # {('Foo', 'Bar'): 'not set',
                    #  ('John', 'Doe'): '07823472222'}

person = ['John', 'Muir', '07424552345']             # person as a list
add_person(phonebook, *person)                       # argument unpacking
add_person(phonebook, surname='Mike', name='White')  # keyword arguments
person = {'surname': 'Mike',
          'name': 'White', 'phone': '07424552345'}   # person as a dictionary
add_person(phonebook, **person)          # arguments unpacked from dictionary
print(phonebook)                         # {('John', 'Doe'): '07823472222',
                                         #  ('Foo', 'Bar'): 'not set',
                                         #  ('John', 'Muir'): '07424552345',
                                         #  ('White', 'Mike'): '07424552345'}
```

Object-Oriented Programming

https://goo.gl/W4QERH

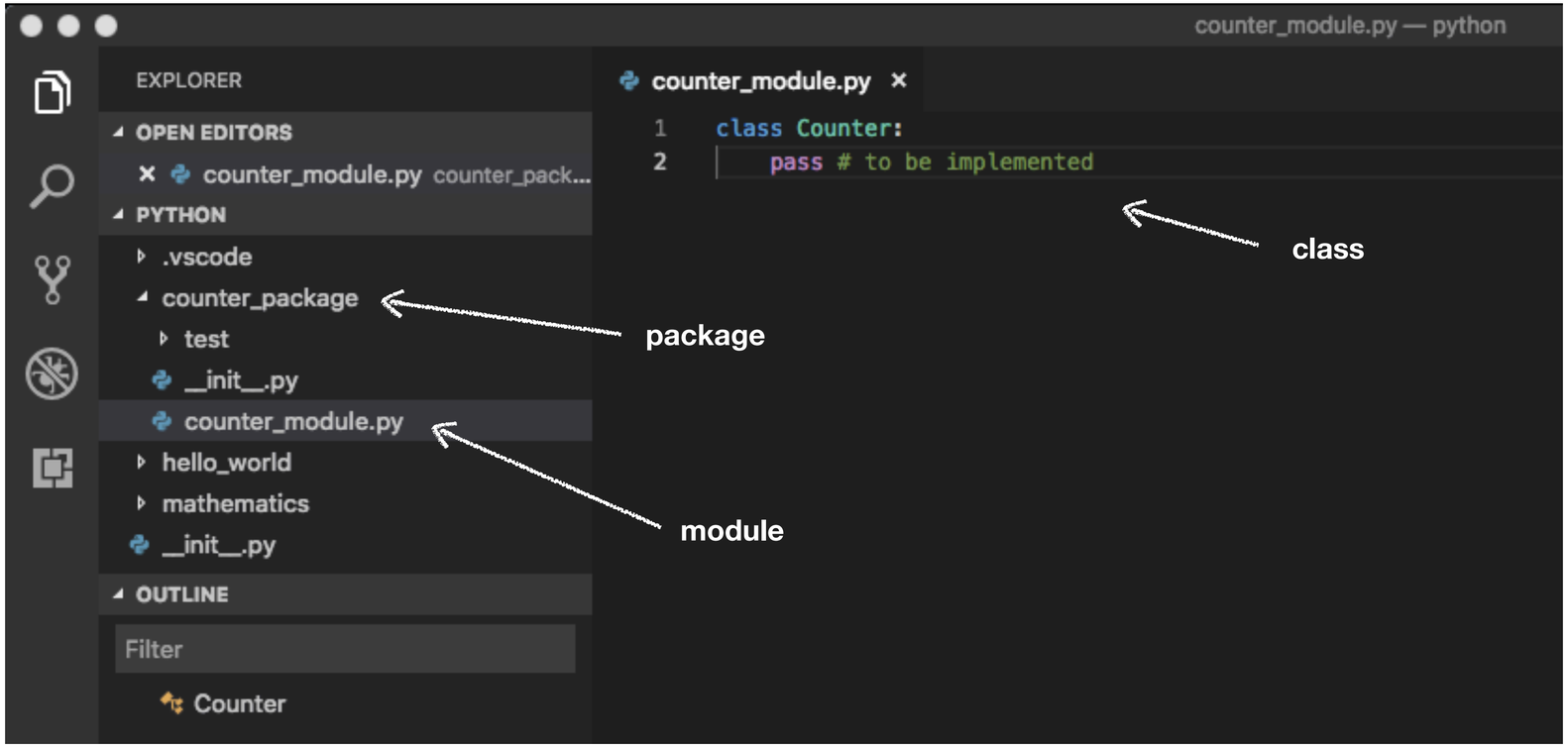
Class



Classes provide a means of **bundling data and functionality together**.

Creating a new class creates **a new type of object**, allowing new instances of that type to be made.

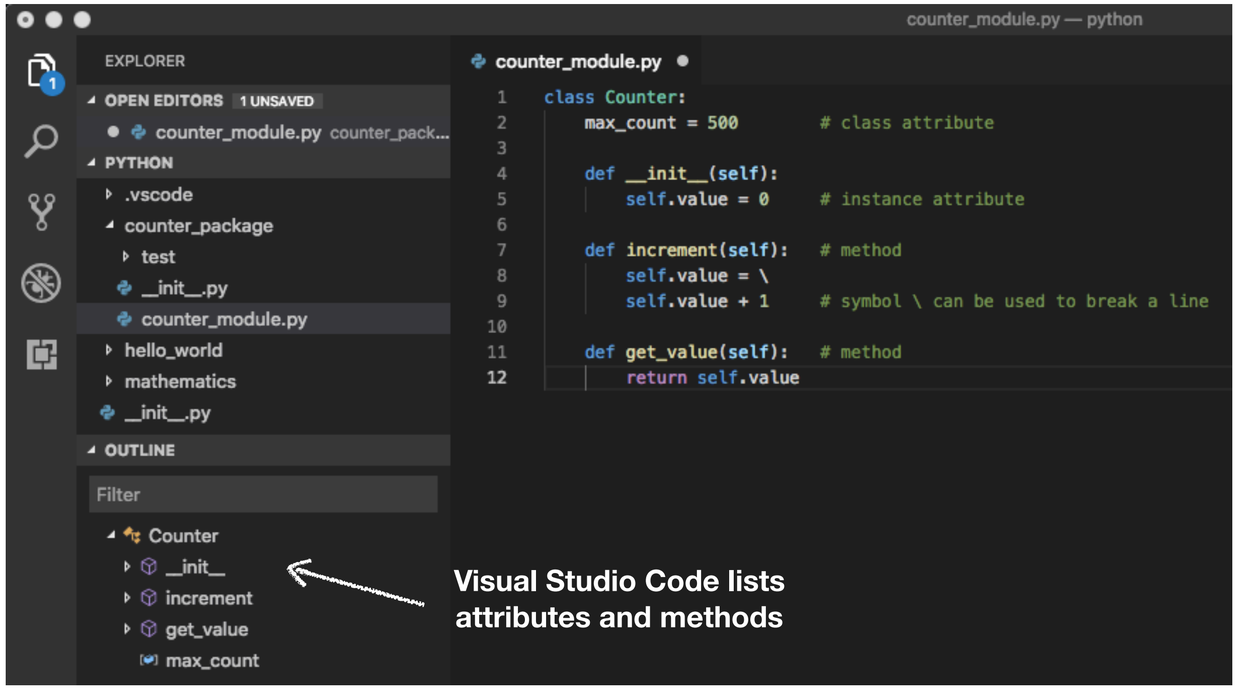
Class



Classes provide a means of **bundling data and functionality together**.

A class defines **a new type of object**, allowing new instances of that type to be made.

Class

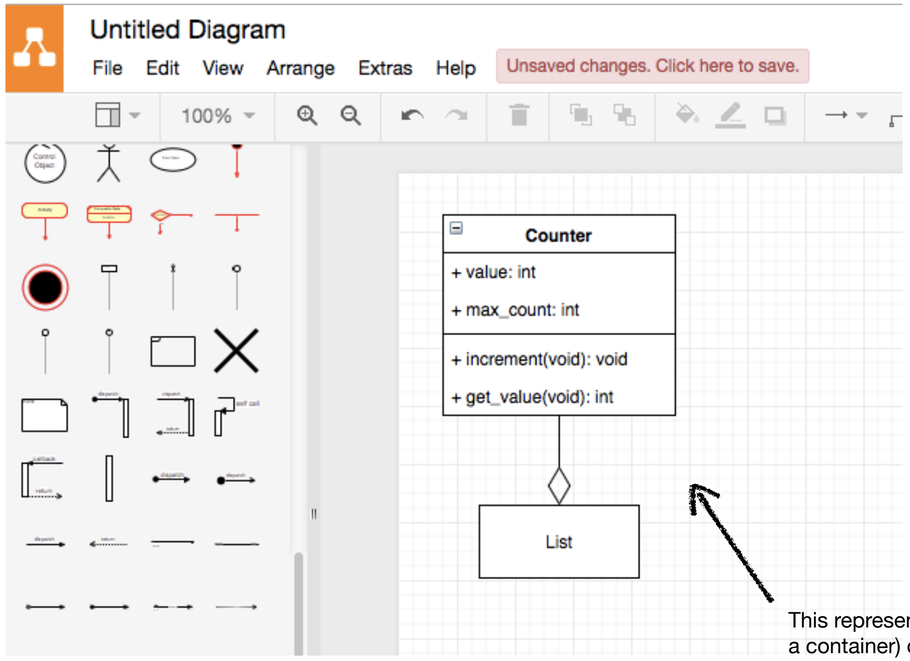


Each class instance can have **attributes** attached to it for maintaining its state.

Class attributes belong to the class (and not to individual instances).

Class instances can also have **methods** (defined by its class) for modifying its state.

Class




