“Practical IoT for Business Schools”

“There are **two kinds of people**: those who understand technology and those who don’t.

People who understand technology can design and control the very structure of the world around them. People who don’t understand it are controlled by those who do”

Mattan Griffel (director at Columbia Business School)

"IoT – Hardware aspects"

"Software does not exist ***per se***; it may be instantiated statically in the memory or dynamically during the execution on hardware processors"

"Real Men Have Fabs" Jerry Sanders, AMD founder

IoT – global picture

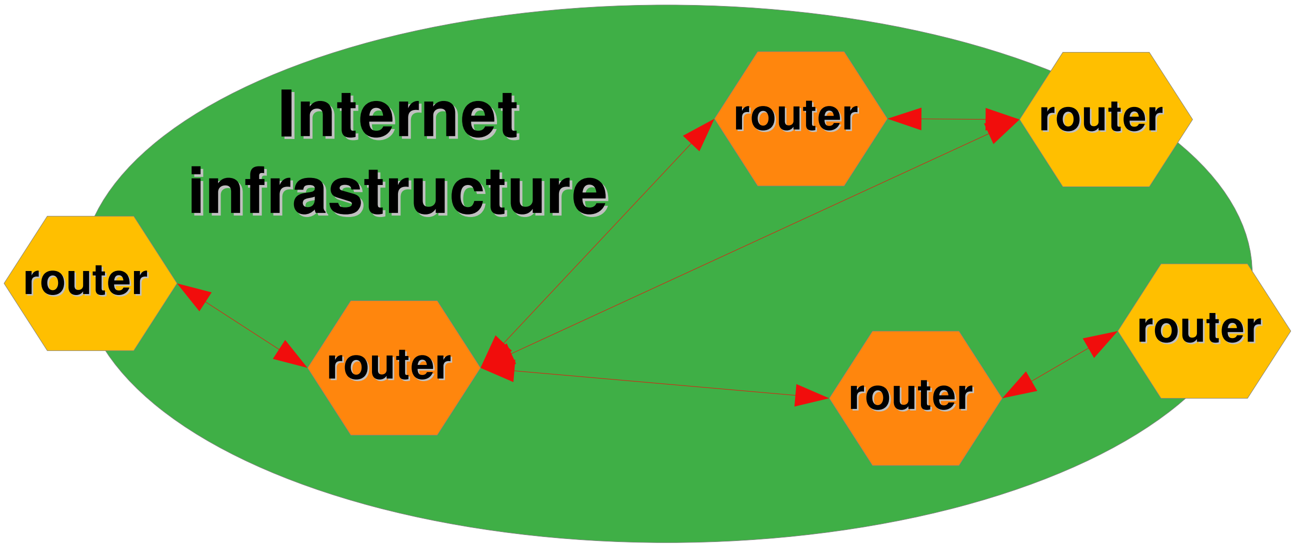
- **IoT – Internet of Things**
- **Things**
- **Internet**

Things: Embedded Software/Hardware

Internet: Communication means

Terminology, terminology, terminology , ..

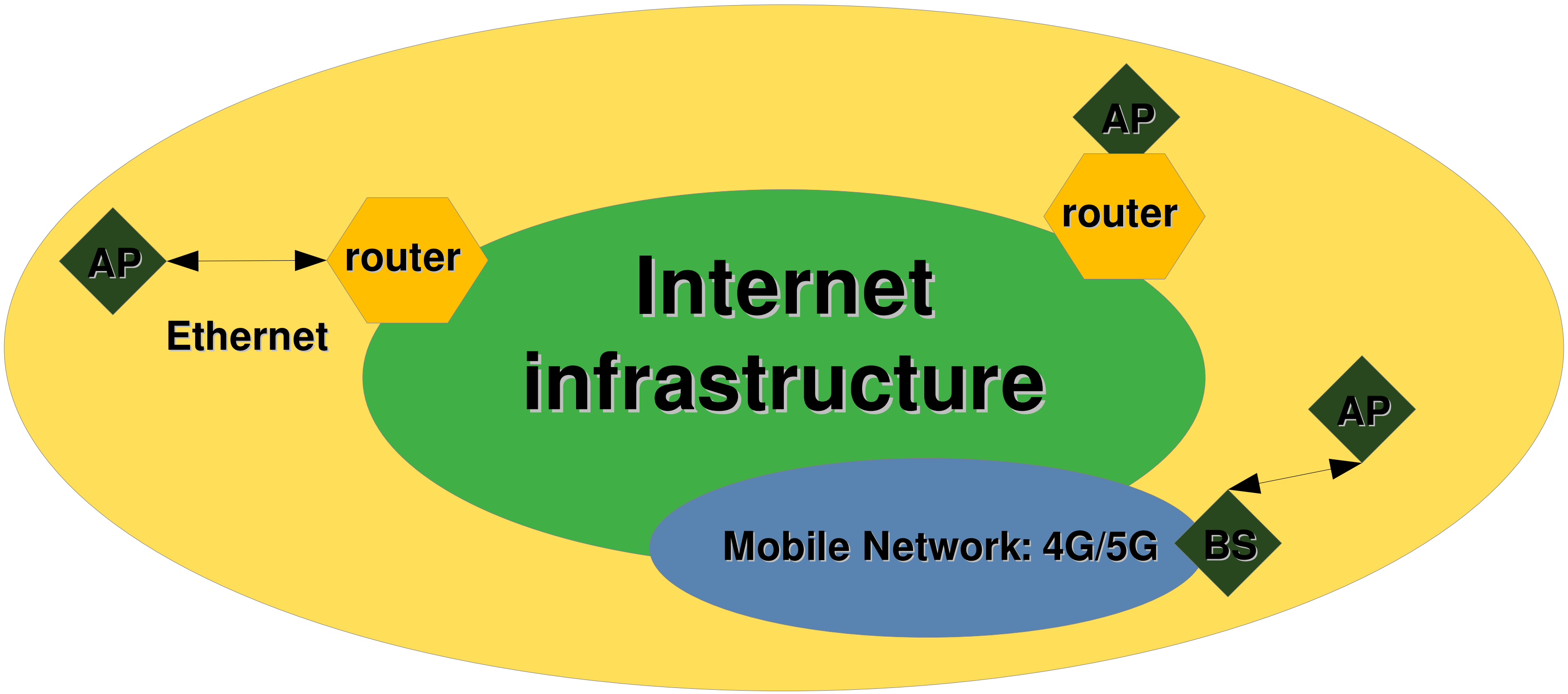
Internet Infrastructure



Router is internal device – IP packets: Packets Per Second, Packet Loss, ..

Long distance links – fiber : Bits Per Second – 10^6, 10^9, 10^{12}

Internet – Access Points

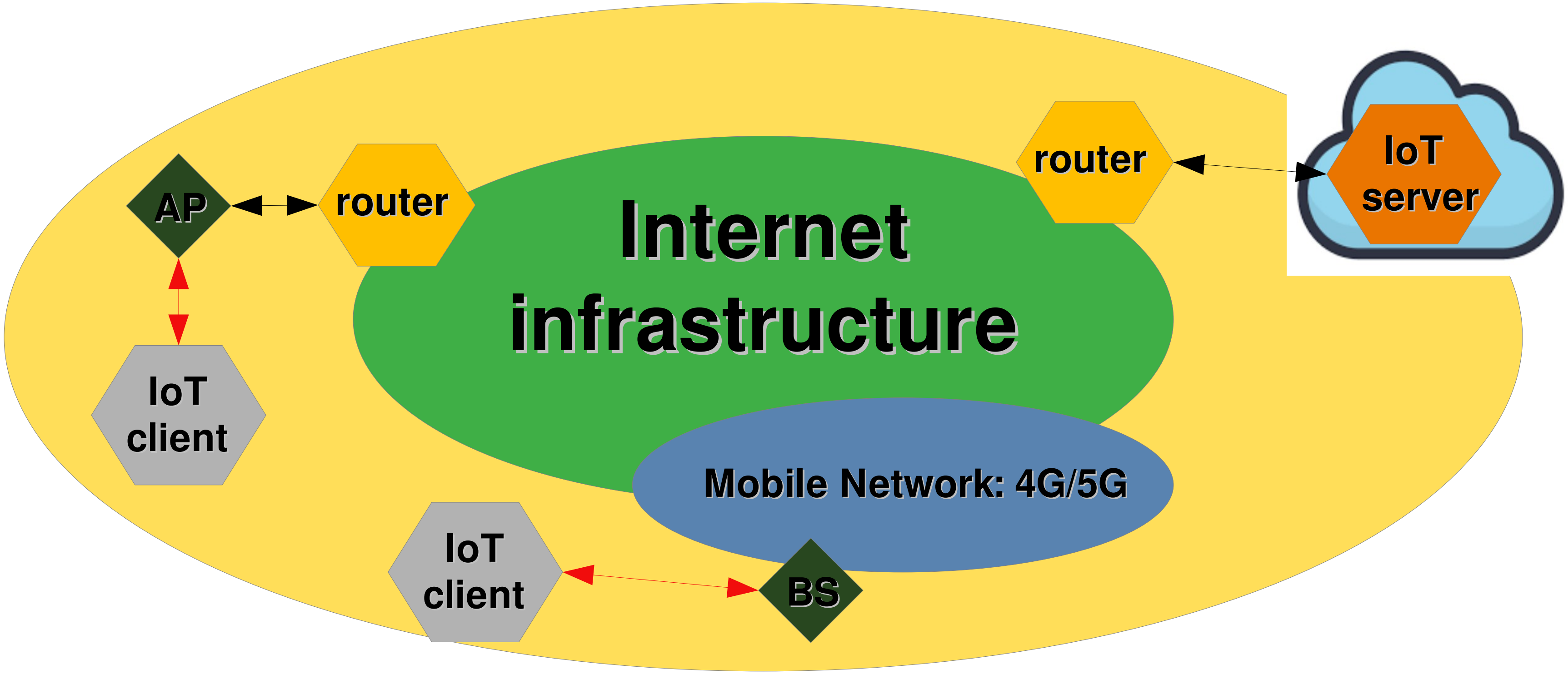


Ethernet – is local wired access to Internet for device, AP, switch, ..

AP – Access Point is a wireless entry (to Internet) for device (WiFi)

BS – Base Station is a wireless entry (to Internet) for device, AP

IoT – Clients and Servers

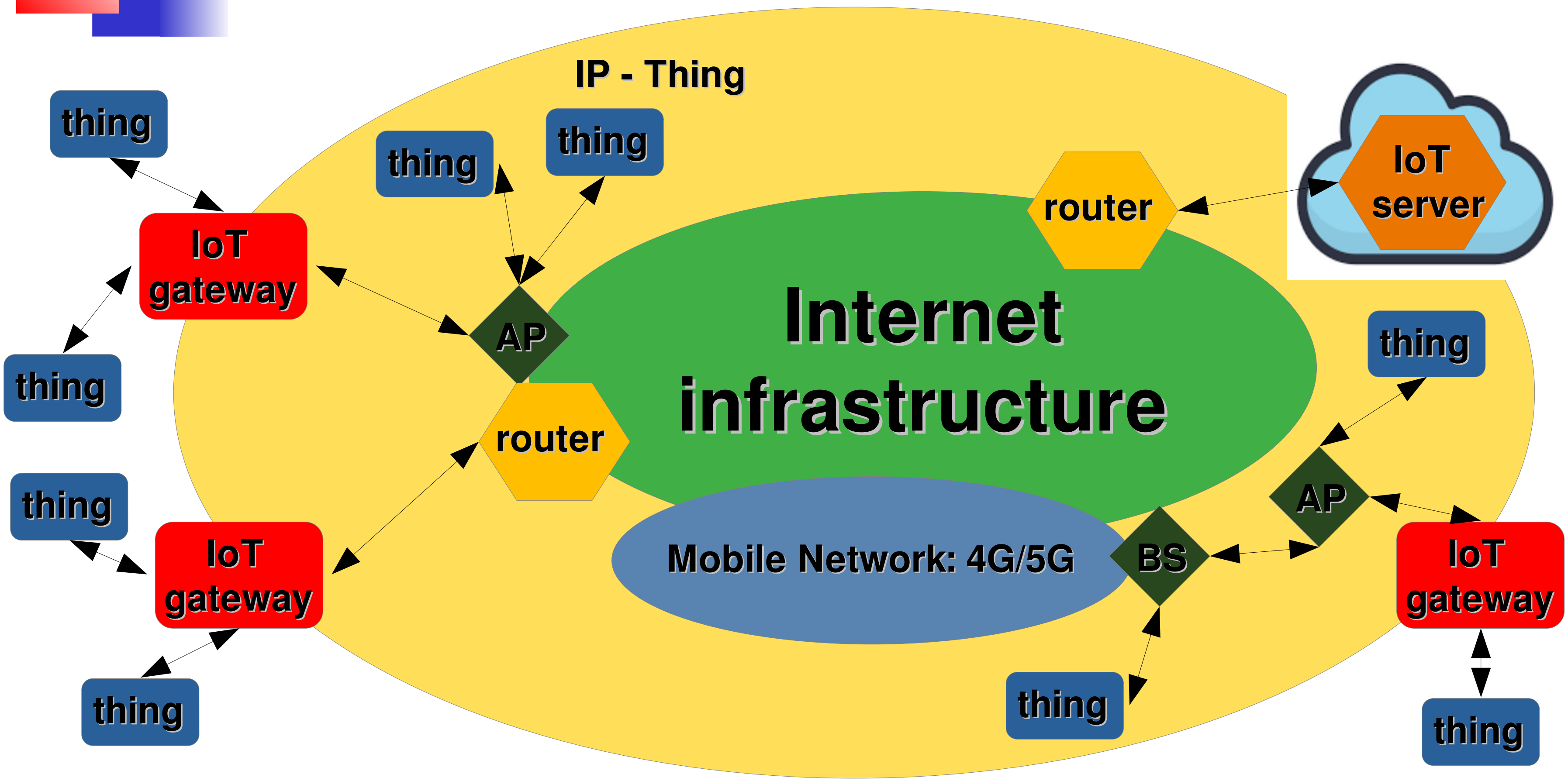


IoT Server and IoT Client are external devices

Client : PC, laptop, tablet, smartphone, IoT device, AIoT device, ..

