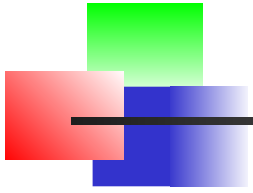


“Practical IoT for Business Schools”

“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



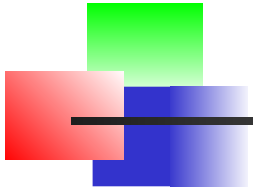
IoT – global picture

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

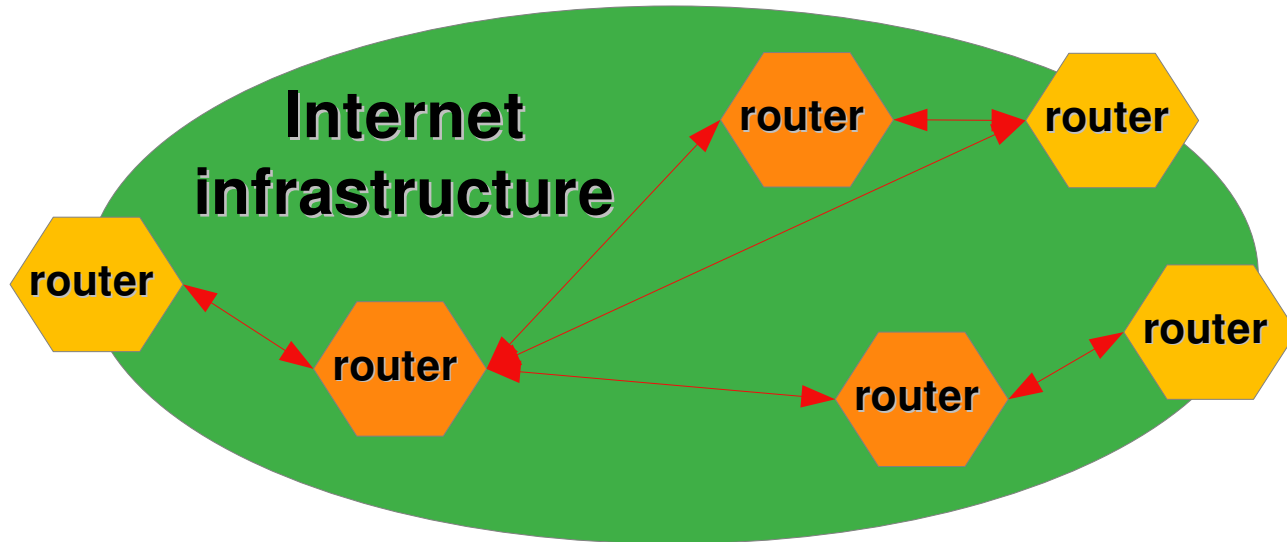
Things: Embedded Software/Hardware

Internet: Communication means

Terminology, terminology, terminology , ..