```
class Counter:
    max_count = 500

    def __init__(self):
        self.value = 0

    def increment(self):
        self.value = \
        self.value + 1

    def get_value(self):
        return self.value
```

This represents a class **`List`** as an aggregation (essentially a container) of objects of type **`Counter`**.

Software engineers typically use **graphical languages** (e.g. UML) to model complex projects involving many classes.

Class

counter_module.py — python

EXPLORER

OPEN EDITORS

counter_module.py counter_pack...

PYTHON

.vscode

counter_package

test

__init__.py

counter_module.py

hello_world

mathematics

__init__.py

OUTLINE

Filter

Counter

__init__

increment

get_value

max_count

c

counter_module.py

```python
class Counter:
    max_count = 500        # class attribute

    def __init__(self):
        self.value = 0     # instance attribute

    def increment(self):   # method
        self.value = \
        self.value + 1     # symbol \ can be used to break a line

    def get_value(self):   # method
        return self.value

if __name__ == "__main__":
    c = Counter()          # create a counter and bind to c
```

instantiation

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Instantiation statement `c = Counter()` creates **a new instance** of the class and bind local variable `x` to **this object**.

Method `__init__(self)` is **automatically invoked** whenever a new instance of the class is created; this method is employed to **initialise** the instance.

“Assignments do not copy data —
they just bind names to objects”

“Assig t bind

```python
class Counter:
    max_count = 500        # class attribute

    def __init__(self):
        self.value = 0     # instance attribute

    def increment(self):   # method
        self.value = \
        self.value + 1     # symbol \ can be used to break a line

    def get_value(self):   # method
        return self.value

if __name__ == "__main__":
    c1 = Counter()         # create a counter and bind to c1
    c2 = c1                # bind the counter to c2 as well
    c1.increment()         # increment the counter
    del c1                 # unbind c1
    print(c2.get_value())  # the object still exists!
    print(c1.get_value())  # NameError: name 'c1' is not defined
```

–Python Tutorial

Class vs Instance Attribute

ADVANCED

```python
class Counter:
    max_count = 500        # class attribute

    def __init__(self, initial_value):
        self.value = initial_value     # initialise counter

    def increment(self):   # method
        self.value = \
        self.value + 1     # symbol \ can be used to break a line

    def get_value(self):   # method
        return self.value

if __name__ == "__main__":
    c = Counter(5)           # create a counter initialised to 5
    c.max_count = 10         # create a new instance attribute max_count
    print(c.max_count)       # print instance attribute; returns 10
    print(Counter.max_count) # print class attribute; returns 100
```

Class vs Static Methods

ADVANCED

```python
import math                # Import system library

class Counter:
    max_count = 500        # class attribute

    def __init__(self, initial_value):
        self.value = initial_value     # initialise counter

    def increment(self):   # method
        self.value = \
        self.value + 1     # symbol \ can be used to break a line

    def get_value(self):   # method
        return self.value

    @classmethod
    def set_max_count(cls, max):
        cls.max_count = max

    @staticmethod
    def square_root(n):
        return math.sqrt(n)    # static method to compute sqrt

if __name__ == "__main__":
    print(Counter.max_count)       # print class attribute
    Counter.set_max_count(10)      # modify class attribute
    print(Counter.max_count)       # print class attribute
    print(Counter.square_root(36)) # use static method
```

Inheritance

ADVANCED