Server : PC, SBC, HPC with data center, HPC with AI center, (Cloud: UP,DOWN)

IP Things and Non-IP Things

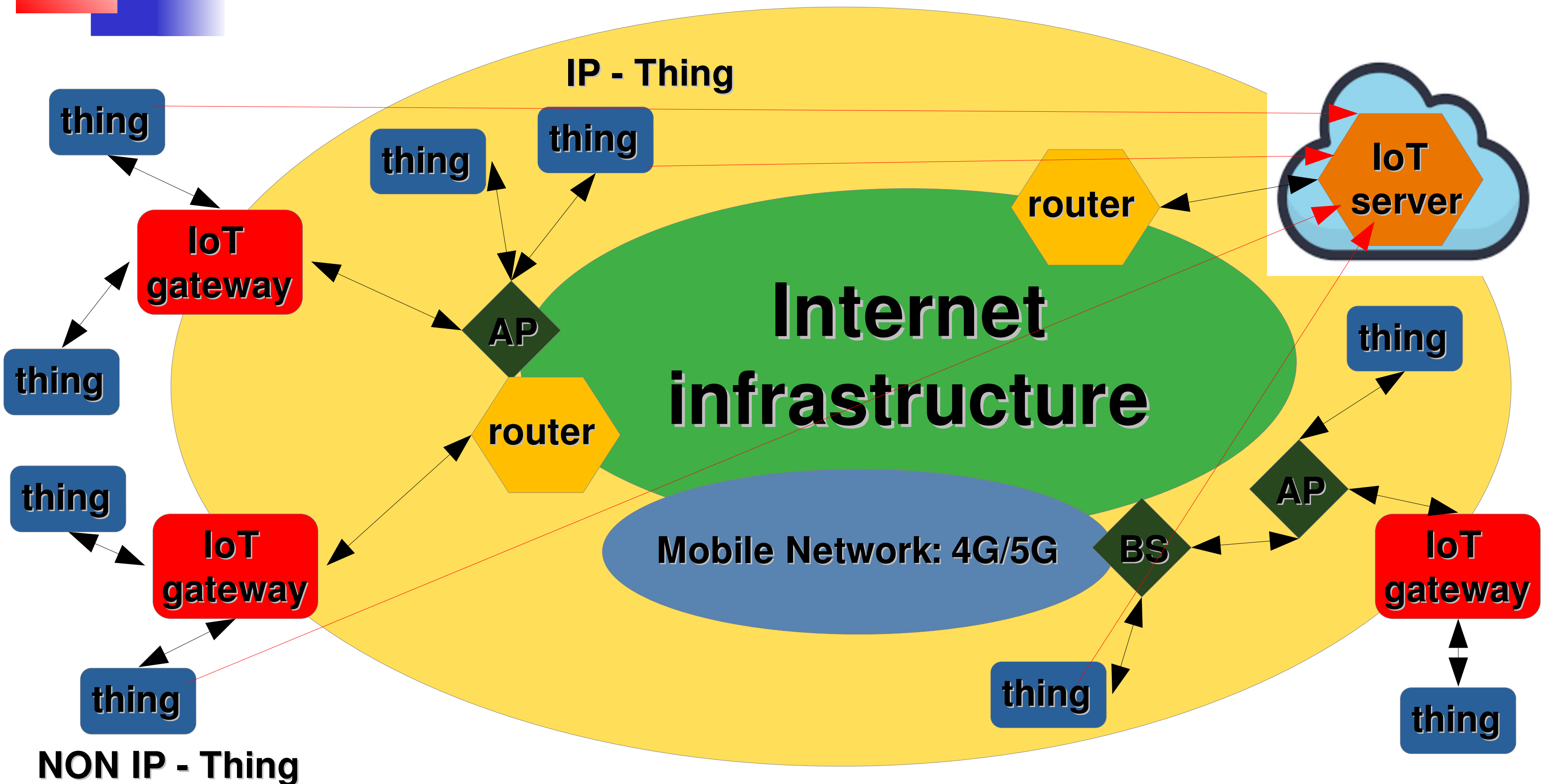


NON IP - Thing

Thing is a Terminal device

IoT gateway is an intermediate device

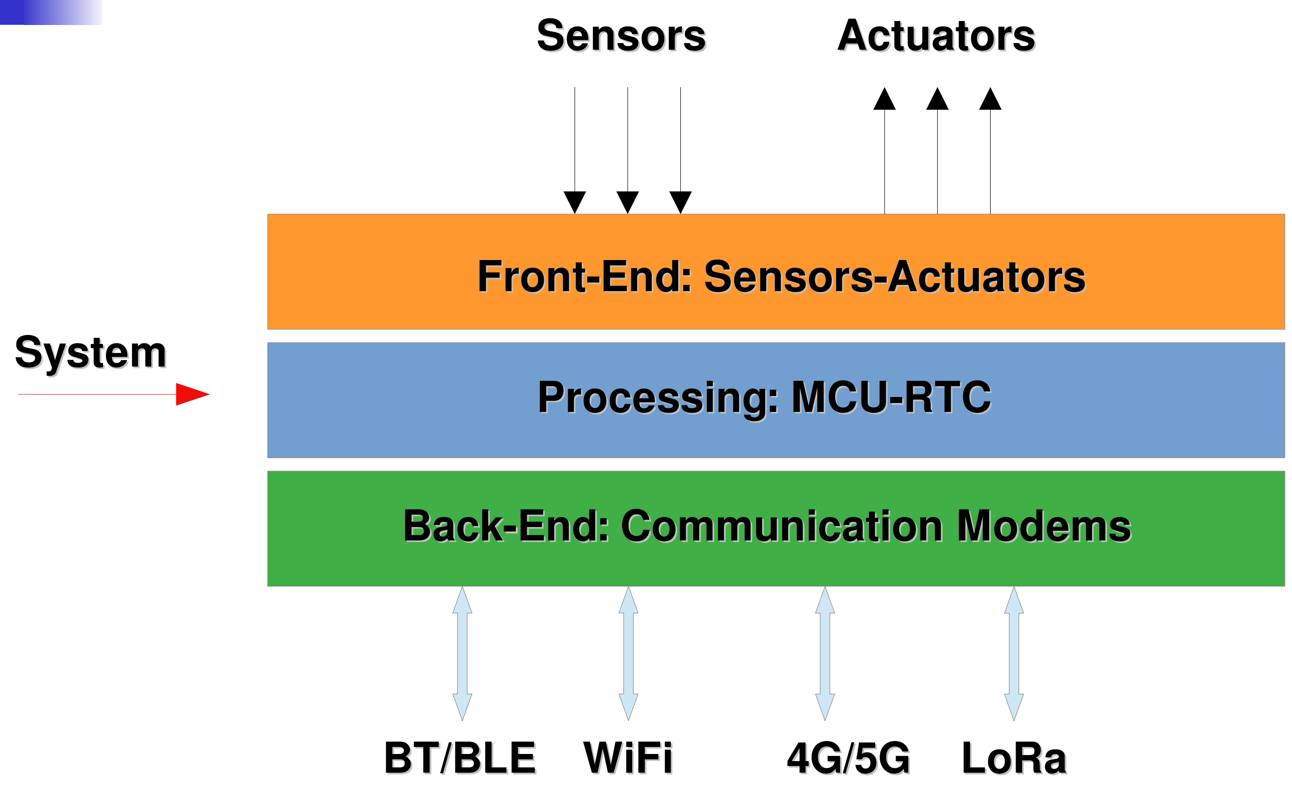
Routes and Examples - Discussion



Routes

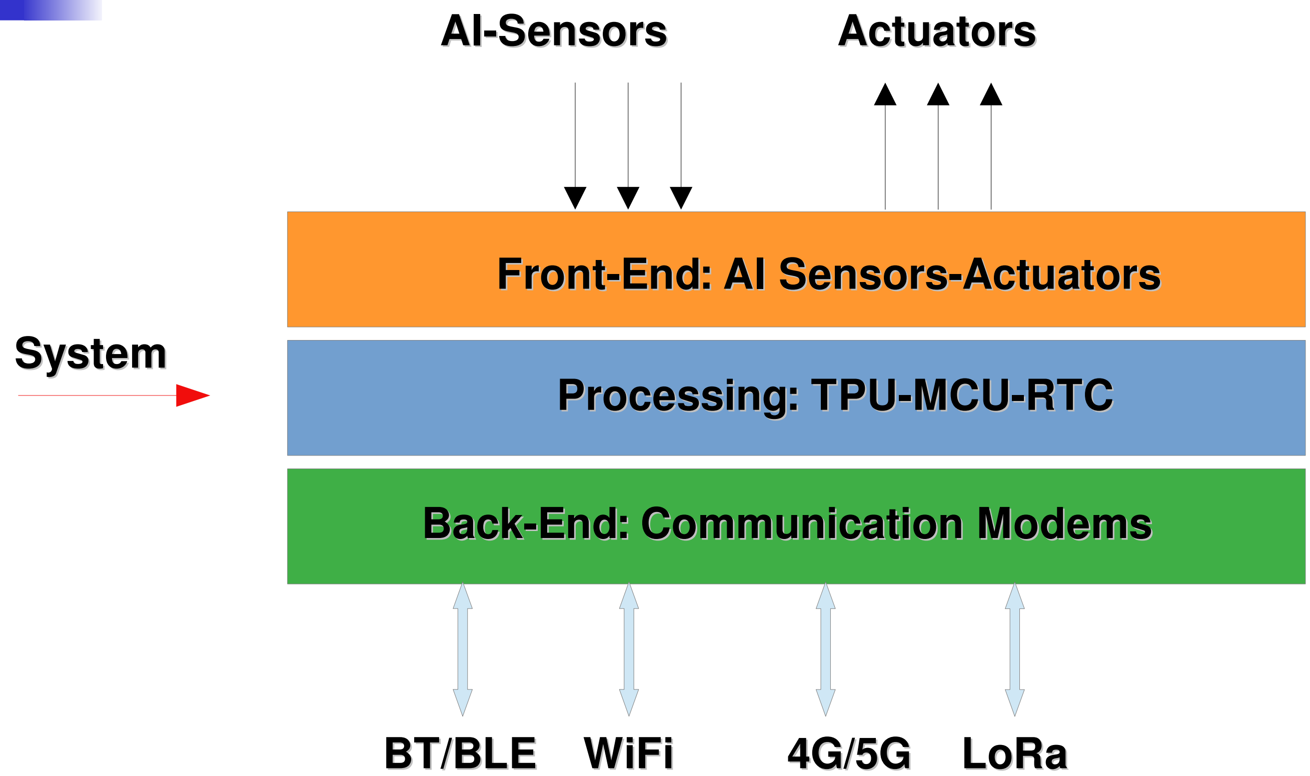
Examples, examples, examples - Discussion

Simple and Intelligent Things



Simple Thing – basic processing of physical data and display and activation of physical devices

Simple and Intelligent Things



Intelligent Thing – AI processing of physical data and display and activation of physical devices

Real example – System on Chip

ESP32 is a series of **low-cost**, **low-power system on a chip - SoC** micro-controllers with **integrated Wi-Fi** and dual-mode **Bluetooth**. The ESP32 series employs either a **Tensilica Xtensa LX6** microprocessor in both dual-core and single-core variations, **Xtensa LX7** dual-core microprocessor or a single-core **RISC-V** microprocessor and includes built-in antenna switches, RF balun, power amplifier, low-noise receive amplifier, filters, and power-management modules.

ESP32 is created and developed by **Espressif Systems**, a **Shanghai**-based Chinese company, and is manufactured by **TSMC** using their **40 nm** process

low-cost	**Wi-Fi**	**Xtensa LX6/LX7**
low-power	**BT-BLE**	**RISC-V**

Real example – System on Chip

Low-cost < $5

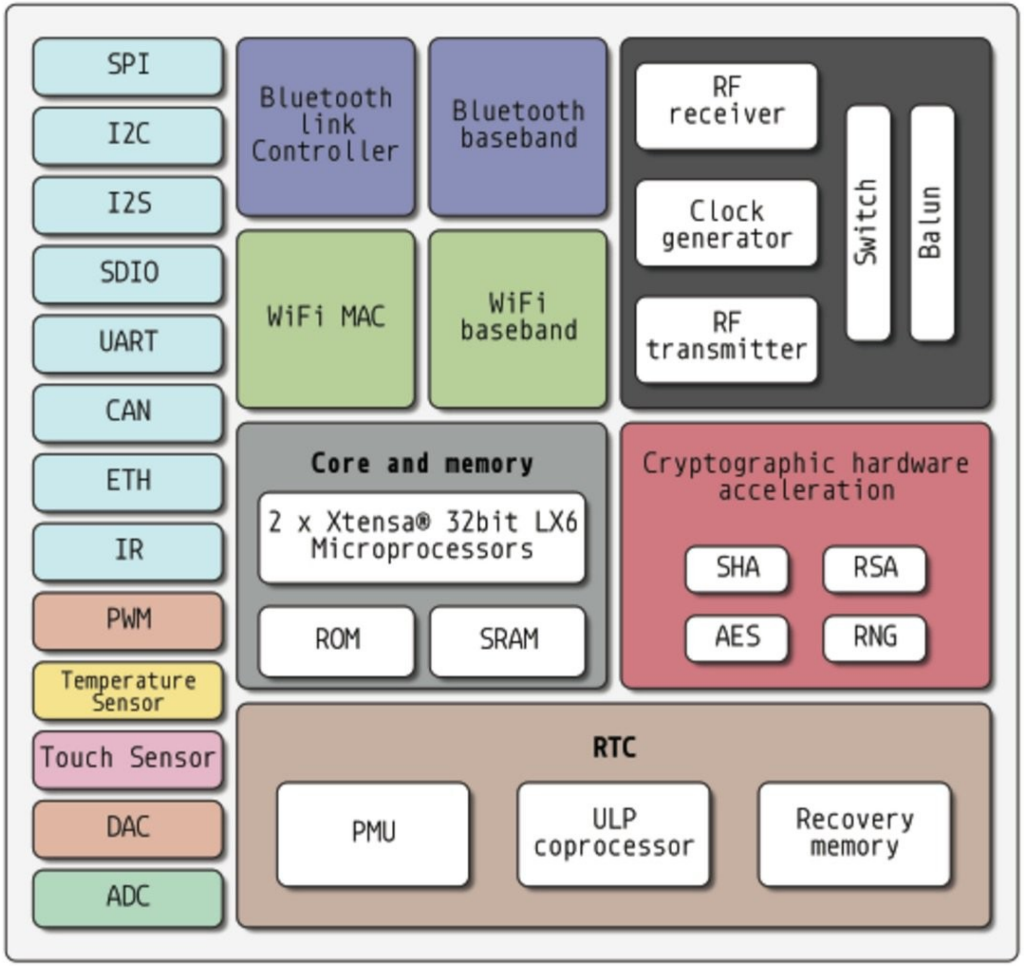
Low-power < 1-100mA (5V)