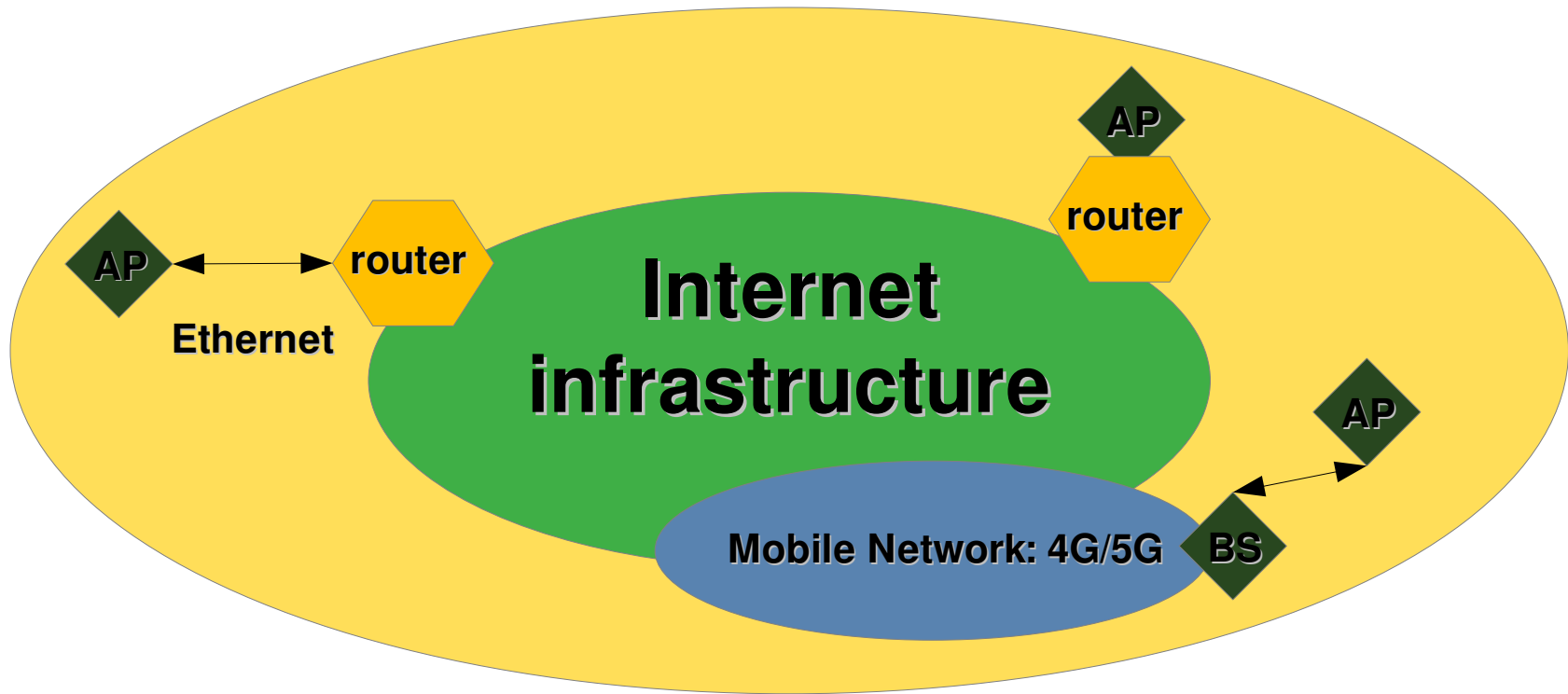
Internet Infrastructure



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

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

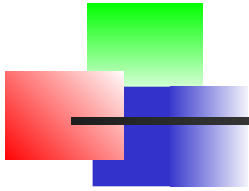
Internet – Access Points



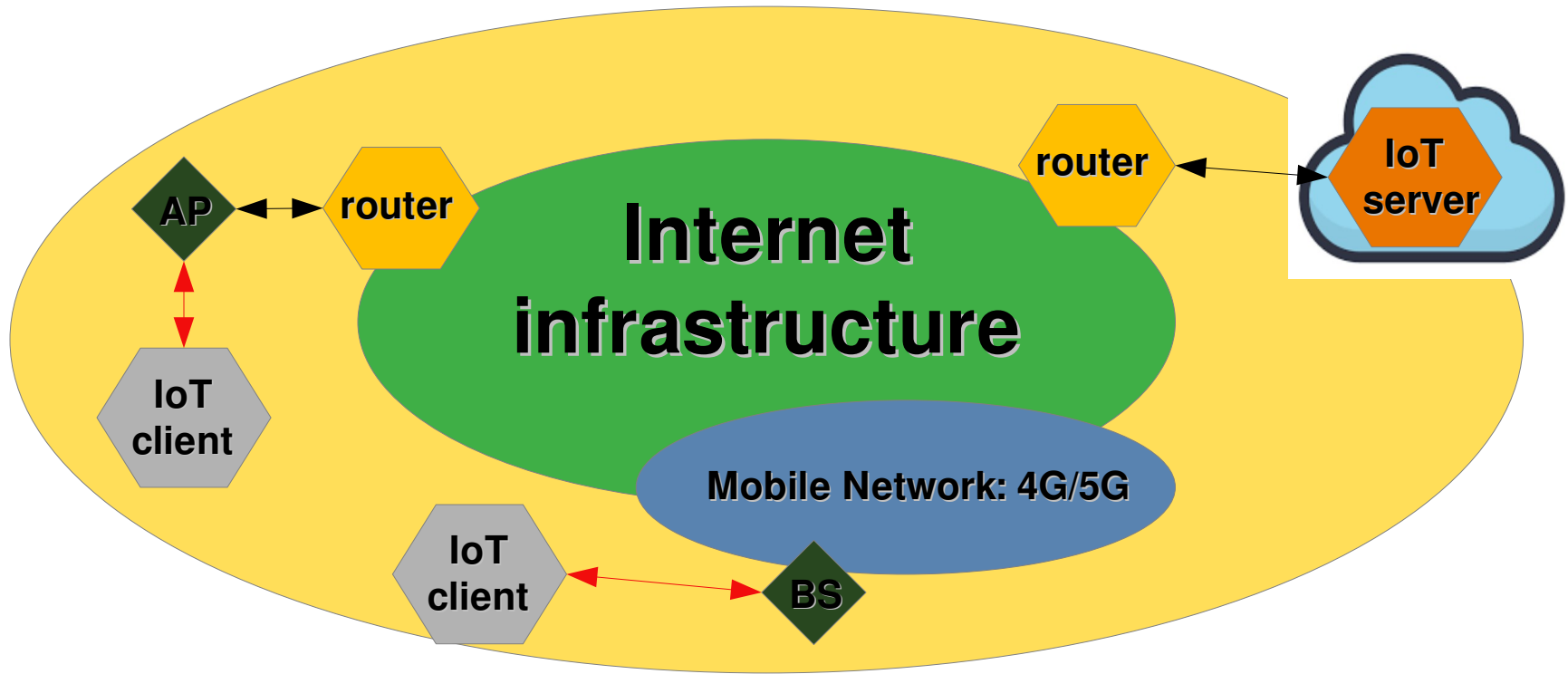
Ethernet – is local wired access to Internet for device, AP, switch, ..

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



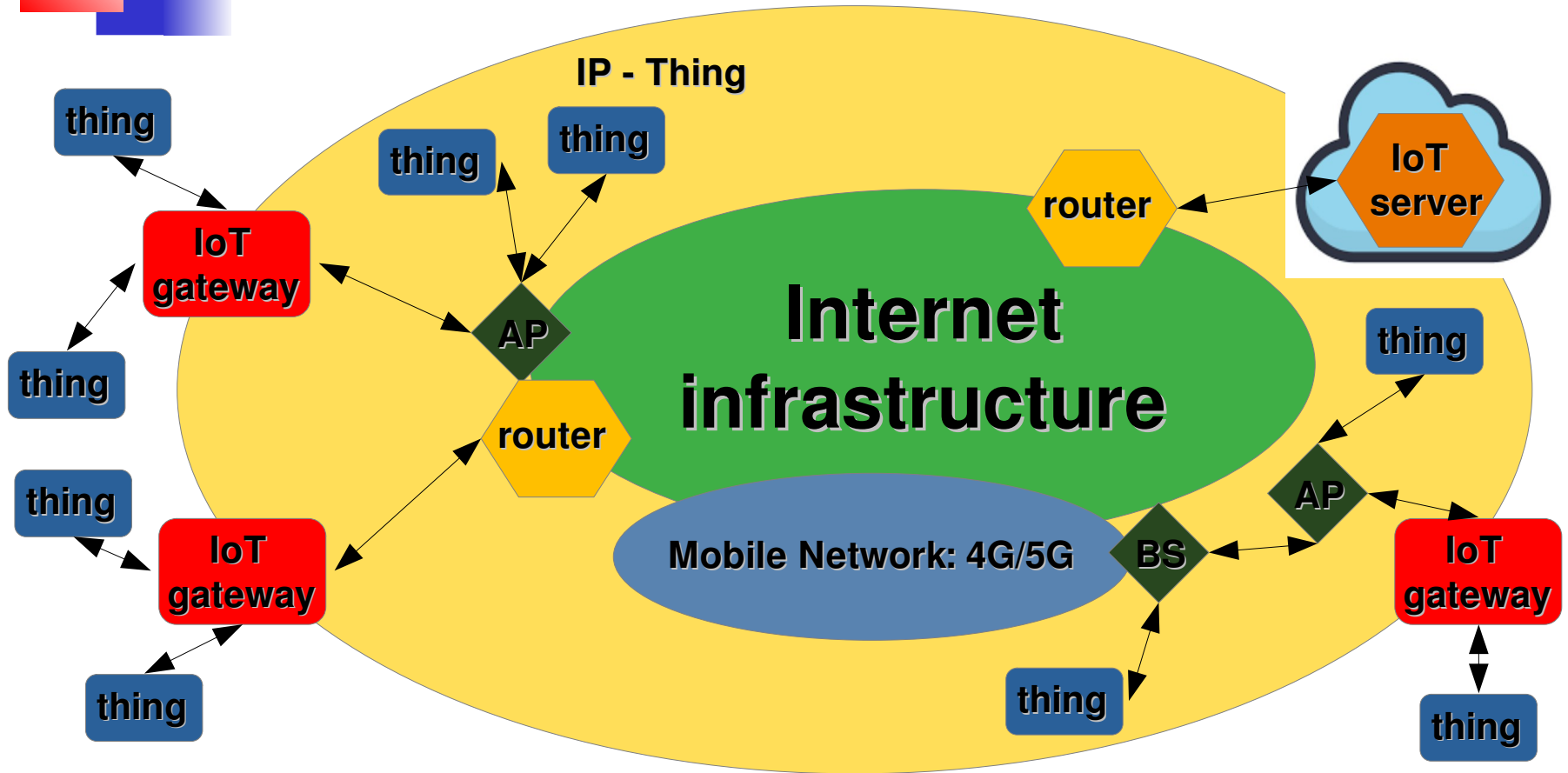
IoT Server and IoT Client are external devices