```python
import math

class Counter:
    max_count = 500        # class attribute

    def __init__(self, initial_value):
        self.value = initial_value     # initialise counter

    def increment(self):   # method
        self.value = \
        self.value + 1     # symbol \ can be used to break a line

    def get_value(self):   # method
        return self.value

class CounterPlus(Counter):
    # CounterPlus "inherits" all attributes and methods of Counter
    def decrement(self):   # new method
        self.value = \
        self.value - 1     # symbol \ can be used to break a line

if __name__ == "__main__":
    cp = CounterPlus(5)    # create a CounterPlus
    cp.increment()         # a CounterPlus inherits method increment from Counter
    cp.decrement()
    print(cp.get_value())  # a CounterPlus inherits method get_value from Counter

    c = Counter(5)         # create a Counter
    c.decrement()          # AttributeError: 'Counter' object has no attribute 'decrement'
```

Why do I need to know about OO?

```python
if __name__ == "__main__":
    a = []      # create a list
    print(a)    # []
    a.append(1) # append an element
    print(a)    # [1]
    a.append(2) # append an element
    print(a)    # [1, 2]
    a.remove(2) # remove first occurrence of 2
    print(a)    # [1] it turns out lists are objects too!
```

In fact, most of the standard modules you will end up using will be OO…

Errors and Exceptions

https://goo.gl/Mr7oeE

Errors and Exceptions

In Python there are (at least) two distinguishable kinds of errors:
syntax errors and **exceptions**.

```
while True print('Hello world')  # syntax error (colon missing)

while True: print('Hello world') # ok
```

Even if a statement or expression is **syntactically correct**, it may cause an error when an attempt is made to execute it.

Errors detected during execution are called **exceptions** and are not unconditionally fatal.

You must learn how to handle them in Python programs.

Errors and Exceptions

```
10 * (1/0)  # ZeroDivisionError: division by zero
4 + spam*3  # NameError: name 'spam' is not defined
‘2' + 2     # TypeError: must be str, not int
```

Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError` and `TypeError`.

Handling Exceptions

```
try:
    x = int(input("Please enter a number: ")) # tries to convert from standard input to int
except ValueError:                            # catches ValueError if not int
    print("Oops!  That was no valid number.  Try again...")
except Exception as err:
    print("Name error: {0}".format(err))      # formatted print for a generic exception
                                              # other than ValueError
else:                                         # (optional) executes if no exception raised
    print("Number entered: " + str(x))
finally:                                      # (optional) executes under all circumstances
    print("Always printed.")
```

Errors and Exceptions

```
10 * (1/0)  # ZeroDivisionError: division by zero
4 + spam*3  # NameError: name 'spam' is not defined
‘2' + 2     # TypeError: must be str, not int
```

Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError` and `TypeError`.

Raising Exceptions

```
class NewException(Exception):                    # user defined exception
    pass

if __name__ == "__main__":
    try:
        raise NameError(‘An error has occurred’) # raise NameError
    except NameError as err:                      # catches NameError
        print("Name error: {0}".format(err))      # formatted print
```

Errors and Exceptions

Fibonacci Series

```
def fibonacci(n):
    '''This procedure prints the first n Fibonacci series elements
    '''
    if not(isinstance(n, int)): # tests if n is integer
        raise Exception('Fibonacci takes only integer values.')

    a, b = 0, 1
    while a < n:
        print(a)
        a, b = b, a+b

if __name__ == "__main__":
    fibonacci(5.5) # Print the first five Fibonacci series elements
```

Output

```
line 5, in fibonacci raise Exception('Fibonacci takes only integer values.')
Exception: Fibonacci takes only integer values.
```

errors exceptions

“There are no ~~mistakes~~, only happy little ~~accidents~~.”

–Roberto Rossi

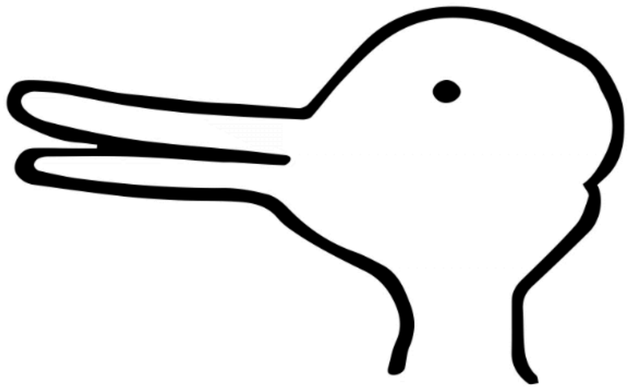
~~–*Robert (Bob) Ross*~~

Duck Typing

https://goo.gl/9hsAzu

Duck Typing

“If it walks like a duck and it quacks like a duck, then it must be a duck (or a rabbit?)”



Duck Typing