Power (W) = Current(A) * Voltage(V)

rich-interfaces

Wi-Fi

BT-BLE



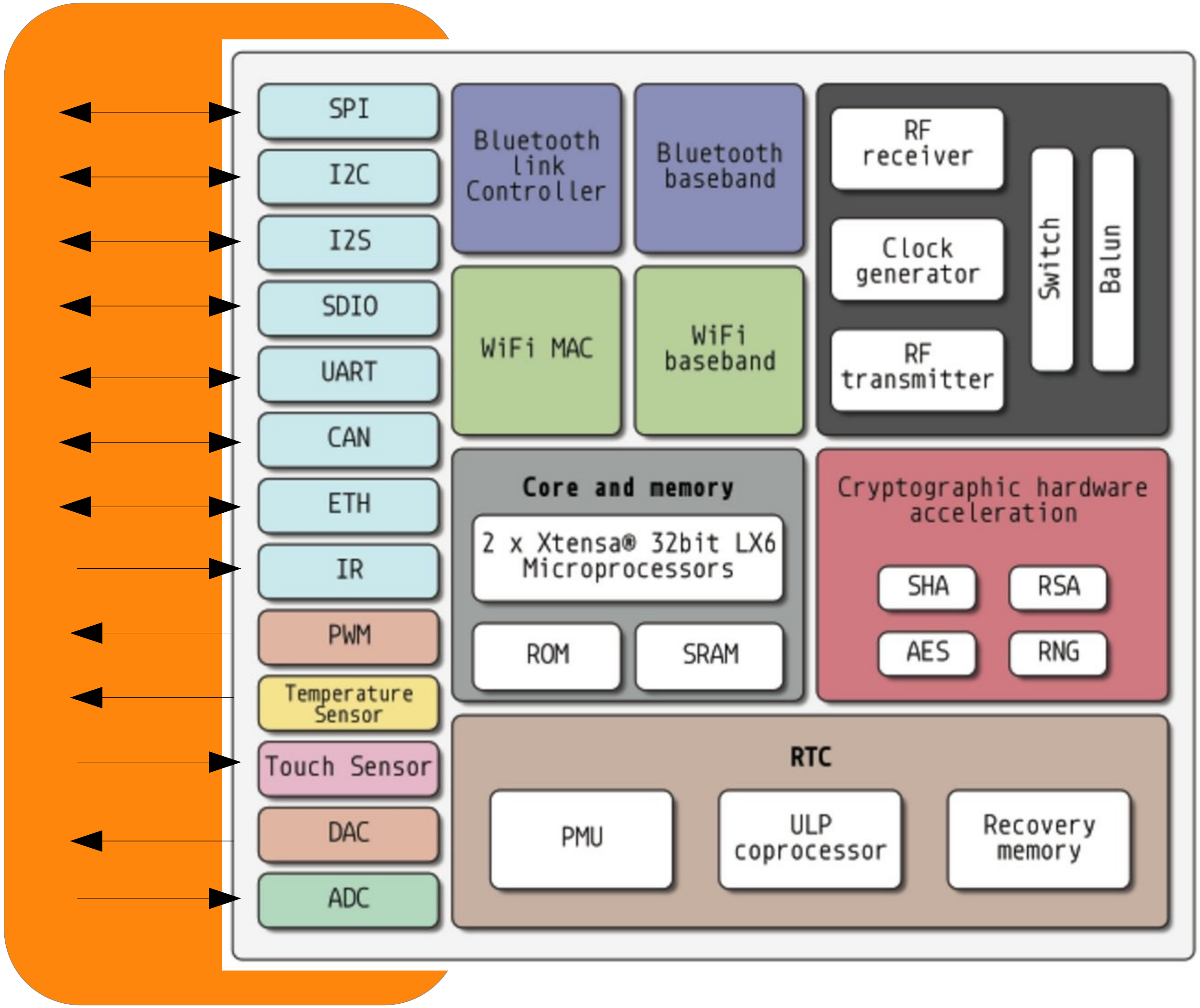
Real example – System on Chip

interfaces
to sensors
and
actuators

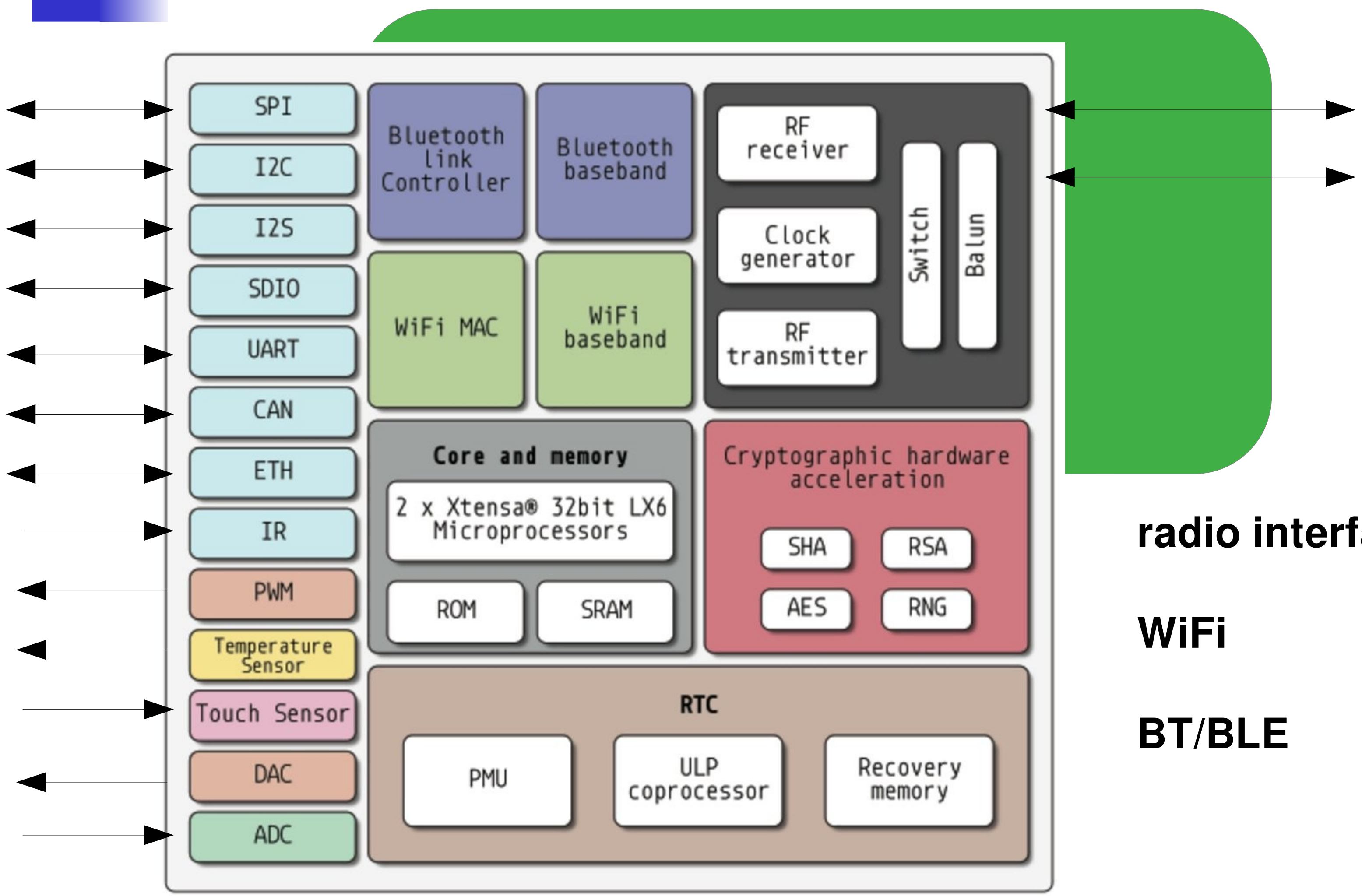
UART

I2C

SPI



Real example – System on Chip

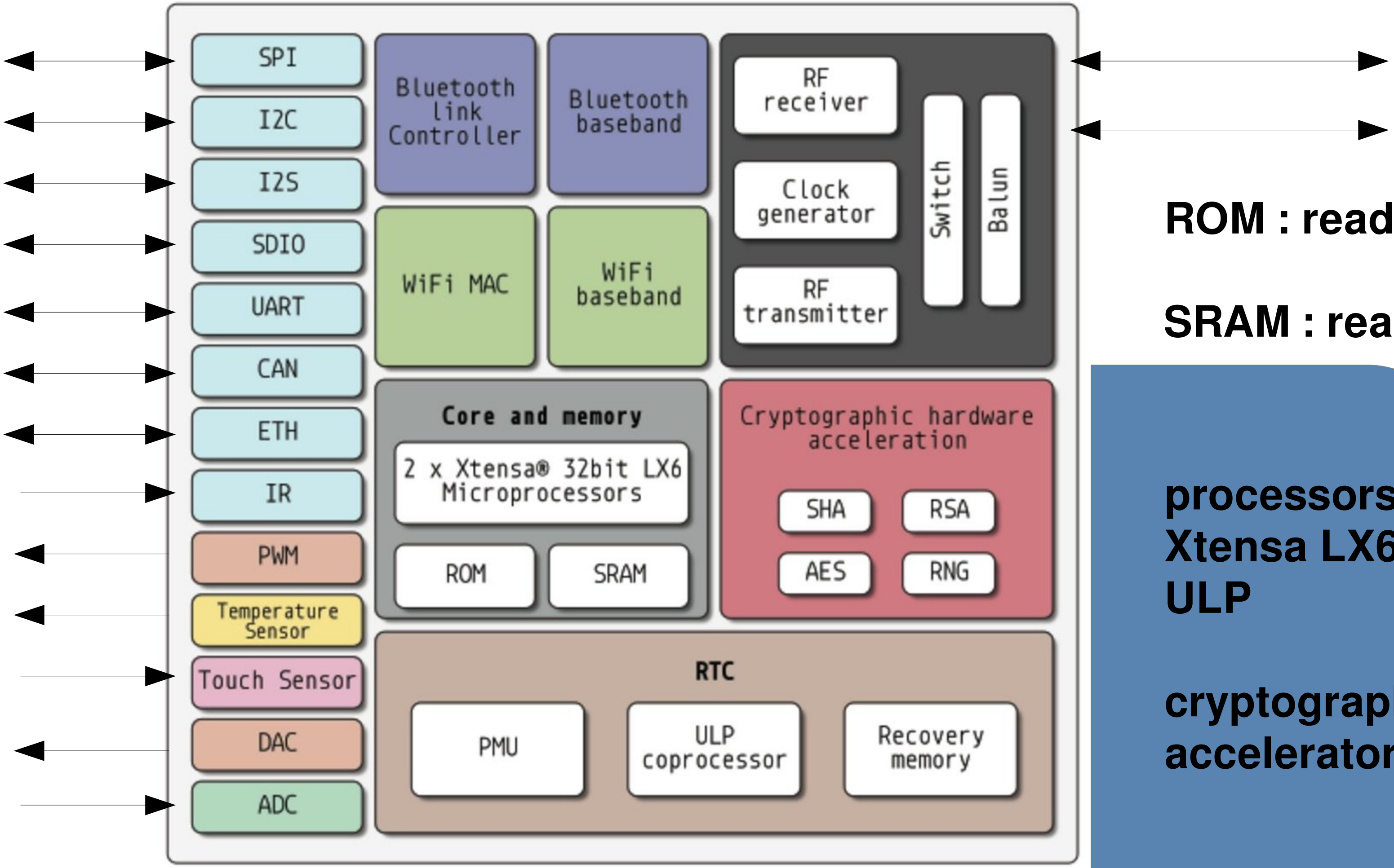


radio interfaces

WiFi

BT/BLE

Real example – System on Chip



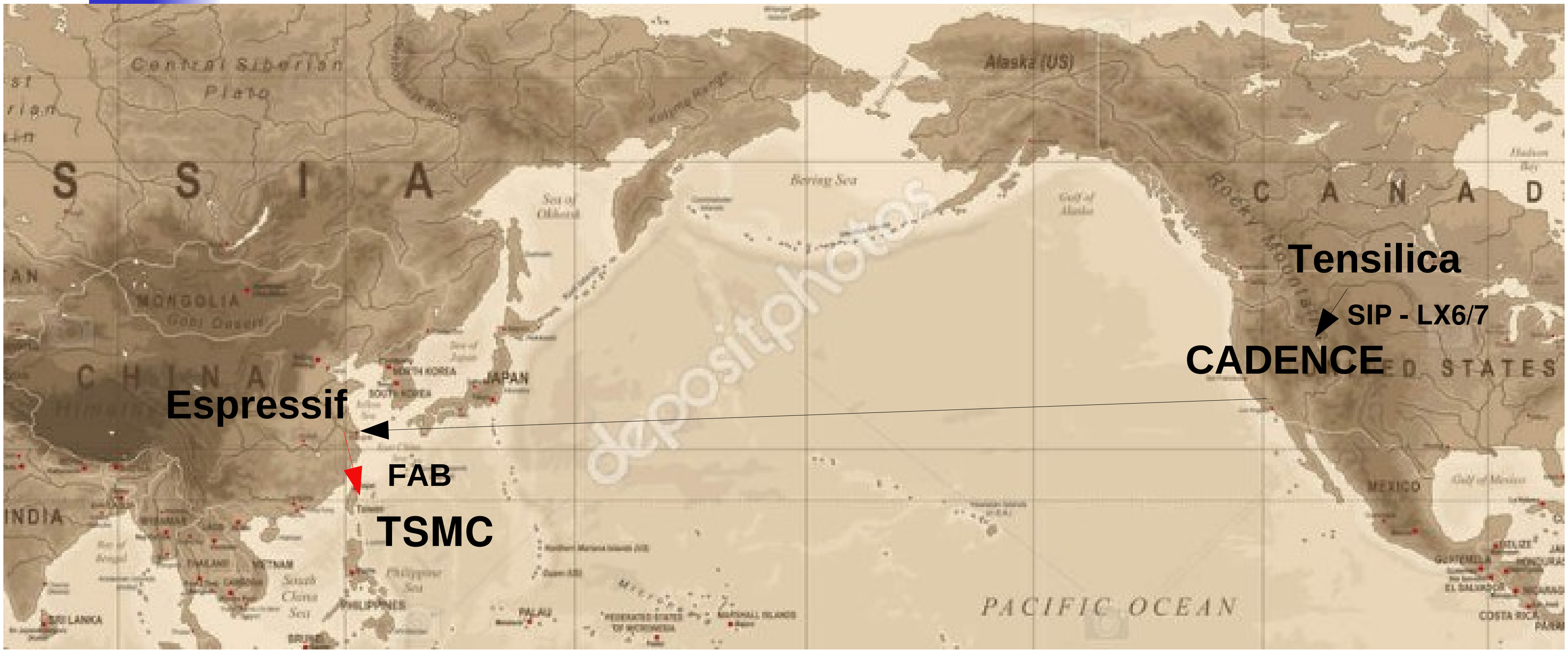
ROM : read only

SRAM : read_write

processors
Xtensa LX6
ULP

cryptographic accelerators

IoT – technology transfer aspects



Cadence Design Systems, Inc., headquartered in San Jose, California, is an American multinational computational software company, founded in 1988. The company produces software, hardware and silicon structures for designing integrated circuits and systems on chips (SoCs) .

IoT – economic aspects

Tensilica is known for its customizable **Xtensa** (LX6/7) microprocessor core.

Tensilica was a company based in Silicon Valley in the **semiconductor intellectual property** (**SIP**) core **business**. It is now a part of Cadence Design Systems.
On March 11, **2013**, Cadence Design Systems bought **Tensilica** for approximately **$380 million in cash**.

Espressif bought **eXtensa LX6/7** license (**SIP**) to design ESP32 SoCs. It went public on Shanghai Stock Exchange in **2019** with **2 billion US dollars**.

Remark: European and US investors were not allowed to buy the shares !

SIP - Silicon Intellectual Propriety

What is SIP

In electronic design, a **semiconductor intellectual property** core (SIP core), IP core, or IP block is a **reusable unit of logic**, cell, or integrated circuit layout design that is the intellectual property of one party.

IP cores can be licensed to another party or owned and used by a single party. The term comes from the **licensing of the patent** or source code copyright that exists in the design.

There are:

- → **Soft cores**
- → **Hard cores**

Remark:
Thing about SIP cores as of “**genetic code**” for the production of digital circuits and systems – SoC.

IoT – SIP: hard and soft cores

Soft cores

IP cores are commonly offered as synthesizable RTL in a **hardware description language** such as Verilog or VHDL. These are analogous to low-level languages such as C in the field of computer programming. IP cores delivered to chip designers as RTL permit chip designers to modify designs at the functional level, though many IP vendors offer no warranty or support for modified design

Hard cores

Hard cores (or hard macros) are analog or digital IP cores whose function cannot be significantly modified by chip designers. These are generally defined as a lower-level physical description that is **specific to a particular process technology.**

Hard cores delivered for one foundry's process cannot be easily ported to a different process or foundry.

SIP and licenses

Licensed functionality

Many of the best known IP cores are **soft microprocessor designs**.

Their instruction sets vary from small 8-bit processors, to 32-bit and 64-bit processors such as the **ESP32 LX6/7**, **ARM** architectures or **RISC-V** architectures.