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

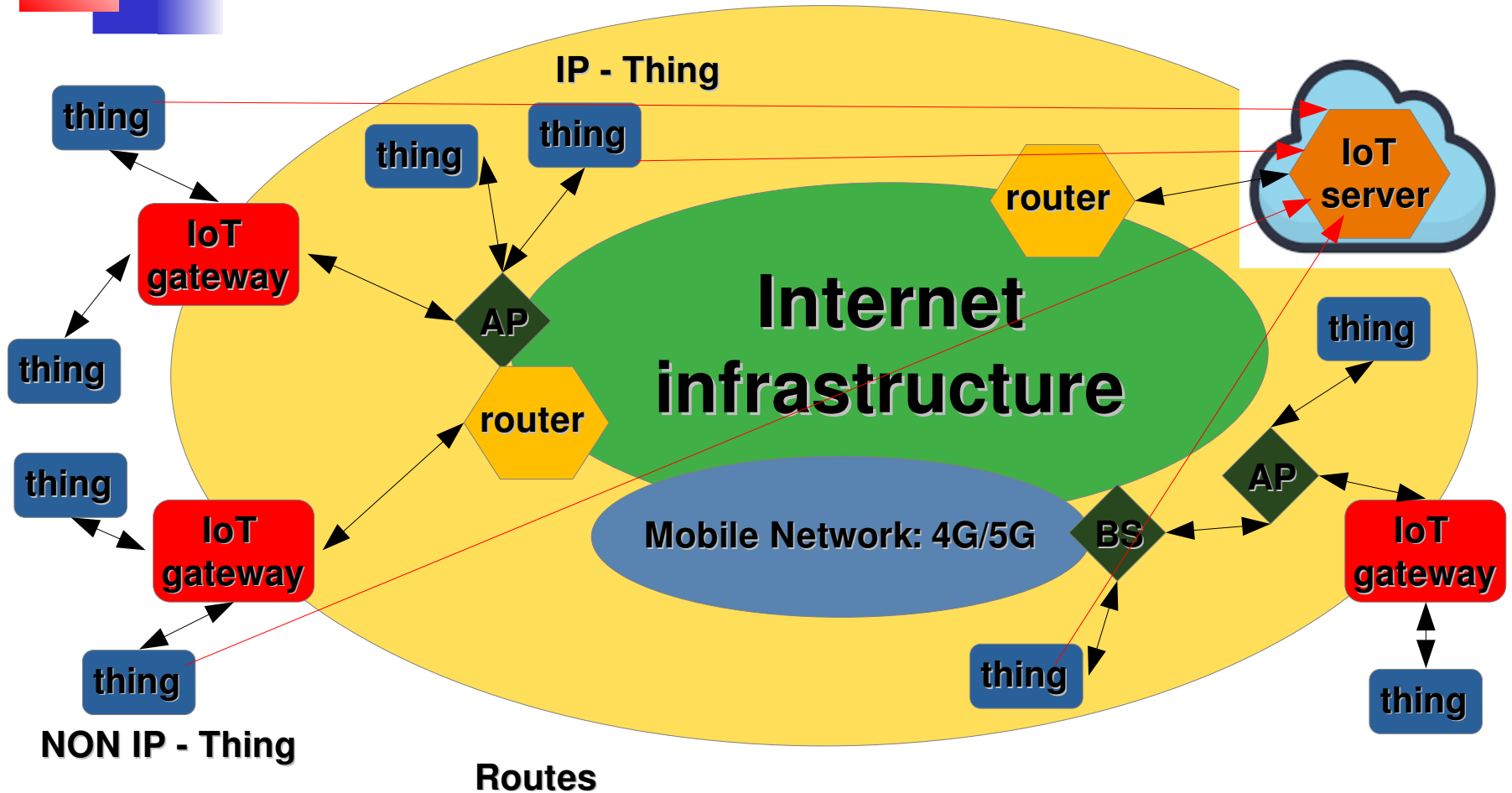


NON IP - Thing

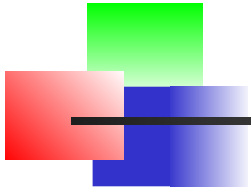
Thing is a Terminal device

IoT gateway is an intermediate device

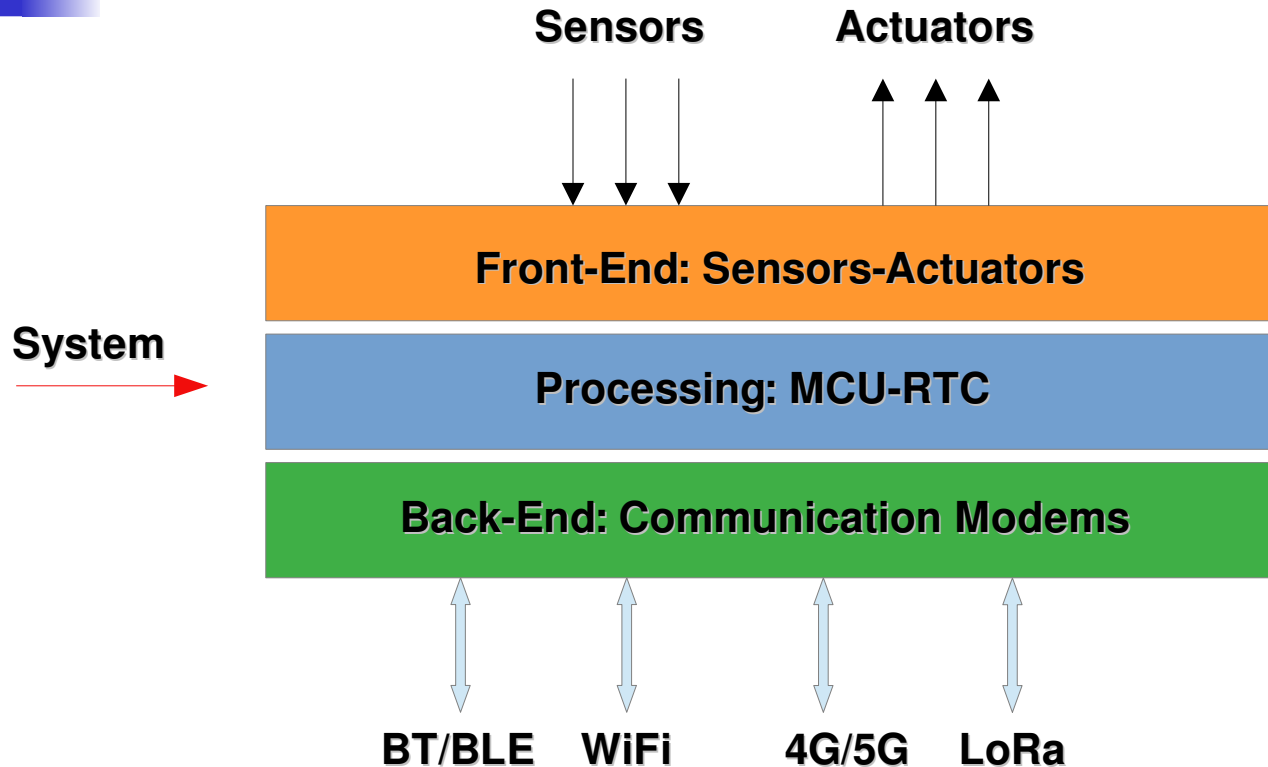
Routes and Examples - Discussion



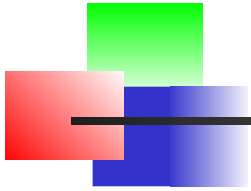
Examples, examples, examples - Discussion