```
class Duck():
    def quack(self):
        return 'Duck Quack!'

class Goose():
    def quack(self):
        return 'Goose Quack!'

class Dog():
    pass

# Generators
def animals_who_quack(animals):
    for a in animals:
        try:
            yield a.quack()                     # duck typing (i.e. try and see if it works)
        except AttributeError:                  # be type agnostic: use duck typing and exceptions
            pass

if __name__ == "__main__":
    duck = Duck()
    goose = Goose()
    dog = Dog()
    animals = [duck, goose, dog]
    print([x for x in animals_who_quack(animals)]) # you don't need to know what animal you are dealing with
```

Test-driven Development

https://goo.gl/KP9k3P

“Where shall I begin, please your Majesty?” he asked.
“Begin at the end,” the King said gravely,
“and go on till you come to the beginning: then stop.”

#testdrivendevelopment

test_counter.py — intro_to_python (Workspace)

EXPLORER

OPEN EDITORS

test_counter.py counter_package...

INTRO_TO_PYTHON (WORKSPACE)

- python
  - .vscode
  - counter_package
    - test
      - __init__.py
      - test_counter.py
    - __init__.py
    - counter_module.py
  - hello_world
  - mathematics
  - __init__.py

OUTLINE

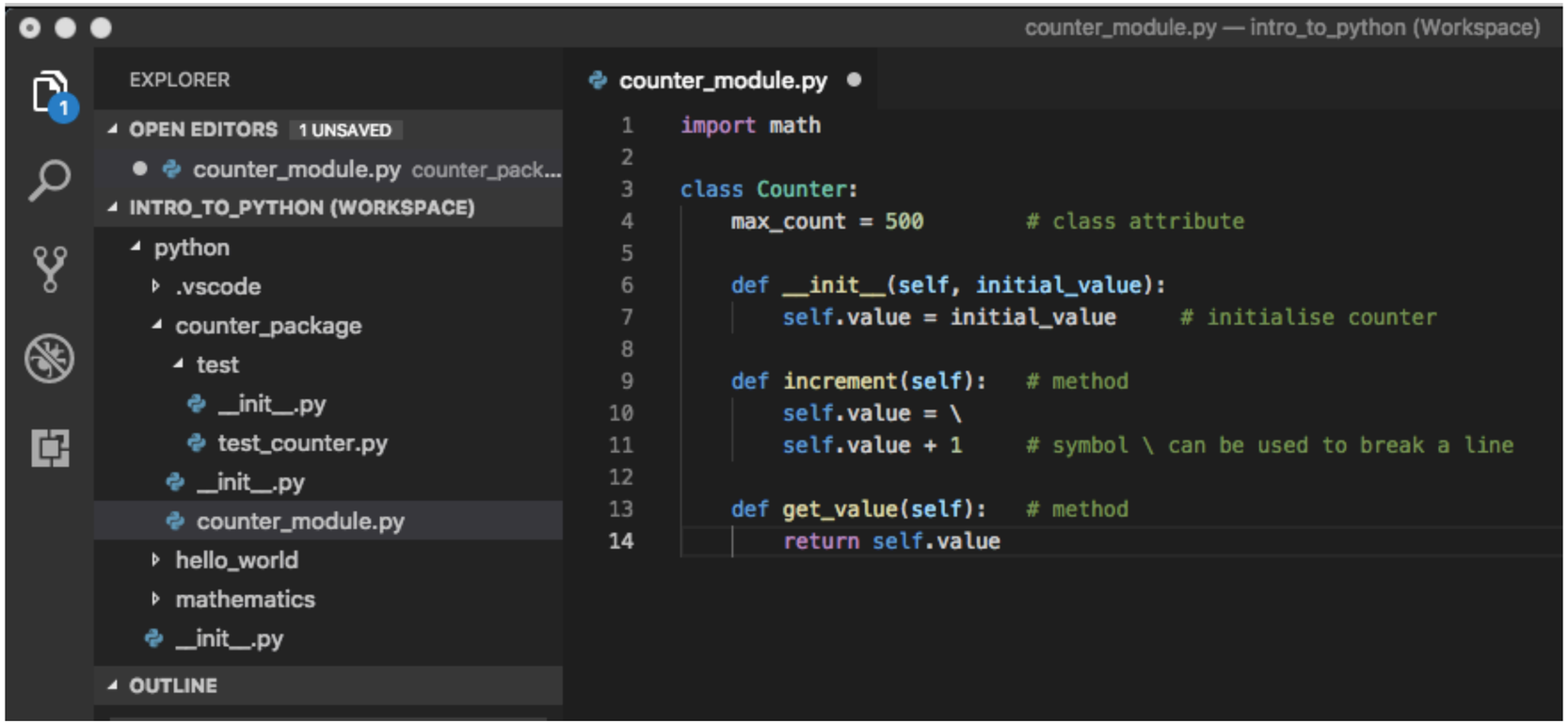
Filter

test_counter.py