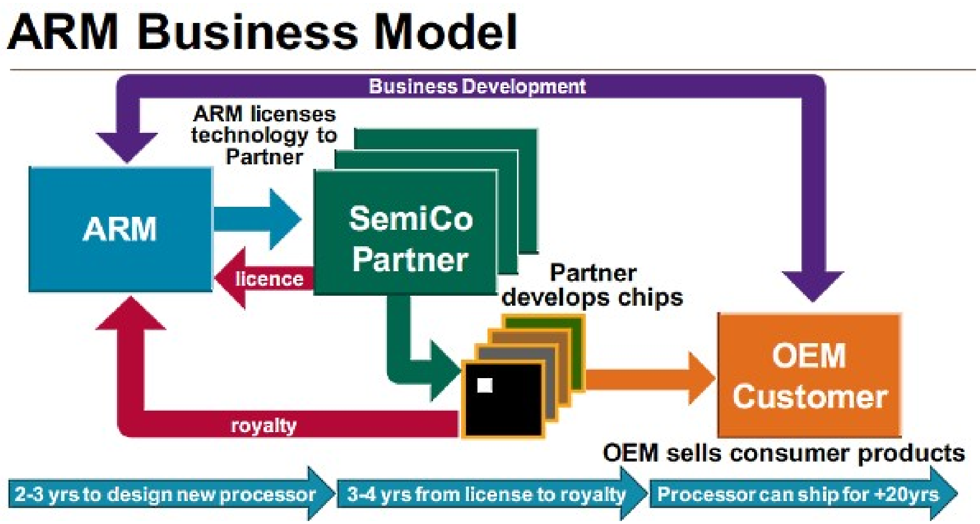
Such processors form the "**brains**" of many **embedded and IoT systems**.

x86 leaders **Intel** and **AMD** heavily protect their processor designs' intellectual property and **don't use this business model** for their x86-64 lines of microprocessors.

ARM business model

ARM's revenue comes **entirely from IP licensing**. It's up to ARM's licensees/partners/customers to actually build and sell the chip. ARM's revenue structure is understandably very different than what we're used to.
There are **two amounts** that all ARM licensees have to pay:

→ an **upfront license fee**, and
→ a **royalty**.



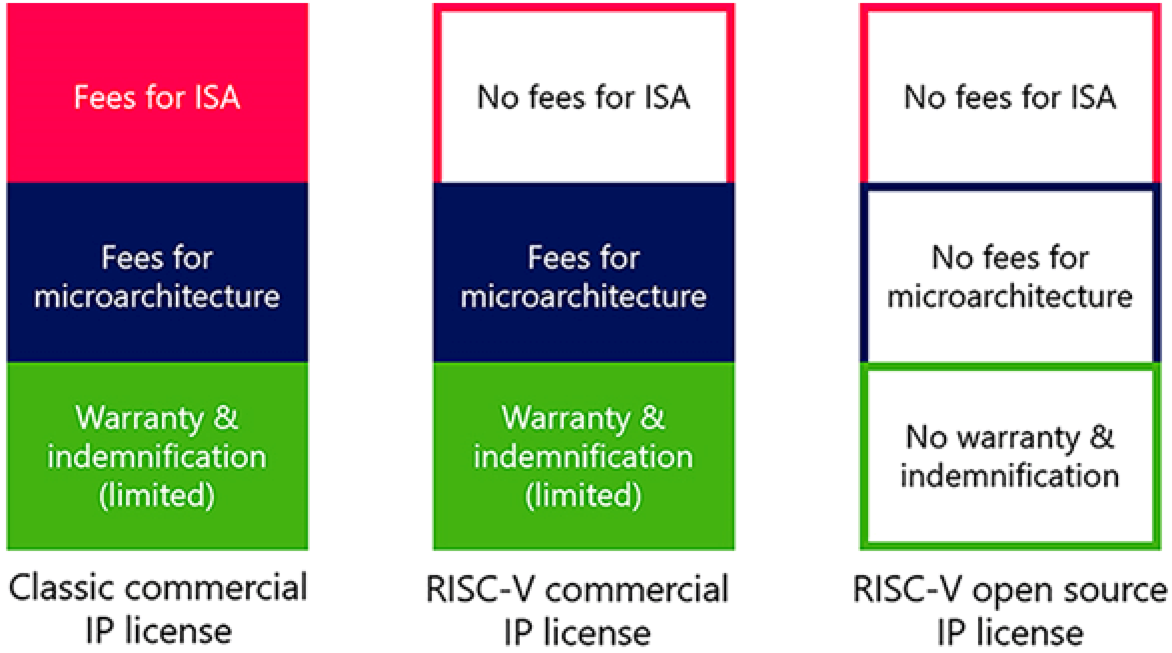
The licensing fees vary between an estimated **$1 million to 10 million**. The royalty is usually **1 to 2% of the selling price of the chip**. Licensing enables ARM to scale the business efficiently.

ARM vs RISC-V business model

ARM’s revenue comes **entirely from IP licensing including ISA**.

RISC-V is a **standard and open architecture** with no fees for **ISA** (**Instruction Set Architecture**)

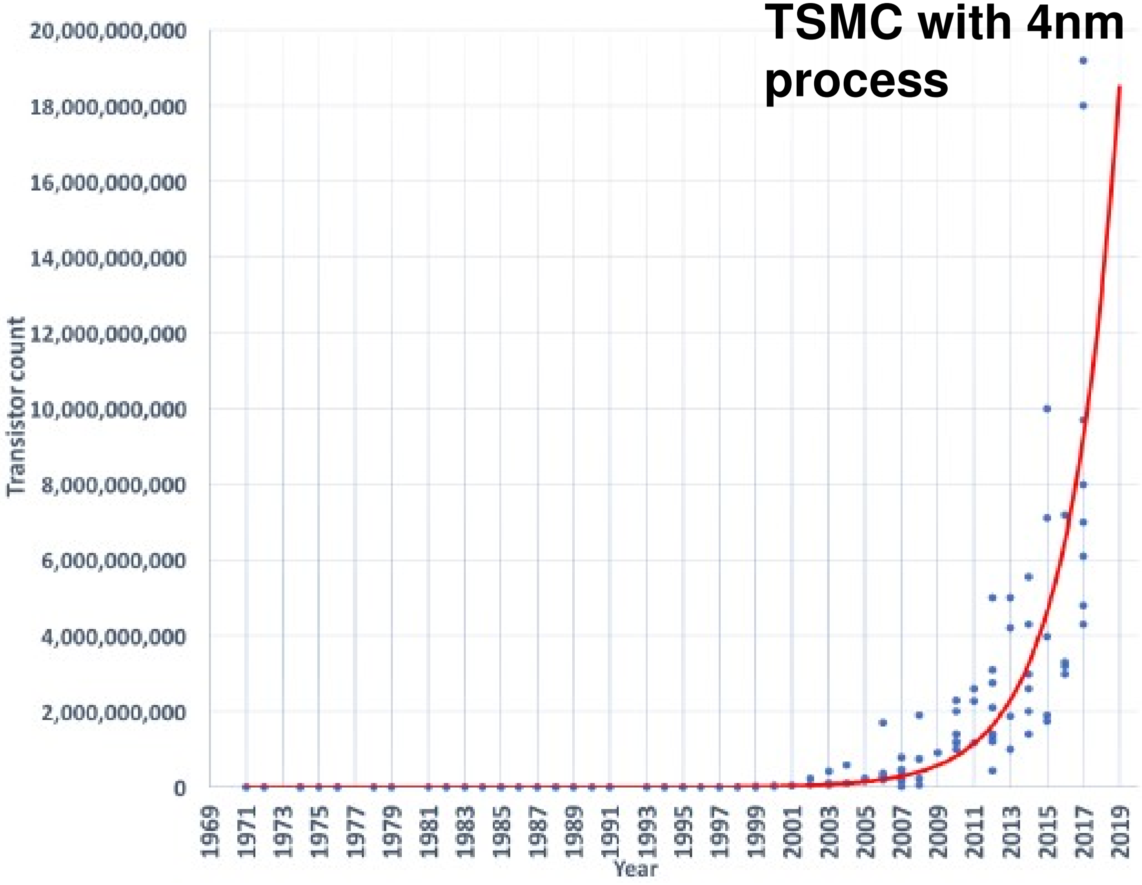
ARM



Moore's Law – digital driver

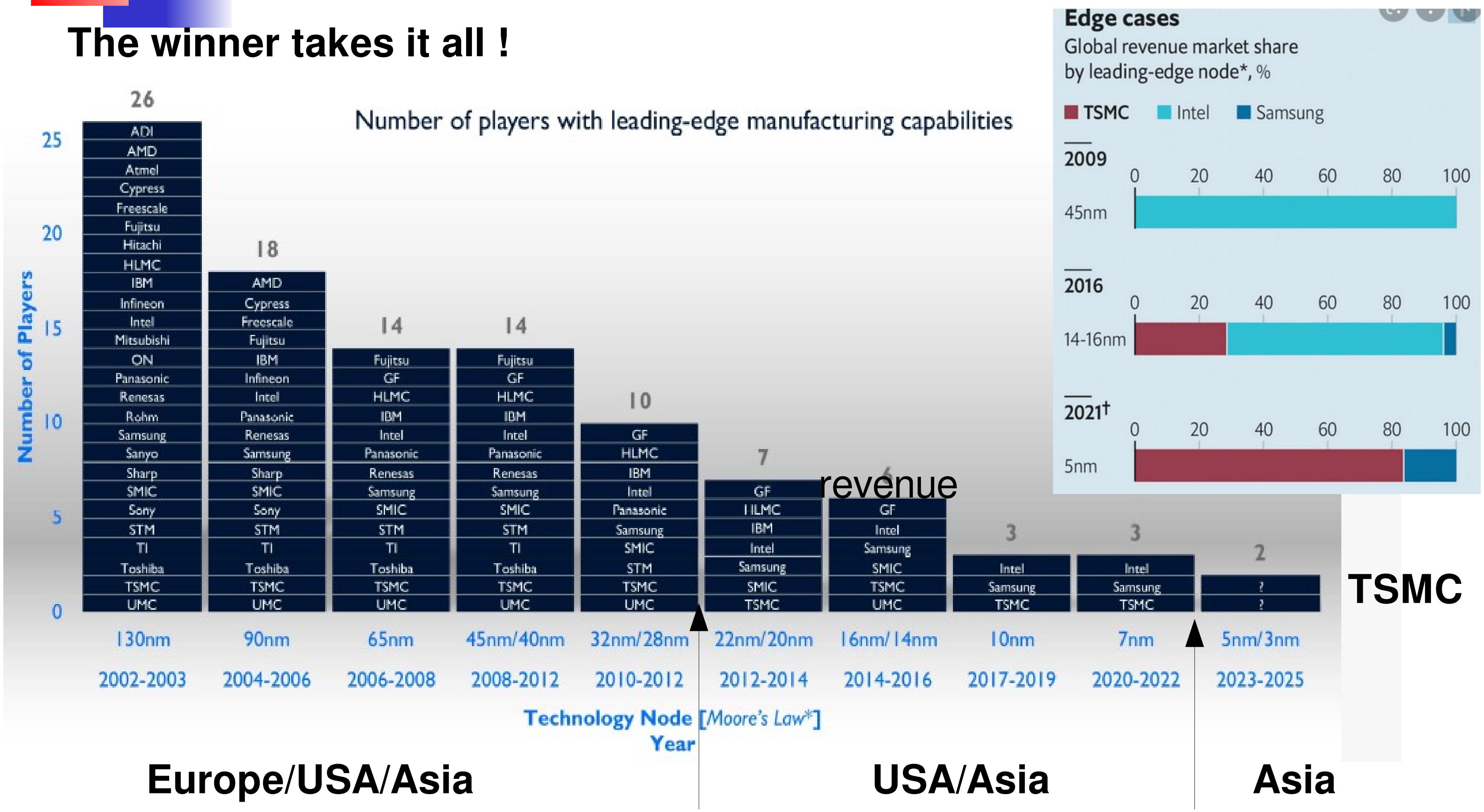
Moore's law is the observation that the **number of transistors** in a dense integrated circuit (IC) **doubles about every two years**. It is **linked to gains from experience in production**.

2022 – 55 billions – Apple – M2 – TSMC with 4nm process



High-end foundries - evolution

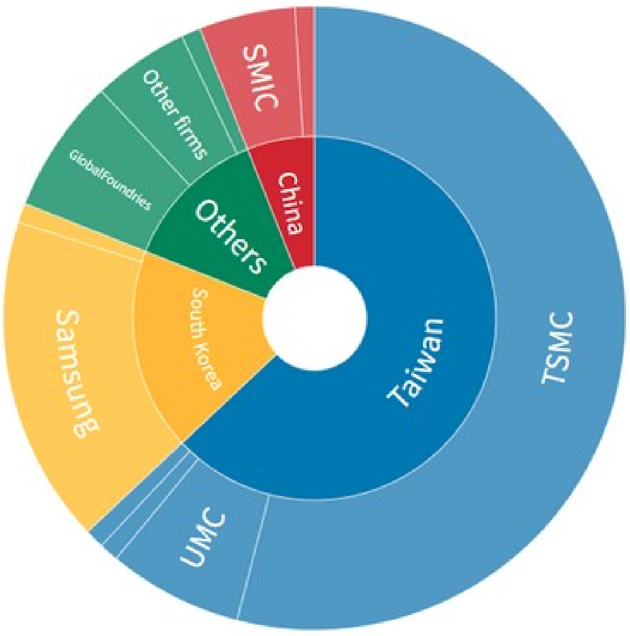
The winner takes it all !



And finally pure-foundries

Key contract manufacturers include (2021 - total foundry revenue ~100 billion):

→ **Taiwan Semiconductor Manufacturing Company (TSMC) Limited**,
→ Global Foundries,
→ United Microelectronics Corporation (UMC),
→ Semiconductor Manufacturing International Corporation (SMIC),
→ **Samsung Group**,
→ Dongbu HiTek, and
→ STMicroelectronics.



TSMC: The World's Most Important Company

TSMC is arguably the **world's most important company**.
(2021) Apple, which accounts for one-fifth of TSMC's revenue, told investors that sales of Macs and iPads would fall by some $3 billion because of supply constraints.

Few numbers:

→ A new foundry (4 in construction by TSMC) costs about **$15 billion** (more than a nuclear plant); TSMC - $44 billion CAPEX for 2022 for **3nm** and **2nm** nodes.
→ One (EUV) chip machine (**ASML**) costs up to **$250 million**. All these machines (production **60/year**) are already sold up to 2024 to TSMC and Samsung.
→ **1 operational second** of such a foundry costs **$200/second** – **100 jet fighters** in operational flight.

Summary

→ **IoT hardware is essential for the development of modern digital infrastructure**

→ **IoT hardware as well as the hardware of all modern digital systems is based on SoC**

→ **IoT SoC are designed by fabless companies using SIP**

→ **The high-end production is done in the silicon foundries such as TSMC**

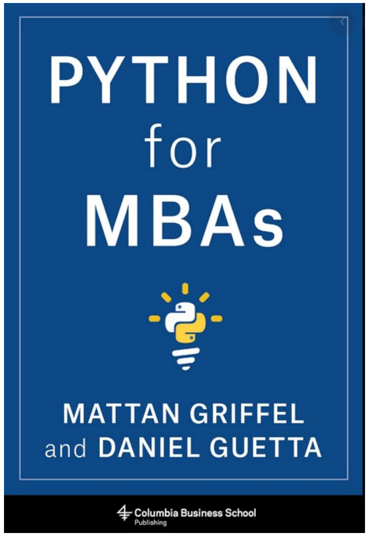
Remark: In Europe there is no high-end silicon foundries

"IoT – Software aspects"

"There are two kinds of people: those who understand technology and those who don't.

People who understand technology can design and control the very structure of the world around them. People who don't understand it are controlled by those who do"

Mattan Griffel (director at Columbia Business School)



IoT – software aspects

ESP32 SoCs are powerful micro-controllers.

Basically they operate under the control of **FreeRTOS**.

The programming may be carried out with:

- **C/C++** or
- **MicroPython**

C/C++ are **source languages** that must be **compiled** into **binary code** before the execution on the processor.