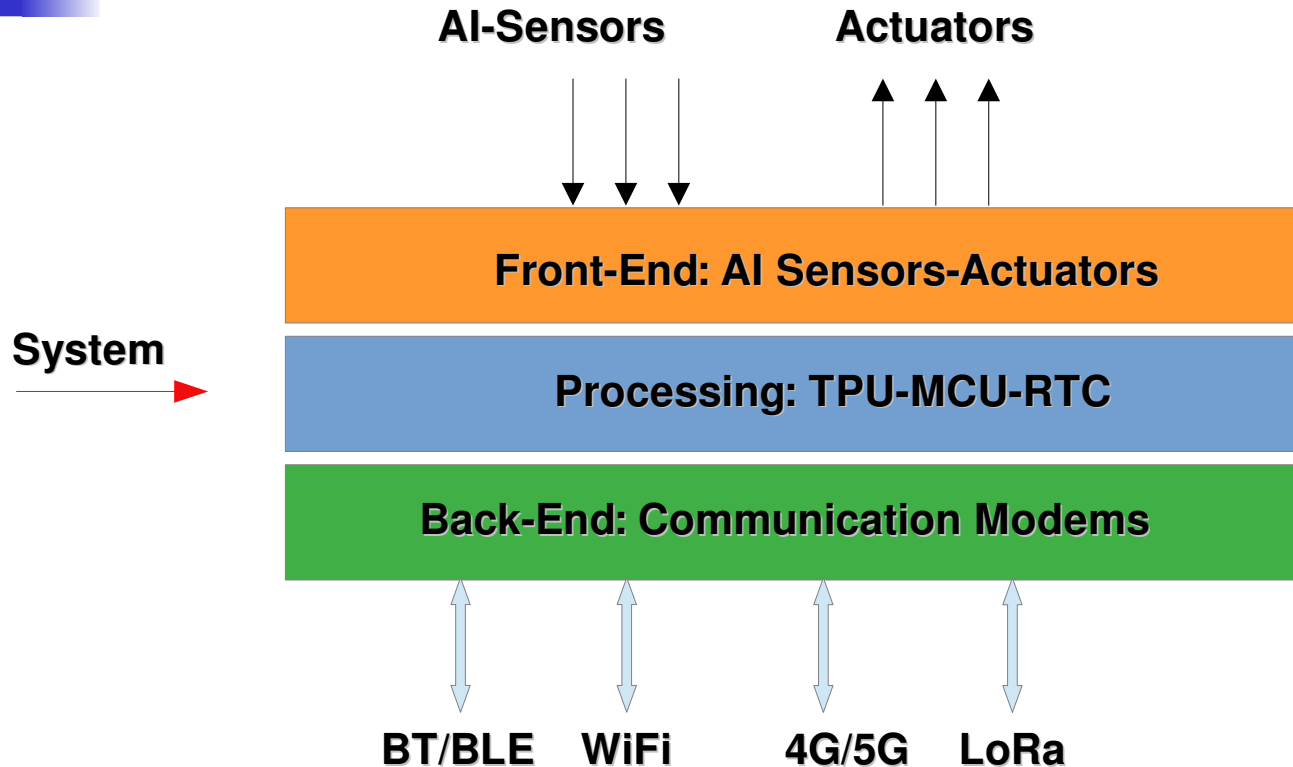
Simple and Intelligent Things



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

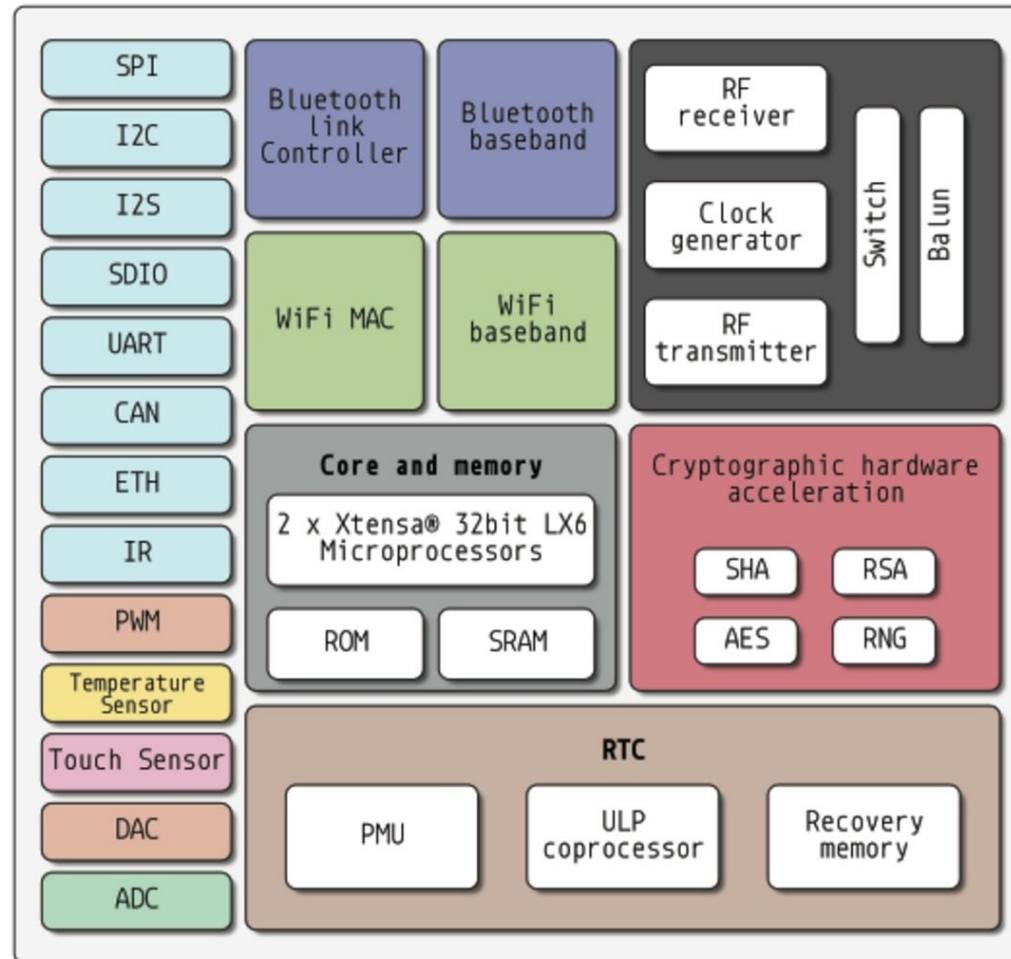
Low-power < 1-100mA (5V)