```python
import unittest
import counter_package.counter_module as cm

Run Test | Debug Test
class TestCounter(unittest.TestCase):
    # extend class unittest.TestCase

    def setUp(self):
        # initialise everything needed in the test
        self.c = cm.Counter(0)

    def tearDown(self):
        # execute actions before terminating, e.g. close files
        pass

    Run Test | Debug Test
    def testCounter(self):
        # test case
        self.c.increment()
        self.assertEqual(self.c.get_value(), 1)
```

import our `counter_module` and define an alias `cm`

First write the test procedures ("begin at the end")…



… and then implement relevant classes/methods ("walk your way back!").

counter_module.py — intro_to_python (Workspace)

EXPLORER

OPEN EDITORS 1 UNSAVED

counter_module.py counter_pack...

INTRO_TO_PYTHON (WORKSPACE)

- python
  - .vscode
  - counter_package
    - test
      - __init__.py
      - test_counter.py
    - __init__.py
    - counter_module.py
  - hello_world
  - mathematics
  - __init__.py

OUTLINE

counter_module.py

```python
import math

class Counter:
    max_count = 500        # class attribute

    def __init__(self, initial_value):
        self.value = initial_value     # initialise counter

    def increment(self):   # method
        self.value = \
        self.value + 1     # symbol \ can be used to break a line

    def get_value(self):   # method
        return self.value
```

0 0 Anaconda, Inc. Python 3.6.4 (base) ✓1

Regularly verify that your code passes all tests!

Extras

Reading and Writing Files

```python
with open('workfile.txt', 'w') as f:     # predefined Clean-up Actions (see python tutorial)
    f.write('This is a test\n')
print(f.closed)

with open('workfile.txt') as f:
    read_data = f.read()      # alternatively f.readline() reads a single line from the file
    print(read_data)
print(f.closed)
```

Handling Input Arguments

```python
import sys # Import system library

def hello_msg():
    '''This procedure prints an information message including:
       the name of the script;
       the number of arguments passed to the script;
       the value of these arguments.
    '''

    print("This is the name of the script: ", sys.argv[0])
    print("Number of arguments: ", len(sys.argv))
    print("The arguments are: " , str(sys.argv))

if __name__ == "__main__":
    hello_msg()                 # Print an information message
```

Terminal