MicroPython (Python) is source language that is **interpretable**.After loading to the SoC memory the (Python-byte-code) may be directly (executed) interpreted. This solution requires an interpreter to be loaded and ready in the SoC **flash memory**.

IoT - Programming IDE

ESP32 SoCs programming is carried out via an **IDE** – Integrated Development Environment.

For C/C++ the IDE tools perform:

→ **Editing** of the source code
→ **Compilation** the source code to binary code
→ **Loading** (to flash memory)

For MicroPython the IDE tools perform:

→ **Editing** of the source code
→ **Loading** (to flash memory)

→ **C/C++: complete and efficient, 3 phases development cycle**

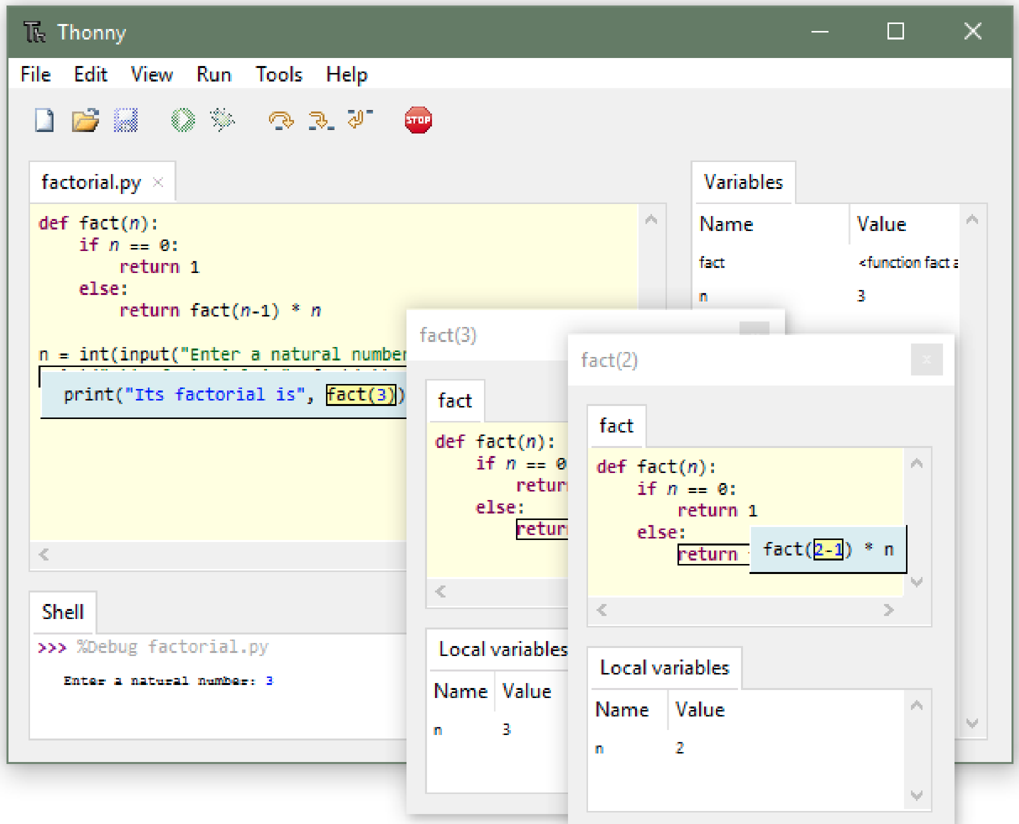
→ **MicroPython/Python – less efficient but easier to write and with 2 phases development cycle (processor independent)**

Thonny IDE – starting with Python

Thonny

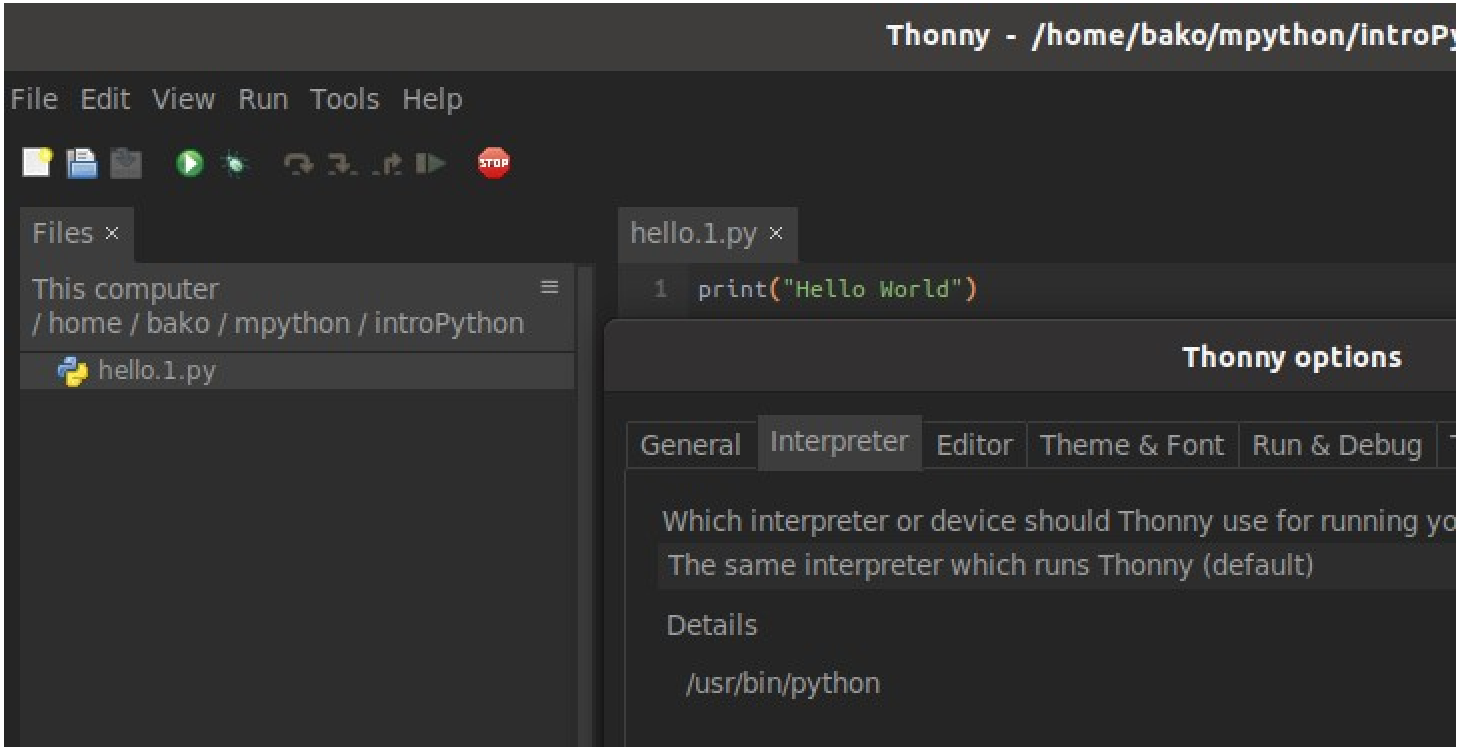
Python IDE for beginners

Download version 3.3.13 for
Windows • Mac • Linux



Python – interpreter

Choose the Python interpreter – the same as Thonny



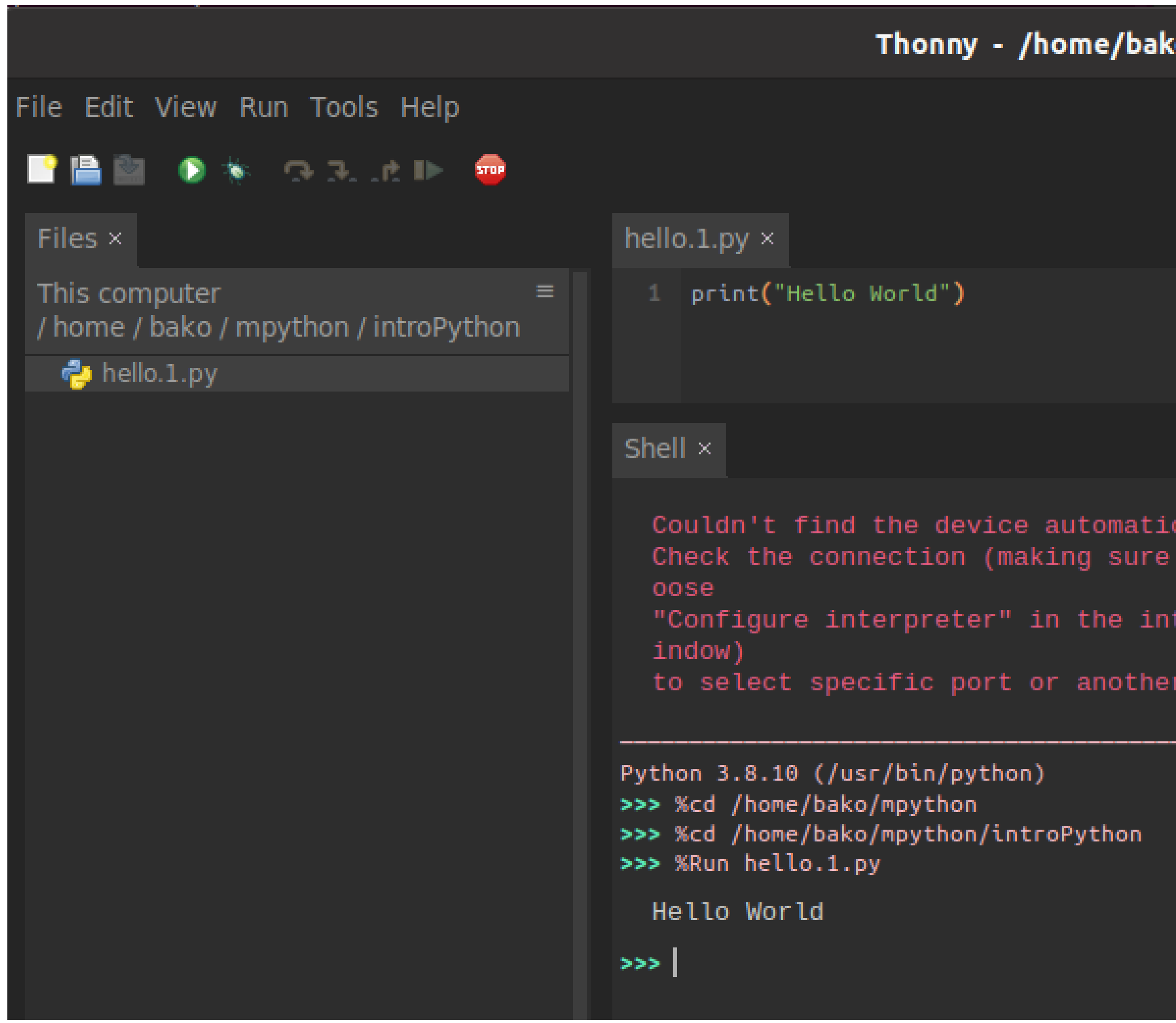
Python – first code

3 windows:

→ Files

→ Editor

→ Shell - terminal

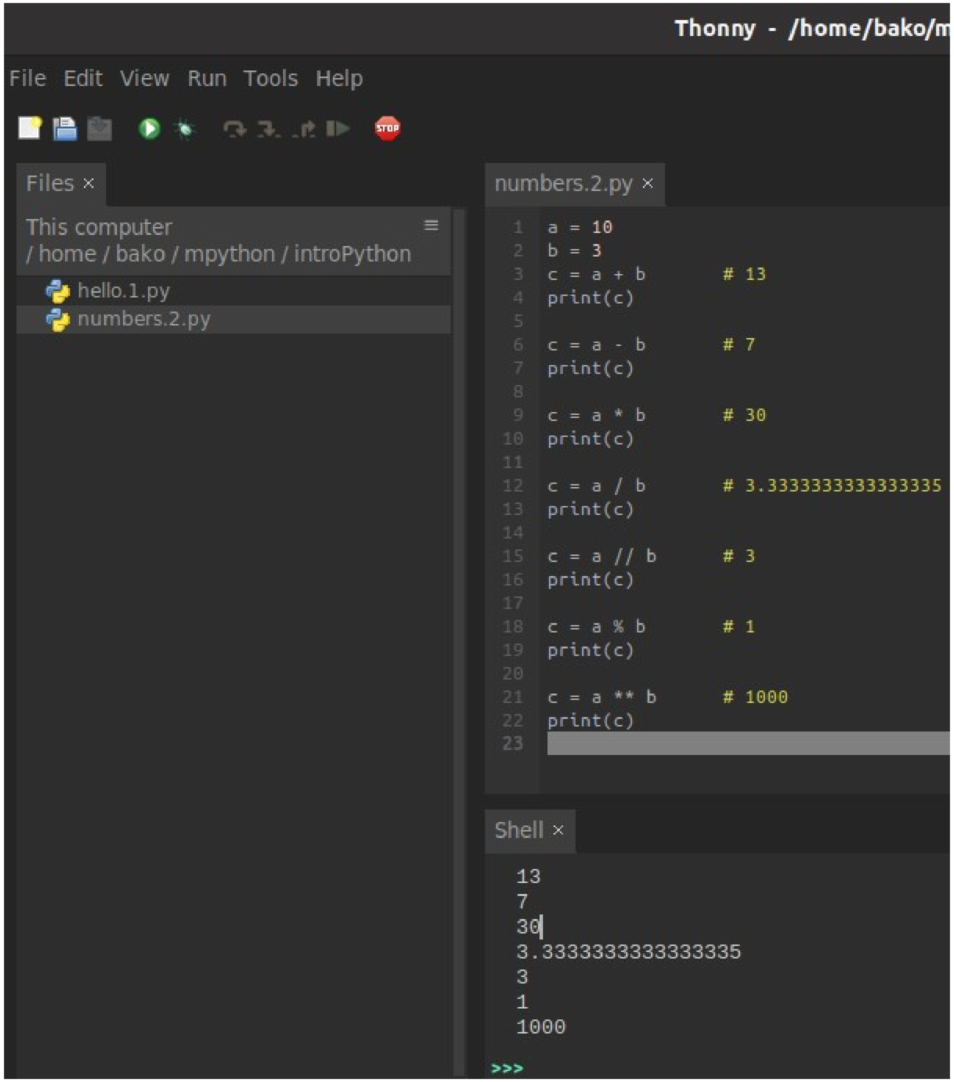


Python – numbers

Python provides for different types of numbers.