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

rich-interfaces

Wi-Fi

BT-BLE



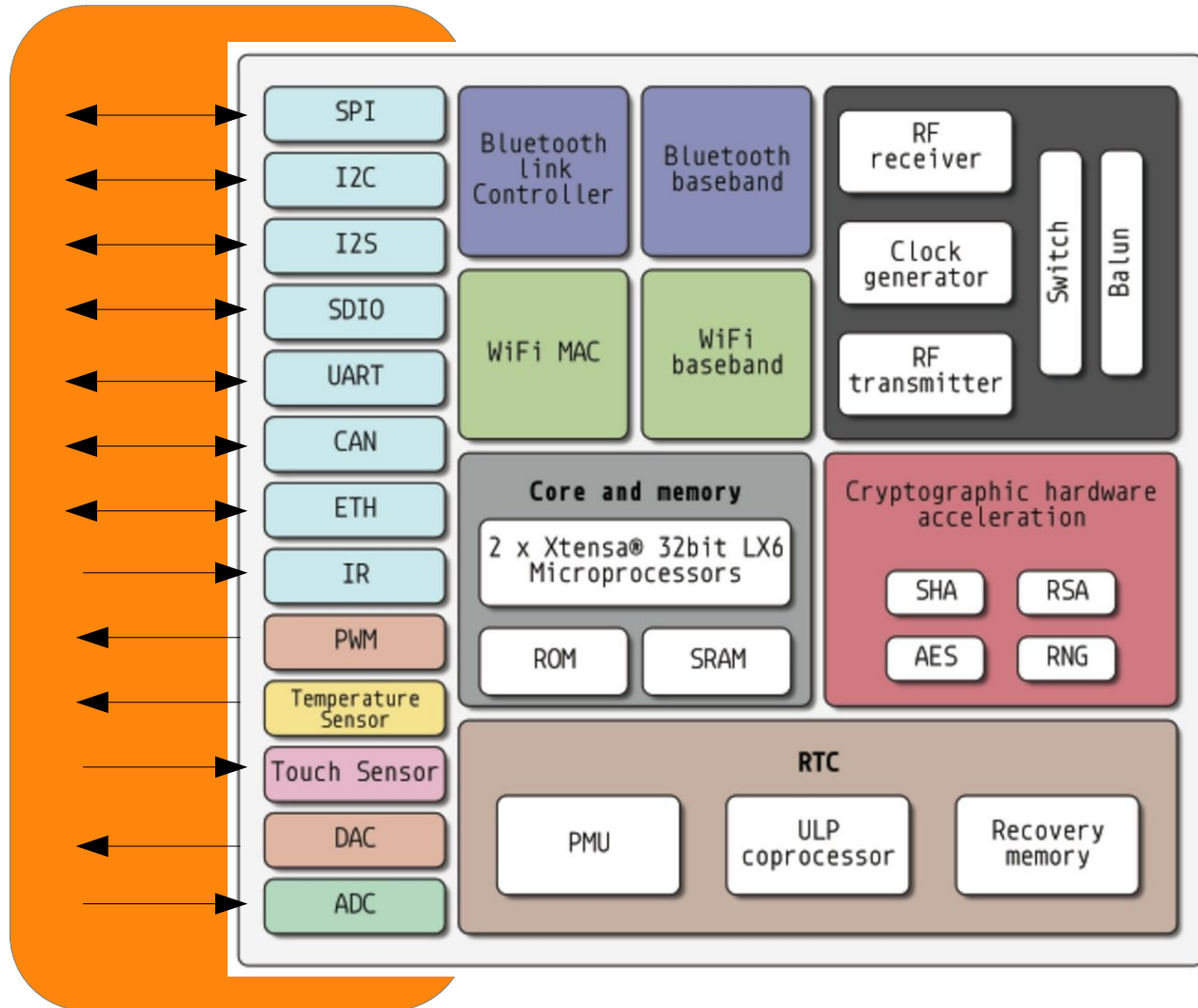
Real example – System on Chip

interfaces
to sensors
and
actuators

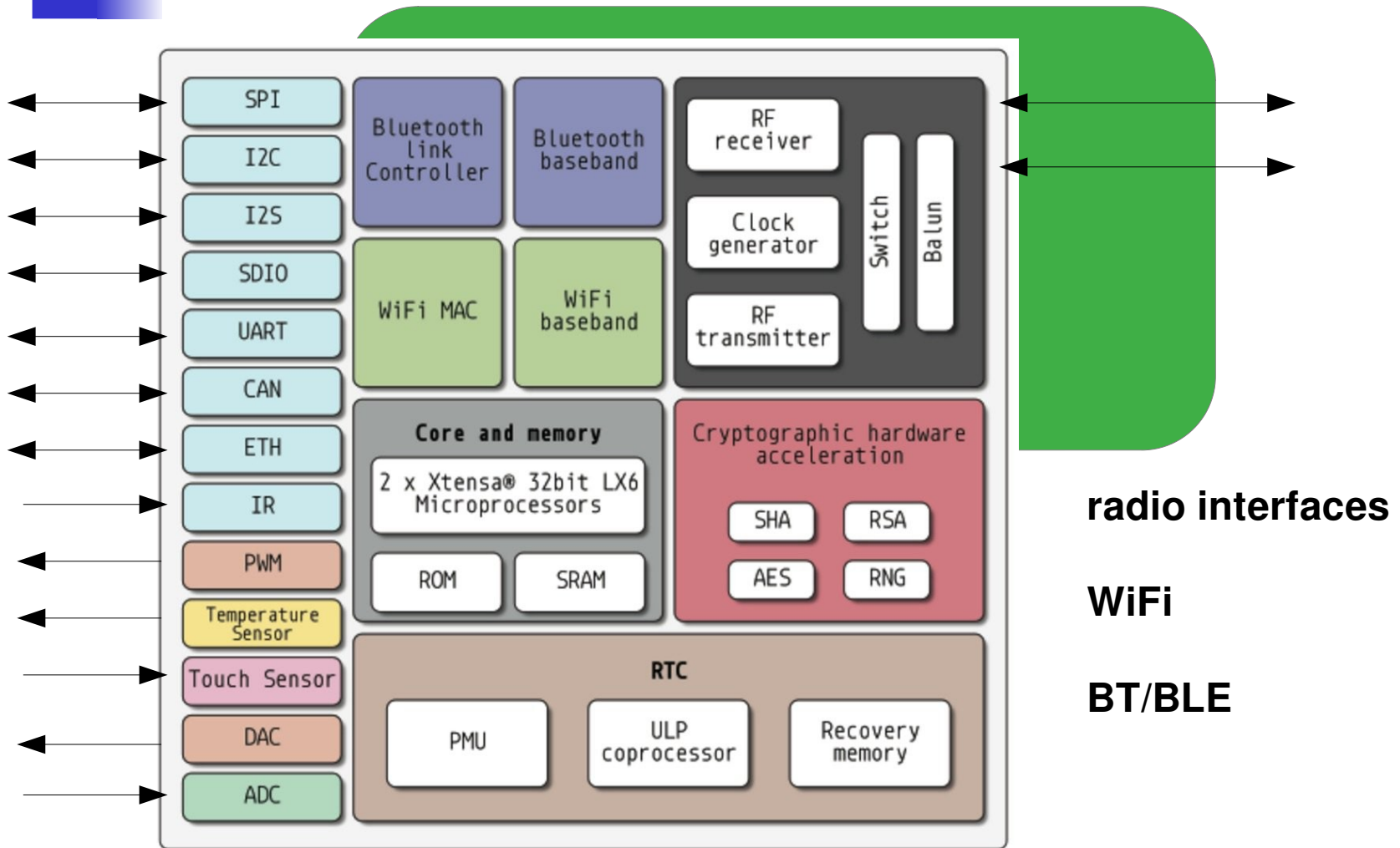
UART

I2C

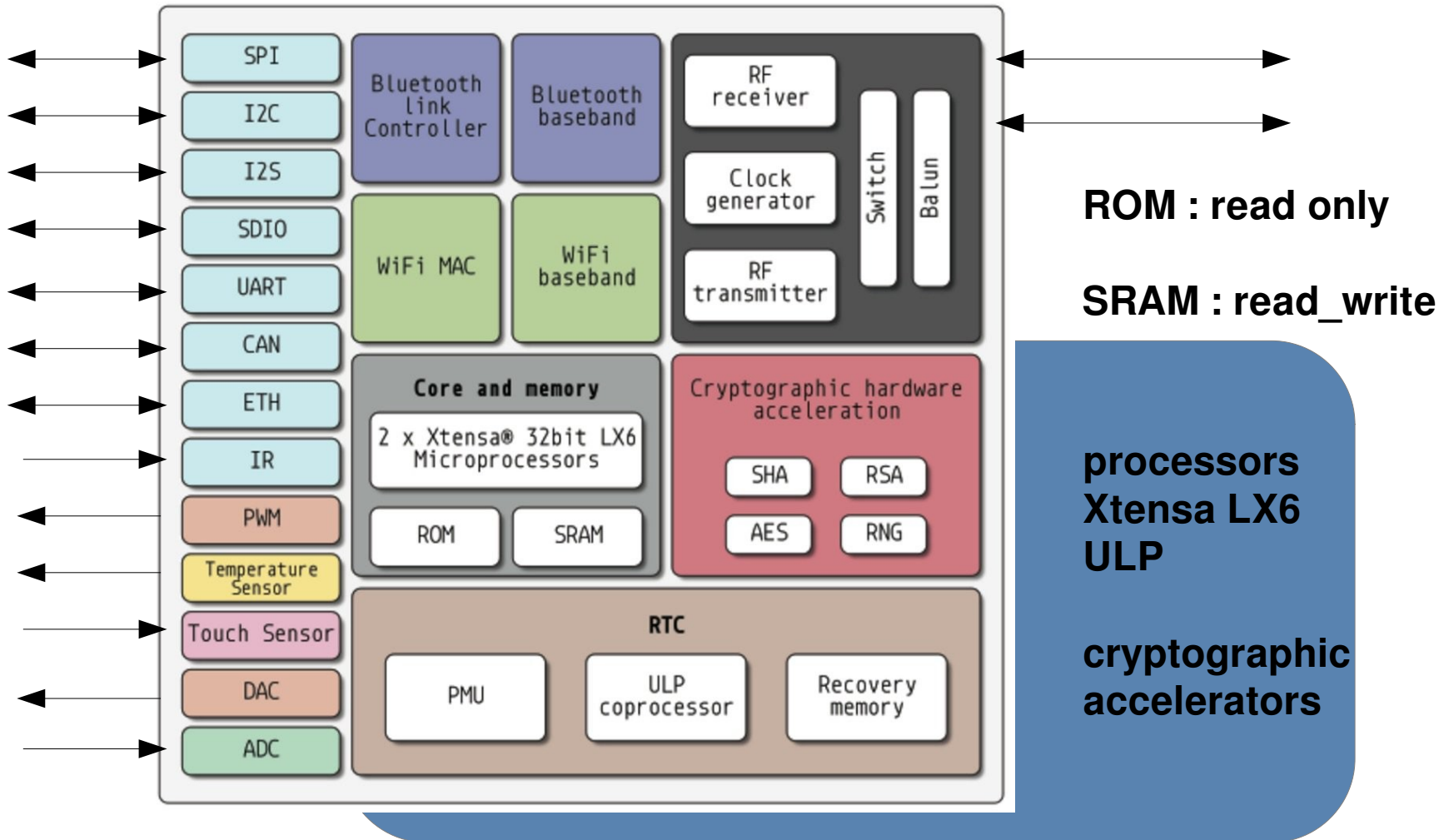
SPI



Real example – System on Chip