```
$ python arguments.py "first argument"
This is the name of the script:  arguments.py
Number of arguments:  2
The arguments are:  ['arguments.py', 'first argument']
$
```

Coding Style

- **Use 4-space indentation, and no tabs.** 4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.

- **Wrap lines so that they don't exceed 79 characters.** This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.

- **Use blank lines to separate functions and classes, and larger blocks of code inside functions.**

- **When possible, put comments on a line of their own.**

- **Use docstrings.**

Coding Style

- **Use spaces around operators and after commas,** but not inside bracketing constructs: `a = f(1, 2) + g(3, 4).`

- **Name your classes and functions consistently;** the convention is to use CamelCase for classes and lower_case_with_underscores for functions and methods. Always use self as the name for the first method argument.

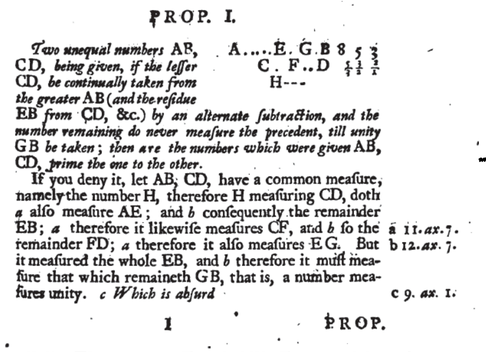
- **Don’t use fancy encodings if your code is meant to be used in international environments.** Python’s default, UTF-8, or even plain ASCII work best in any case.

- **Likewise, don’t use non-ASCII characters in identifiers** if there is only the slightest chance people speaking a different language will read or maintain the code.

Assignments

Euclid’s GCD Algorithm

- Develop a Python implementation of Euclid’s GCD algorithm. Use a test-driven development approach!



PROP. I.

Two unequal numbers AB, CD, *being given, if the lesser* CD, *be continually taken from the greater* AB (*and the residue* EB *from* CD, &c.) *by an alternate subtraction, and the number remaining do never measure the precedent, till unity* GB *be taken; then are the numbers which were given* AB, CD, *prime the one to the other.*

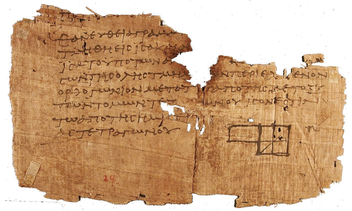
A.....E. G.B 8 5 3
C. F..D 5 3 2
H---

If you deny it, let AB, CD, have a common measure, namely the number H, therefore H measuring CD, doth *a* also measure AE; and *b* consequently the remainder EB; *a* therefore it likewise measures CF, and *b* so the remainder FD; *a* therefore it also measures EG. But it measured the whole EB, and *b* therefore it must measure that which remaineth GB, that is, a number measures unity. *c Which is absurd*

a 11. ax. 7.
b 12. ax. 7.
c 9. ax. 1.

I PROP.

Euclid's Elements, Book VII, Proposition 1,
by Isaac Barrow, Master of Trinity College, Cambridge



A fragment of Euclid's Elements on part of the Oxyrhynchus papyri

When two unequal numbers are set out, and the less is continually subtracted in turn from the greater, if the number which is left never measures the one before it until a unit is left, then the original numbers are relatively prime.

Euclid's Elements, Book VII, Proposition 1

Pseudocode implementation

The algorithm can be expressed as:

```
function gcd(a, b)
    while a ≠ b
        if a > b
            a := a − b;
        else
            b := b − a;
    return a;
```

…of course there are plenty of Python implementations on line,
but try to come up with your implementation!

https://goo.gl/wwQz5A

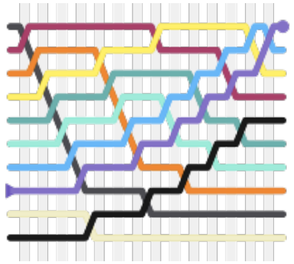
Bubble Sort Algorithm

- Develop a Python implementation of the Bubble Sort algorithm. Use a test-driven development approach!

Pseudocode implementation [edit]

The algorithm can be expressed as (0-based array):

```
procedure bubbleSort( A : list of sortable items )
    n = length(A)
    repeat
        swapped = false
        for i = 1 to n-1 inclusive do
            /* if this pair is out of order */
            if A[i-1] > A[i] then
                /* swap them and remember something changed */
                swap( A[i-1], A[i] )
                swapped = true
            end if
        end for
    until not swapped
end procedure
```



Static visualisation of bubble sort

…of course there are plenty of Python implementations on line, but try to come up with your implementation!

https://goo.gl/r828xJ

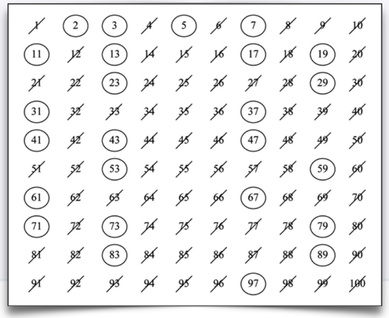
Eratostene’s Sieve

- Develop a Python implementation of Eratostene’s sieve. Use a test-driven development approach!

Pseudocode [edit]

The sieve of Eratosthenes can be expressed in pseudocode, as follows:[7][8]

```
Input: an integer n > 1.

Let A be an array of Boolean values, indexed by integers 2 to n,
initially all set to true.

for i = 2, 3, 4, ..., not exceeding √n:
  if A[i] is true:
    for j = i², i²+i, i²+2i, i²+3i, ..., not exceeding n:
      A[j] := false.