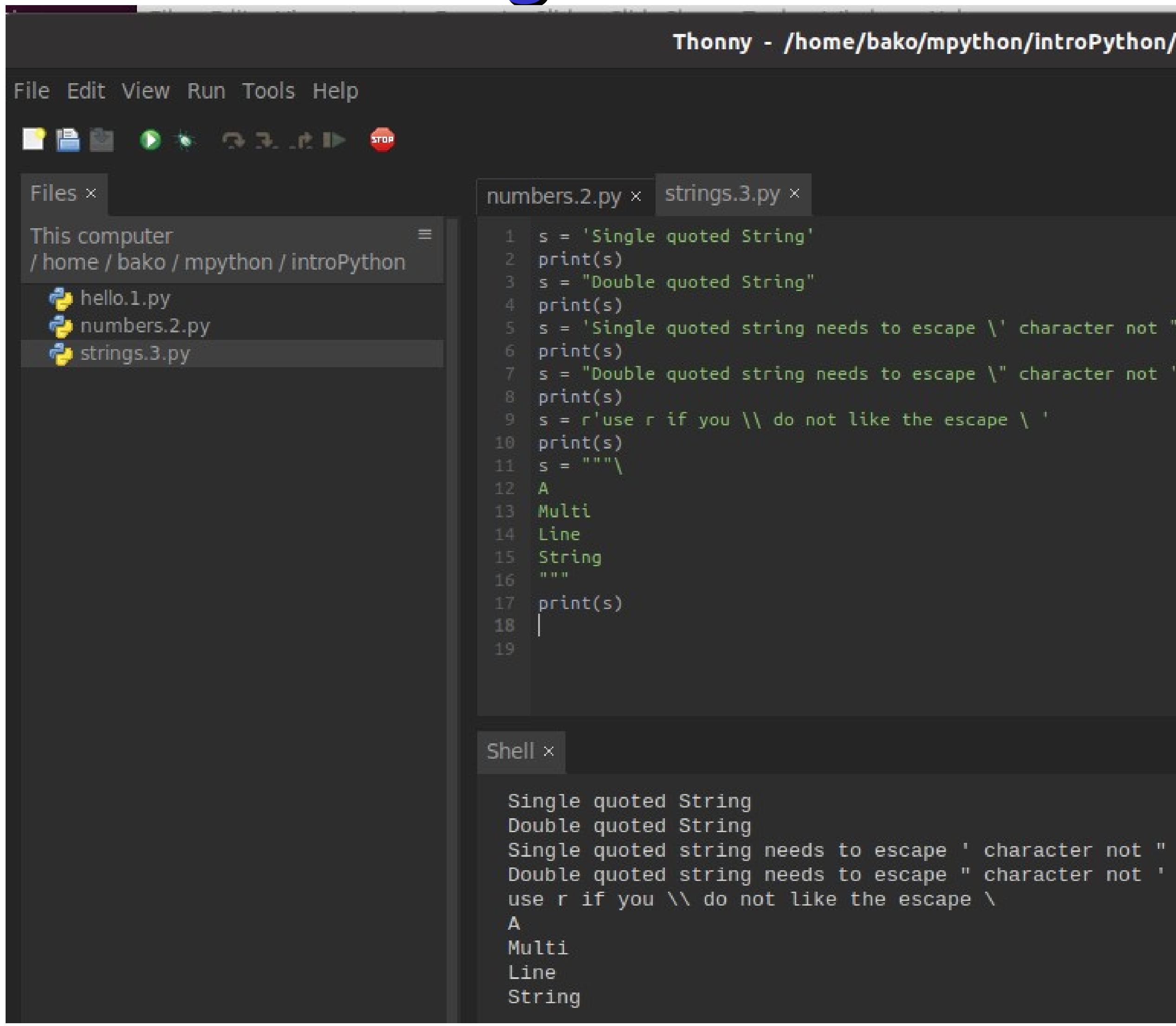
We have **integers**, **floats**, ..

We have arithmetical operators:
+, −, *, / , // , %, **

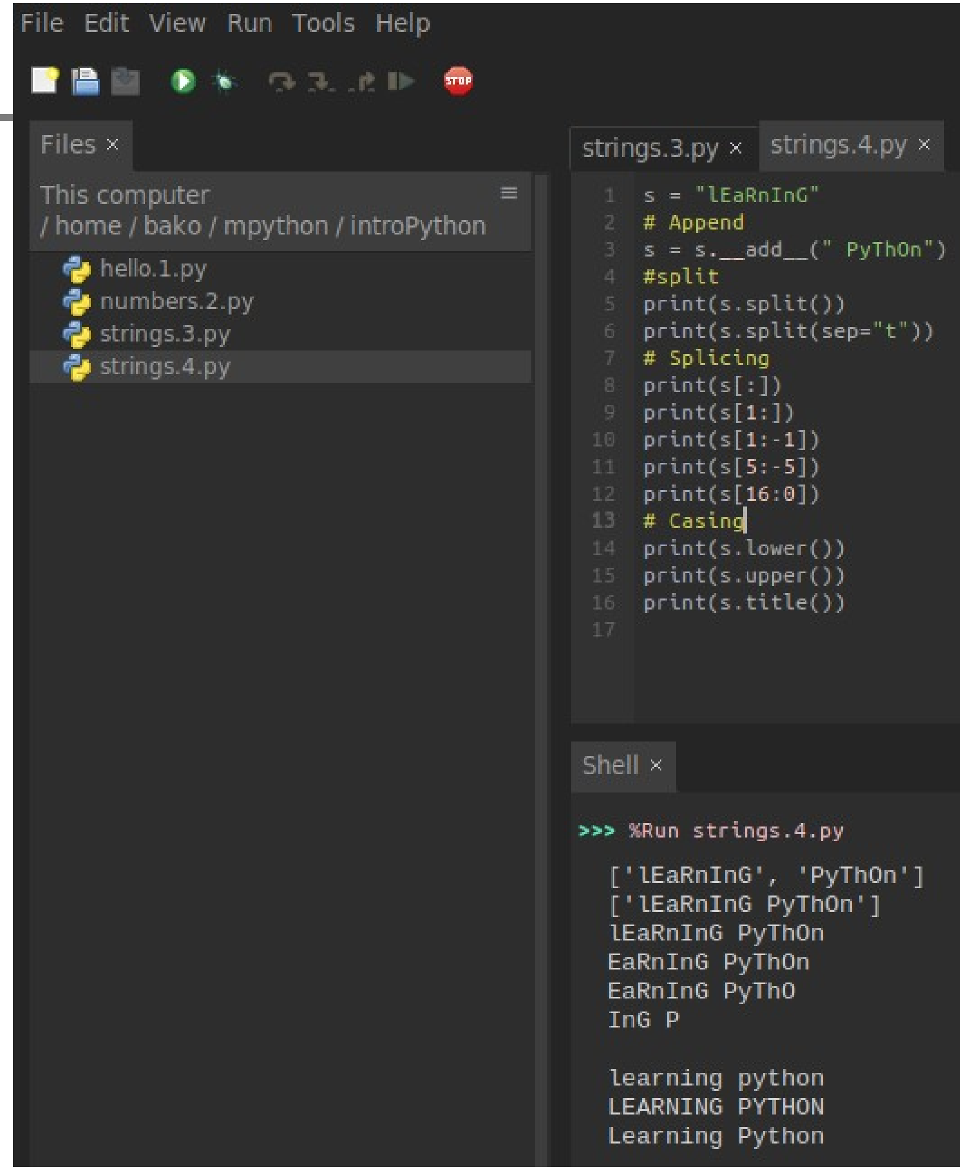


Python – Strings 1

Python Strings



Strings 2



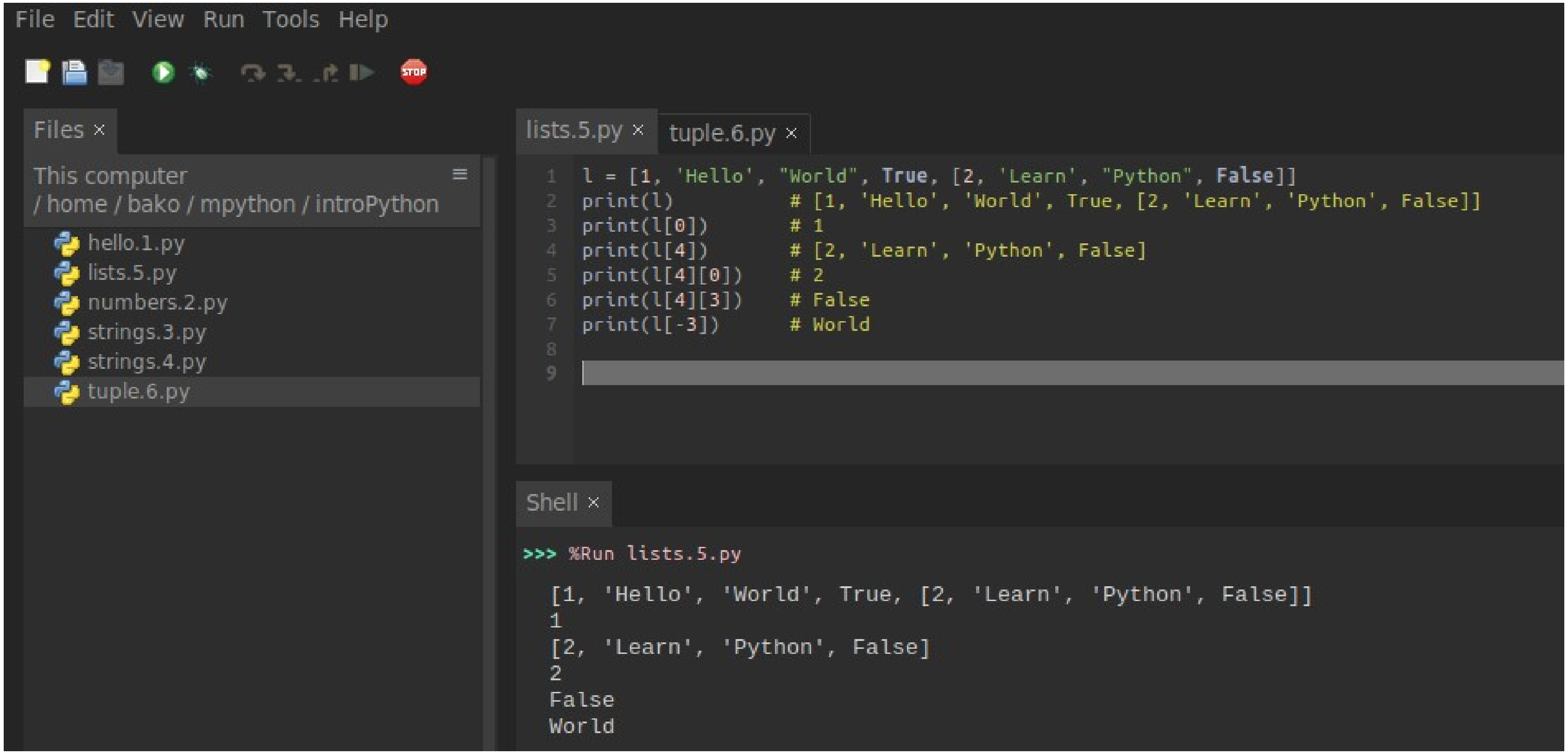
Test the code and change some values:

[1:] to [3:], [5:-5] to [3:-3], etc

```
#s = s.__add__(" PyThOn")
s = s + " PyThon"
```

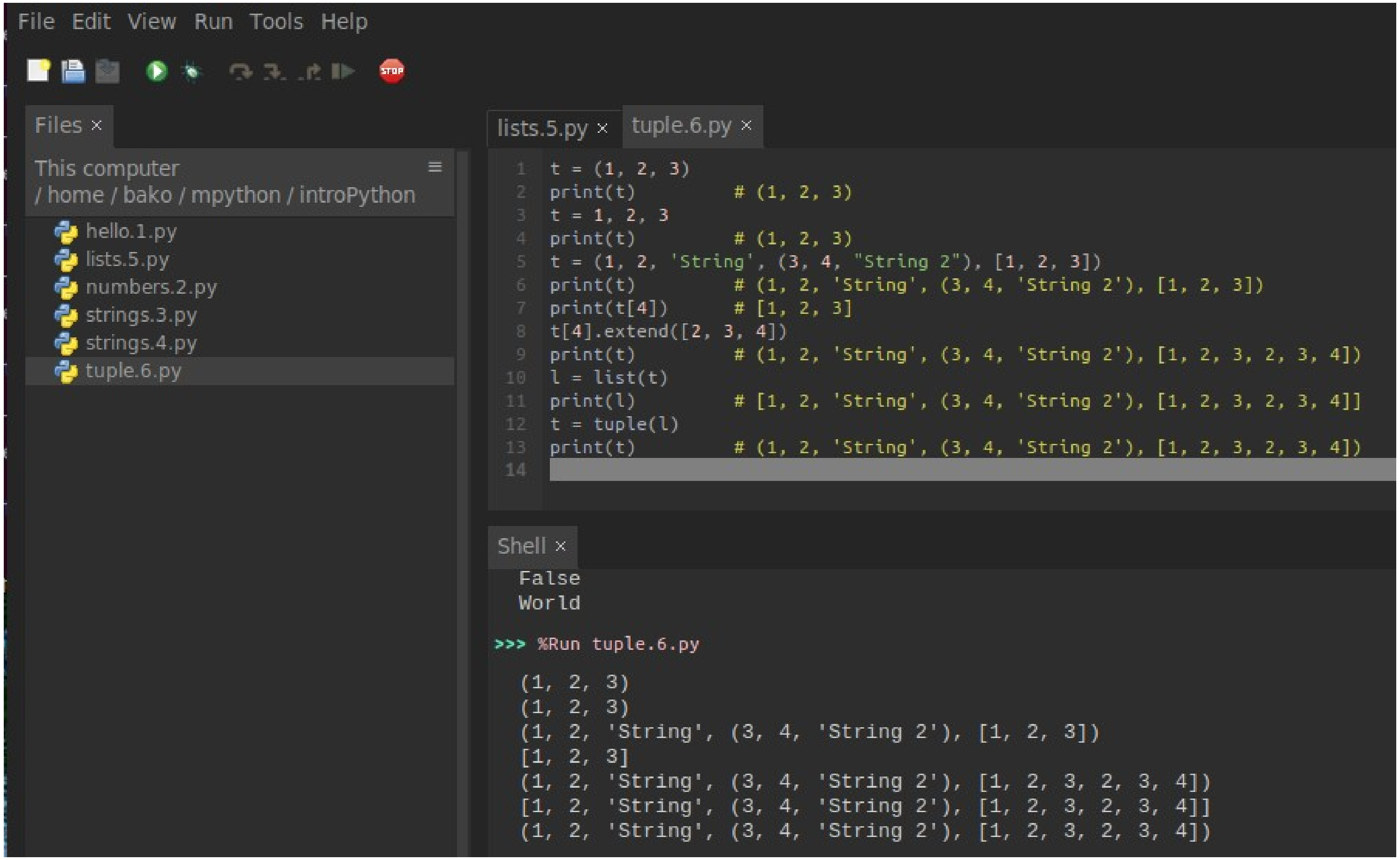
Lists

Python Lists can be used to **process data in groups**. A list can contain a collection of any type of data, including other lists.