Real example – System on Chip



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

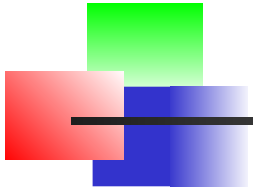
Soft cores

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.

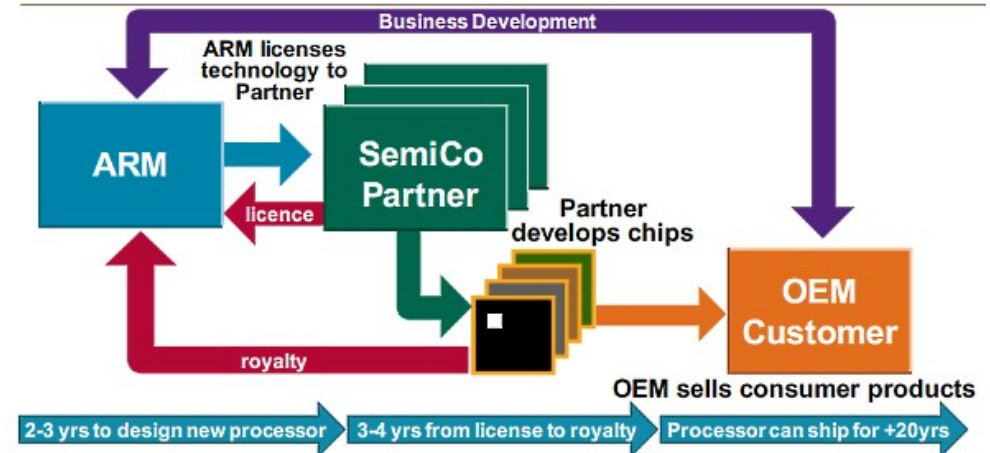
ARM business model

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**.