Output: all i such that A[i] is true.
```



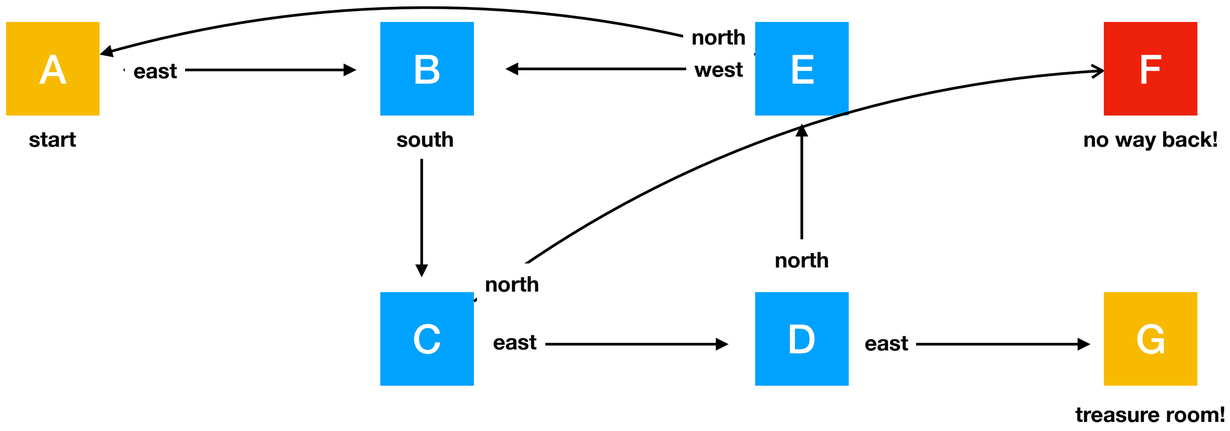
Prime numbers up to 100

…of course there are plenty of Python implementations on line, but try to come up with your implementation!

https://goo.gl/KAiXUN

Treasure Hunt

- Develop an OO code to navigate the following maze and play the game described below.



The player starts in cell A and can move in the directions indicated.
The game ends when the player reaches the treasure room G,
or ends in room F and dies.

I don’t think you will find this one online… ^_^

Book Catalogue

- Goodreads is an app that maintains three lists: books you have read; books you are reading; and books you want to read.

- Develop an OO code as close as possible to Goodreads: ideally you should be able to insert/remove items into any of the three lists and print any of the lists. Once more, use test-driven development.

The Zen of Python

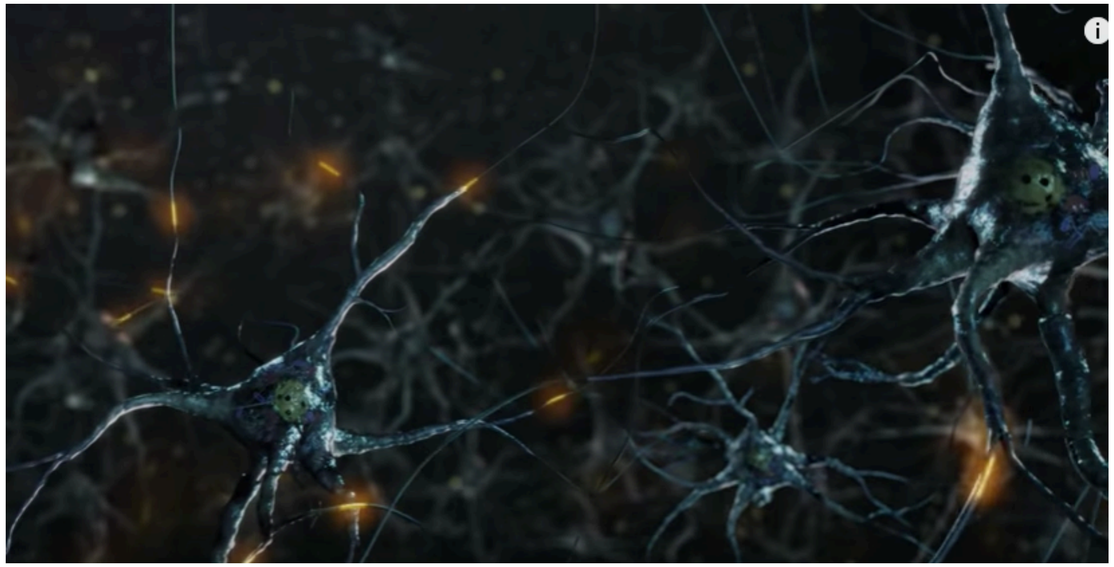
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Tim Peters

References

My favourite YouTube Python Course:

https://goo.gl/SFPPw6



python™

University of Edinburgh