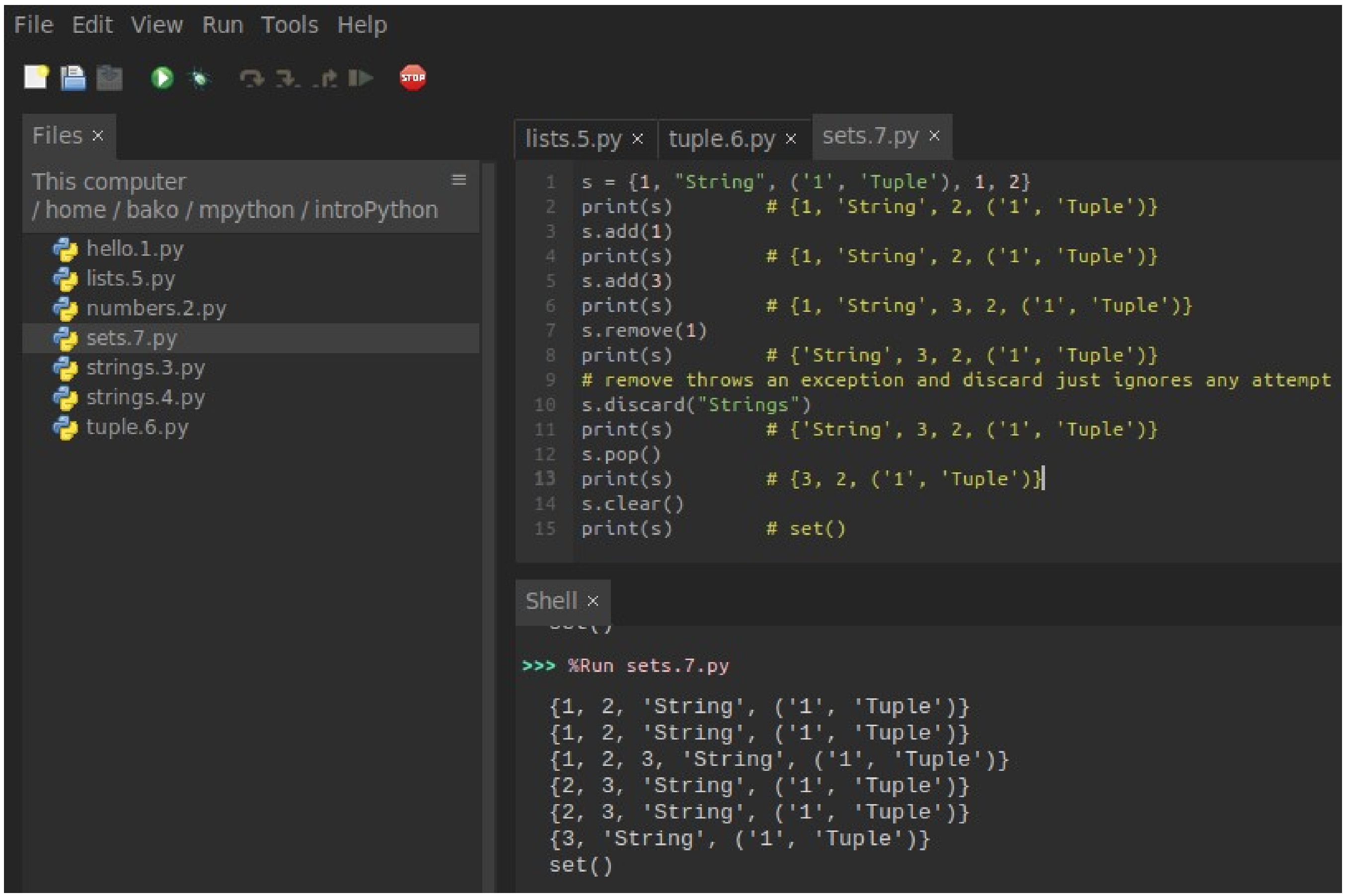
Tuples

Python tuple are **like lists , but are immutable**, they can not be changed once they are defined.



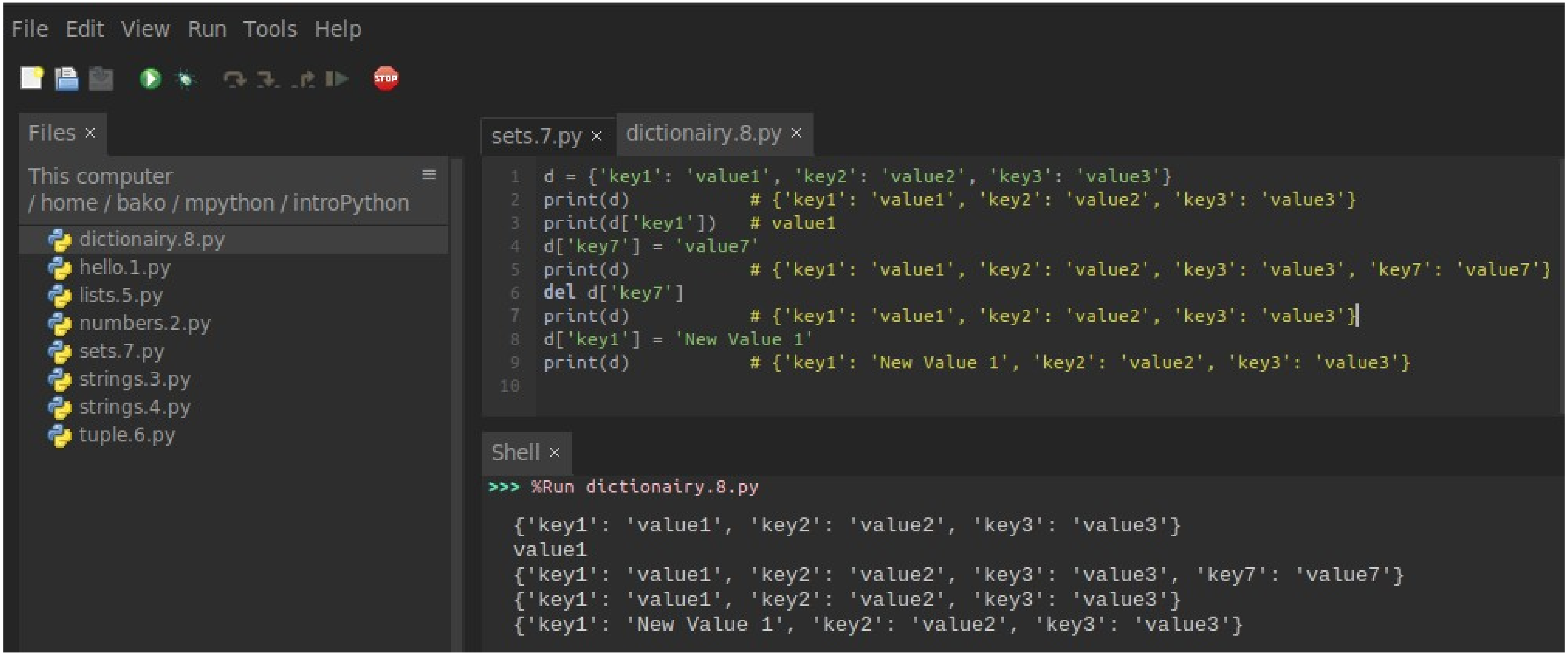
Sets

Sets **do not have any order of elements**. They are defined by data enclosed in **curly braces** **`{..}`** .



Dictionaries

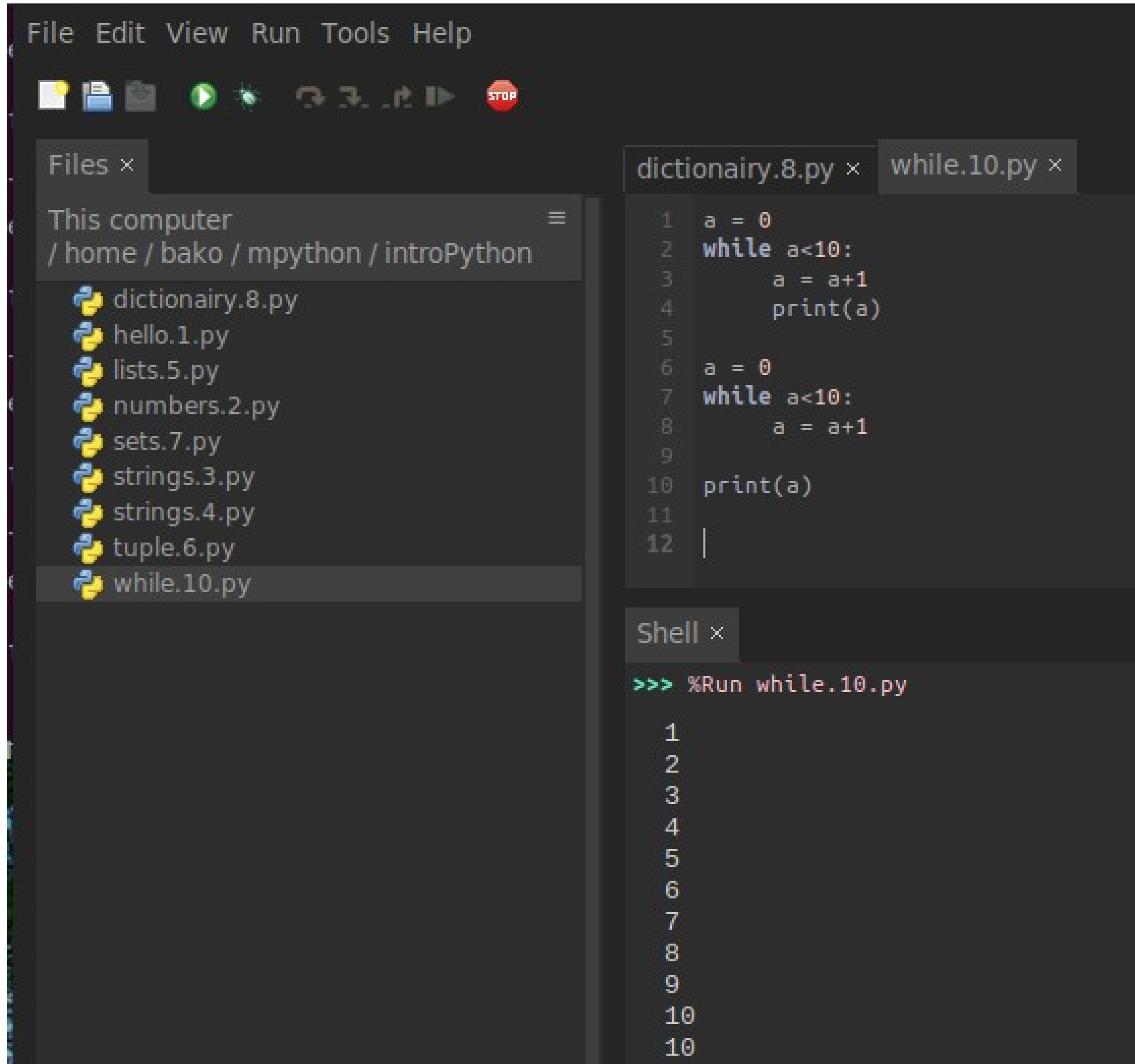
Dictionaries are a special **set of keys** with a value associated with each **key**.



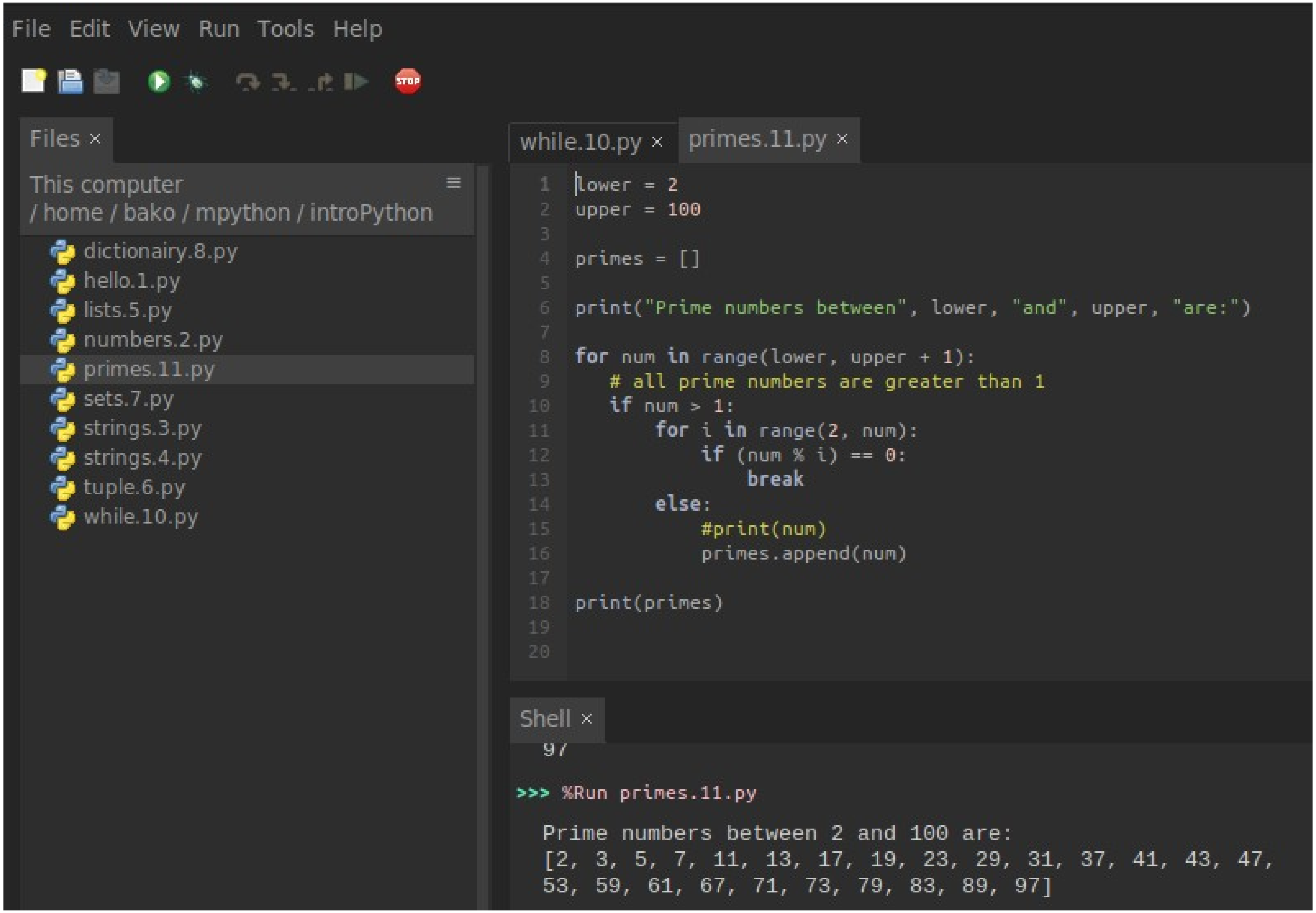
Code flow – `while` loop

Programming is all about **data and decisions**.

Let us check out how decisions can be made with **`while`** loop.