ARM Business Model

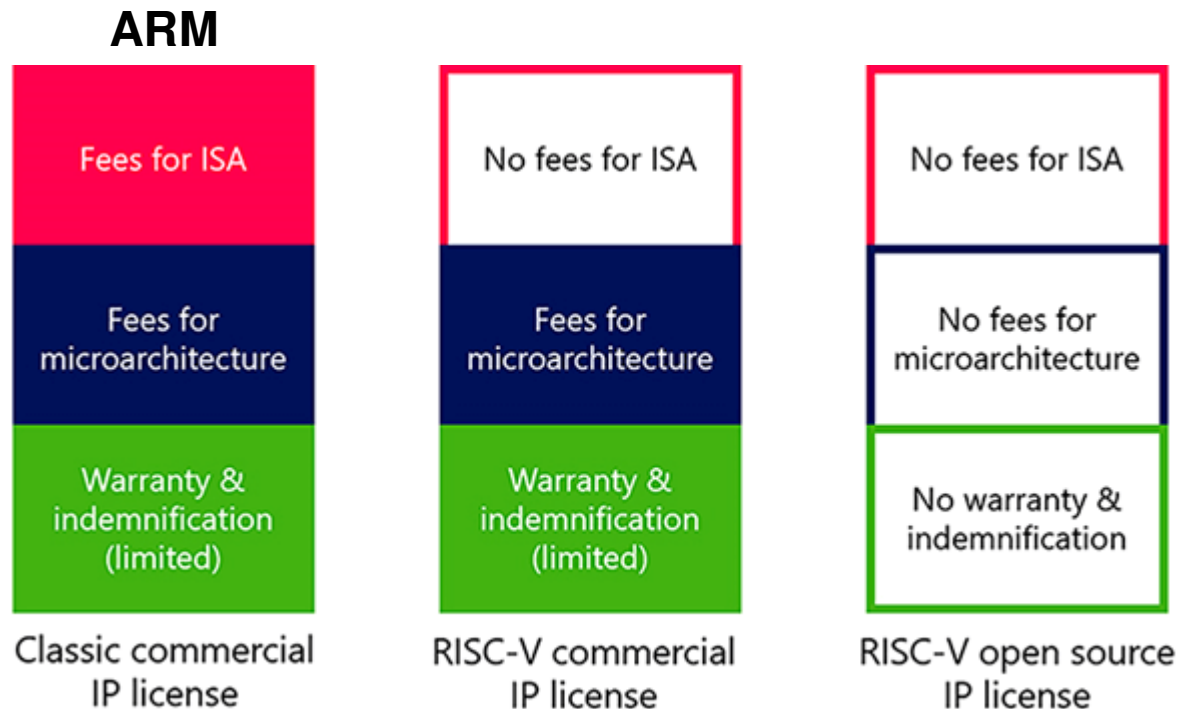


The licensing fees vary between an estimated **\$1 million to 10 million**. The royalty is usually **1 to 2% of the selling price of the chip**. Licensing enables ARM to scale the business efficiently.

ARM vs RISC-V business model

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

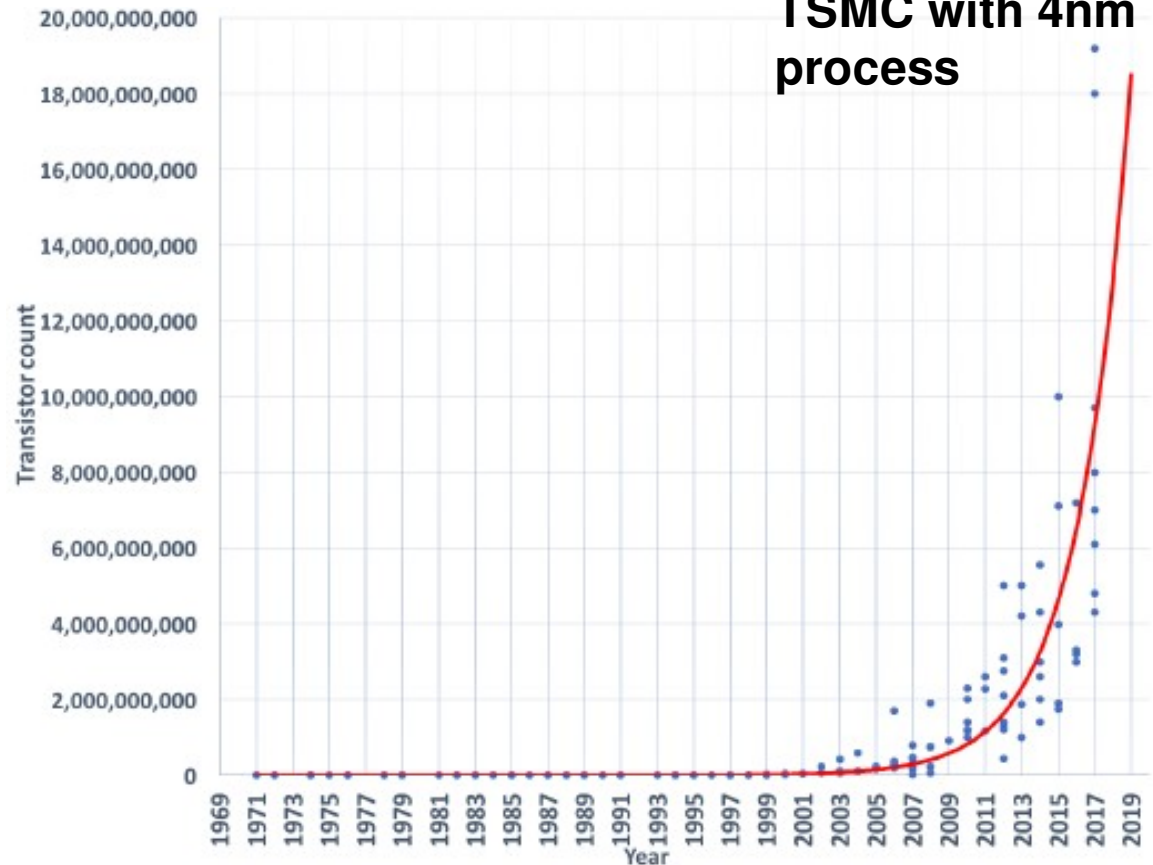
RISC-V is a **standard and open architecture** with no fees for ISA (Instruction Set Architecture)