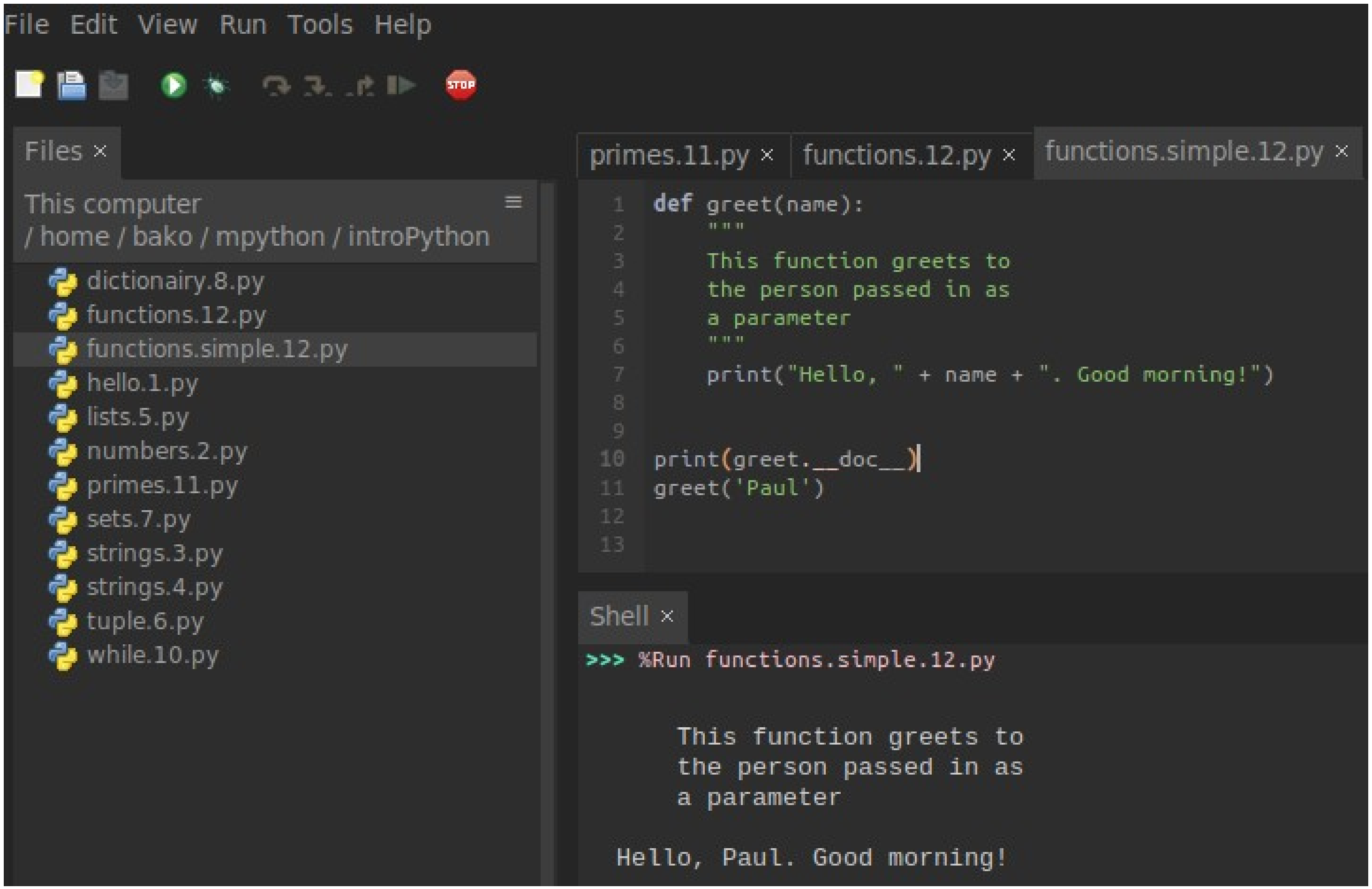


Primes (`for .. in range()`)



Functions, modeles and packages

Simple **function definition** and **call**:



Complex function

primes.11.py × functions.12.py × functions.simple.12.py ×

```python

def getFunction(full=True):
    'Outer Function'
    print(getFunction.__doc__)

    def p(frm=0, to=1, step=1):
        'Inner Function'
        print(p.__doc__)
        return (x ** 3 for x in range(frm, to, step))

    if (full):
        return p
    else:
        return lambda frm = 0, to = 1, step = 1: (x ** 3 \
            for x in range(frm, to, step))

print(__doc__)
t = getFunction()
print("Check the elaborate function")
for v in t(step=1, to=10):
    print(v)
t = getFunction(False)
print("Check the lambda function")
for v in t(1, 5):
    print(v)
```

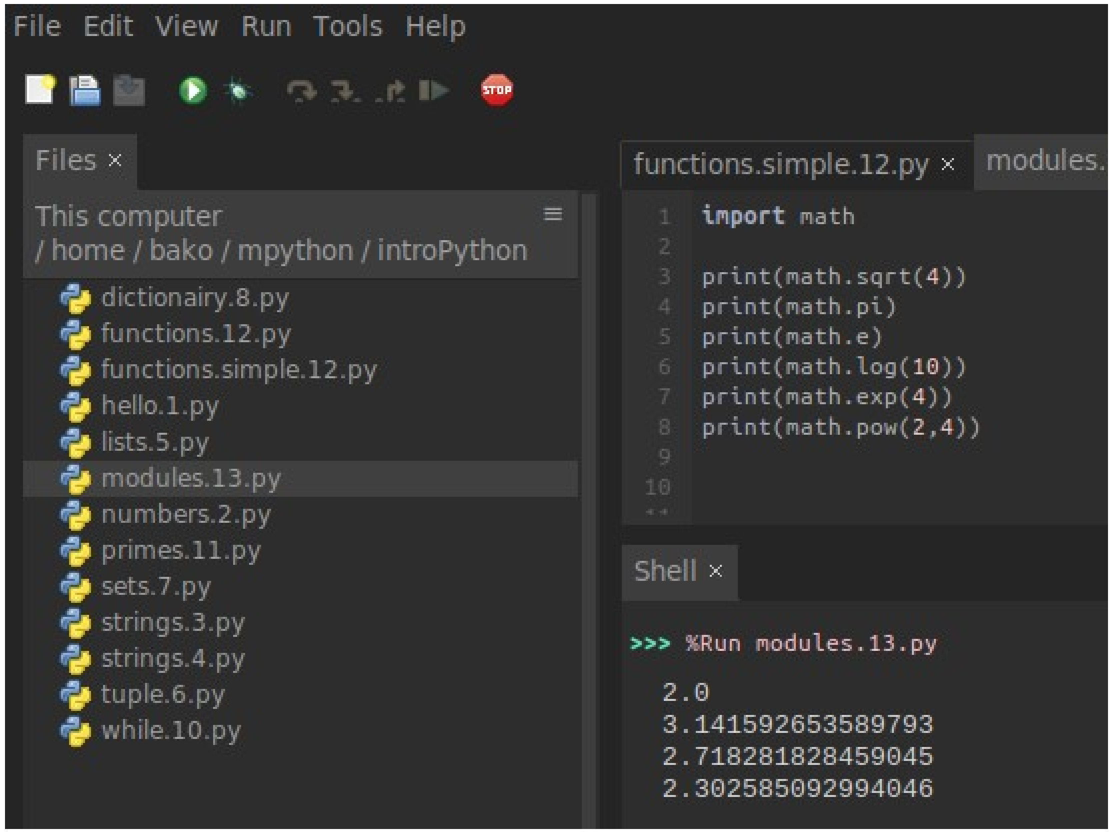
```python
print(__doc__)
t = getFunction()
```

Shell ×

```
>>> %Run functions.12.py

  None
  Outer Function
  Check the elaborate function
  Inner Function
  0
  1
  8
  27
  64
  125
  216
  343
  512
  729
  Outer Function
  Check the lambda function
  1
  8
  27
  64
```

Modules - math



class definition

The simplest form of **class definition** looks like this:

```
class ClassName:
    <statement-1>
    .
    <statement-N>
```

Class definitions, like function definitions (**def** statements) must be executed before they have any effect.

Class objects support **two kinds of operations**:
→ **attribute references** and
→ **instantiation**.

`class` definition and references

Attribute references use the standard syntax used for all attribute references in Python: **`obj.name`**.

Valid attribute names are all the names that were in the class's **namespace** when the class object was created.

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

then **`MyClass.i`** and **`MyClass.f`** are valid attribute references, returning an **integer** and a **function object**, respectively.
Class attributes can also be assigned to, so you can change the value of **`MyClass.i`** by assignment. **`__doc__`** is also a valid attribute, returning the docstring belonging to the class: "**`A simple example class`**".

`class` instantiation , `__init__` method

Class instantiation uses function notation. Just pretend that the class object is a parameterless function that returns a **new instance of the class**.

For example (assuming the above class):

```
x = MyClass()
```

creates a new instance of the class and **assigns this object to the local variable** `x`.

The instantiation operation (“calling” a class object) creates an **empty object**. Many classes like to create objects with instances customized to a specific initial state.

Therefore a class may define a special method named `__init__()`, like this:

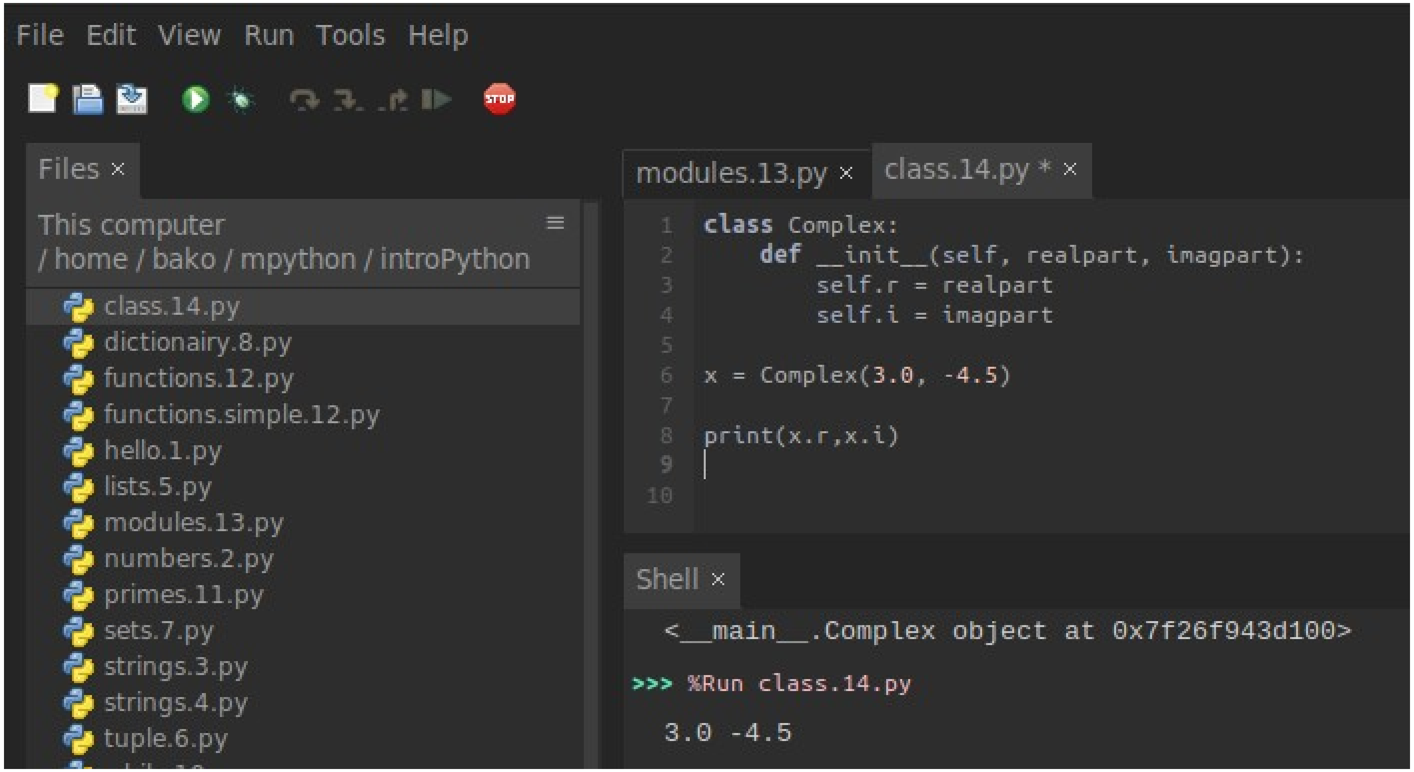
```
def __init__(self):
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly-created class instance.

class instantiation , __init__ method

The `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`.

For example,



Instance objects

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names: **data attributes** and **methods**.

Data attributes need not be declared; like local variables, **they spring into existence** when they are first assigned to.

For example, if `x` is the instance of `MyClass` created above, the following piece of code will print the value 16, without leaving a trace:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

Instance objects

Data attributes need not be declared; like local variables, **they spring into existence** when they are first assigned to.

For example, if **`x`** is the instance of **`MyClass`** created above, the following piece of code will print the value 16, without leaving a trace.

class.15.py

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'

x=MyClass()

x.counter = 1
x.toto =3

while x.counter < 10:
    x.counter = x.counter * 2

print(x.counter)
print(x.f())
print(x.i)
print(x.toto)
```

Shell

```
Python 3.8.10 (/usr/bin/python)
>>> %Run class.15.py

  16
  hello world
  12345
  3
```

Method objects

A method is called right after it is bound:

```
x.f()
```

In the **`MyClass`** example, this will return the string '**`hello world`**'.

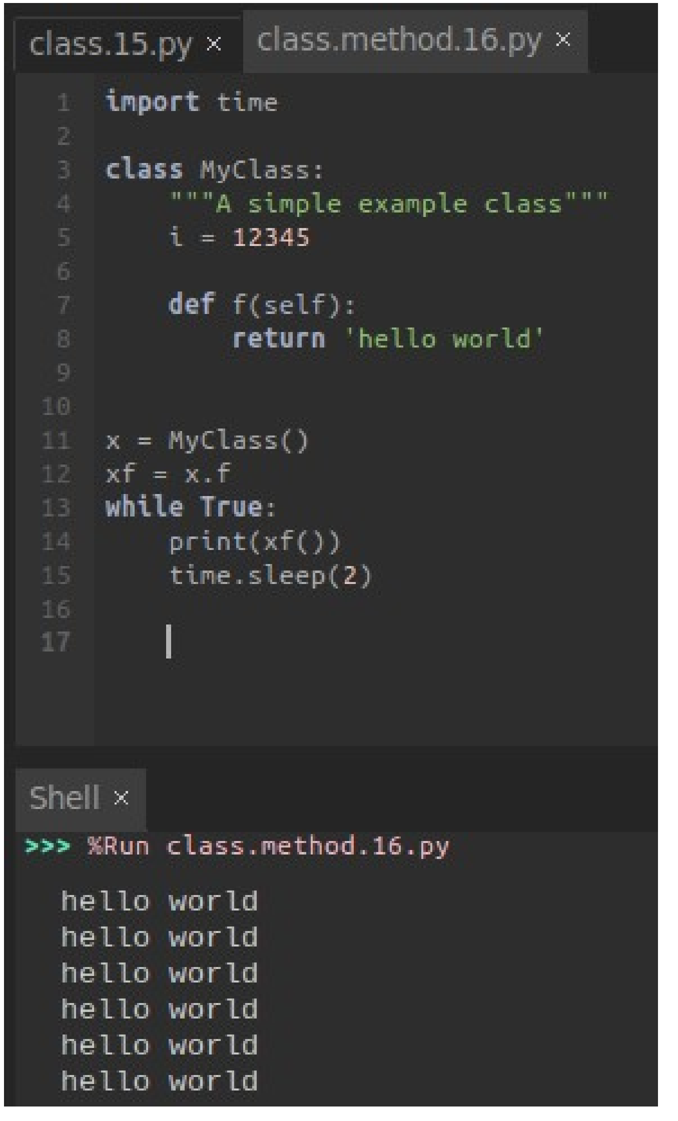
`x.f` is a **method object**, and can be stored away and called at a later time.

For example:

```
xf = x.f
while True:
    print(xf())
    time.sleep(2)
```

will continue to print **`hello world`** until the end of time.

The call **`x.f()`** is exactly equivalent to **`MyClass.f(x)`**



Class and instance variables

Instance variables are for **data unique to each instance** and

Class variables are for attributes and methods **shared by all instances of the class**.

class.method.16.py × | class.inst.variables.py * ×

```python
class Dog:
    kind = 'canine'
    # class variable shared by all instances

    def __init__(self, name):
        self.name = name
        # instance variable unique to each instance

d = Dog('Fido')
e = Dog('Buddy')
print(d.kind)
print(e.kind)                # shared by all dogs
print(d.name)                # unique to d
print(e.name )               # unique to e
```

Shell ×

```
>>> %Run class.inst.variables.py

  canine
  canine
  Fido
  Buddy
```

Class and instance variables

Instance variables are for **data unique to each instance** and

Class variables are for attributes and methods **shared by all instances of the class**.

class.inst.variables.py * × class.inst.var.init.py * ×

```python
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []
        # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
print(d.tricks)
print(e.tricks)
```

Shell ×

```
>>> %Run class.inst.var.init.py
  ['roll over']
  ['play dead']
```

Summary

Python has been one of the world's most popular programming languages for a long time, and for good reason.

Due to its relatively straightforward syntax, it's one of the easiest languages to learn, and it's so remarkably scalable and general-purpose that it's used in a huge array of fields, from web development to machine learning.

It remains one of the **best programming languages for entrepreneurs** to learn because of this general-use nature.

MicroPython is a simplified version of Python (3) with some additional features to program embedded systems and IoT devices.

It is our choice to develop practical IoT architectures based on our IoT DevKits