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

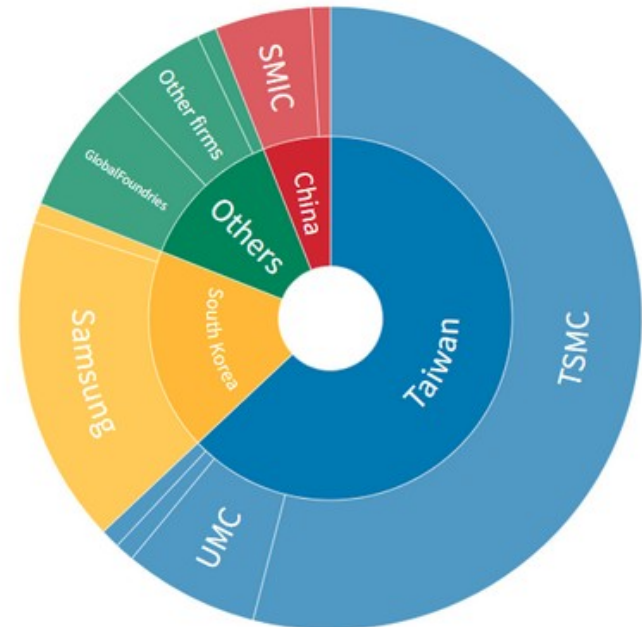
The winner takes it all !



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:

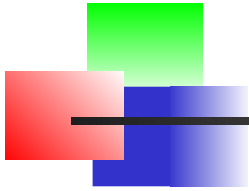
- A new foundry (4 in construction by TSMC) costs about **\$15 billion** (more than a nuclear plant); TSMC - \$44 billion CAPEX for 2022 for **3nm** and **2nm** nodes.
- One (EUV) chip machine (**ASML**) costs up to **\$250 million**. All these machines (production **60/year**) are already sold up to 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

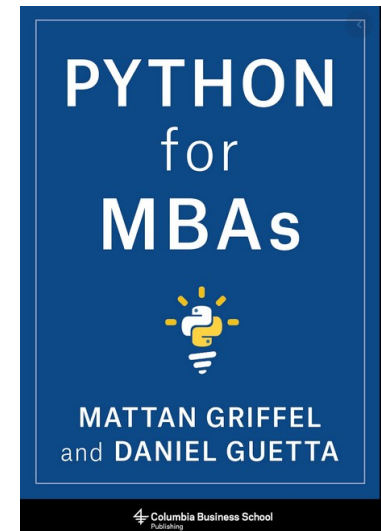


“IoT – Software aspects”

“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](#)

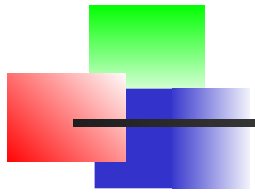
The screenshot displays the Thonny IDE interface. The main editor window shows a Python script named 'factorial.py' with the following code:

```
def fact(n):  
    if n == 0:  
        return 1  
    else:  
        return fact(n-1) * n  
  
n = int(input("Enter a natural number"))  
print("Its factorial is", fact(3))
```

The Shell window at the bottom shows the execution of the script with the following output:

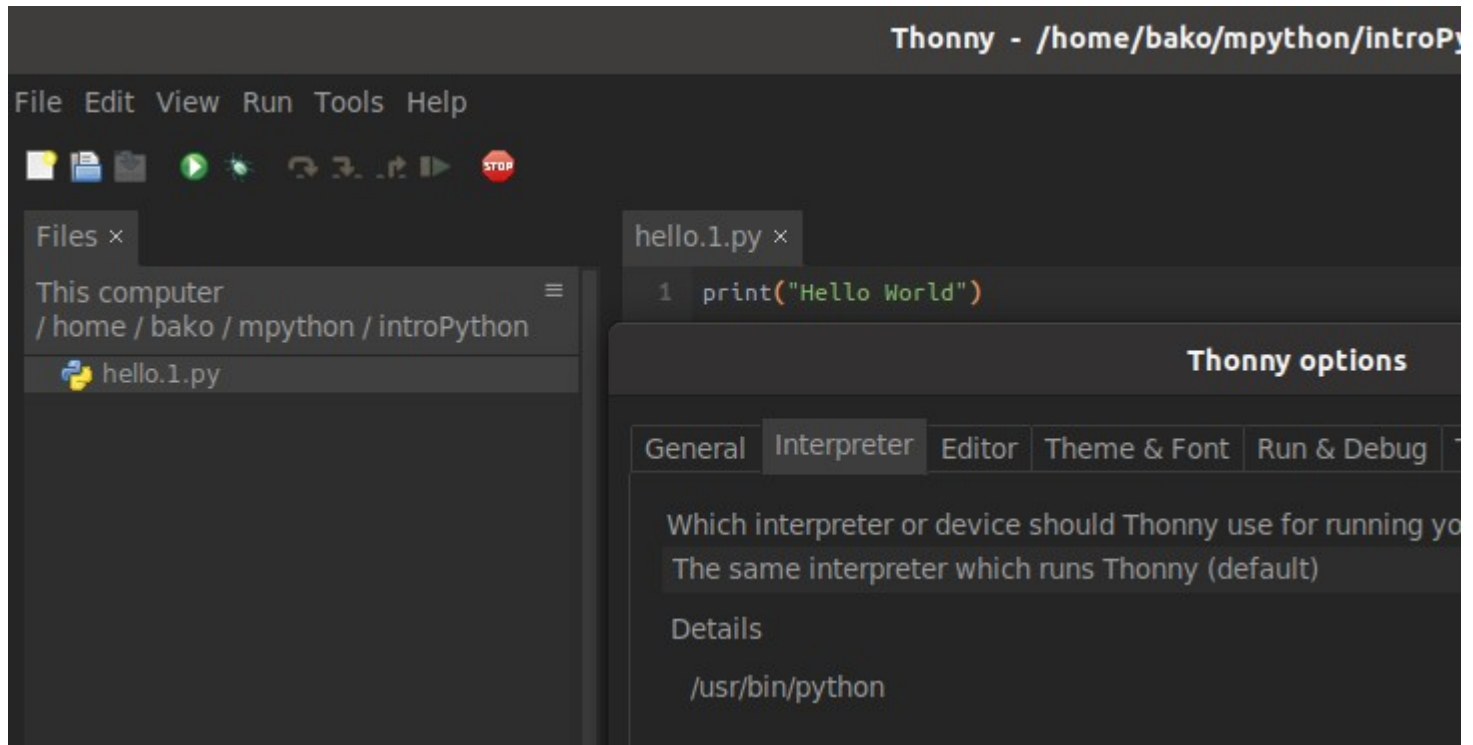
```
>>> %Debug factorial.py  
Enter a natural number: 3
```

Two debug windows are open, showing the state of the function call. The 'fact(3)' window shows the function definition and the local variable 'n' with the value 3. The 'fact(2)' window shows the function definition and the local variable 'n' with the value 2. The 'fact(2)' window also shows the return statement 'return fact(2-1) * n' with the value 2-1 highlighted.



Python – interpreter

Choose the Python interpreter – the same as Thonny





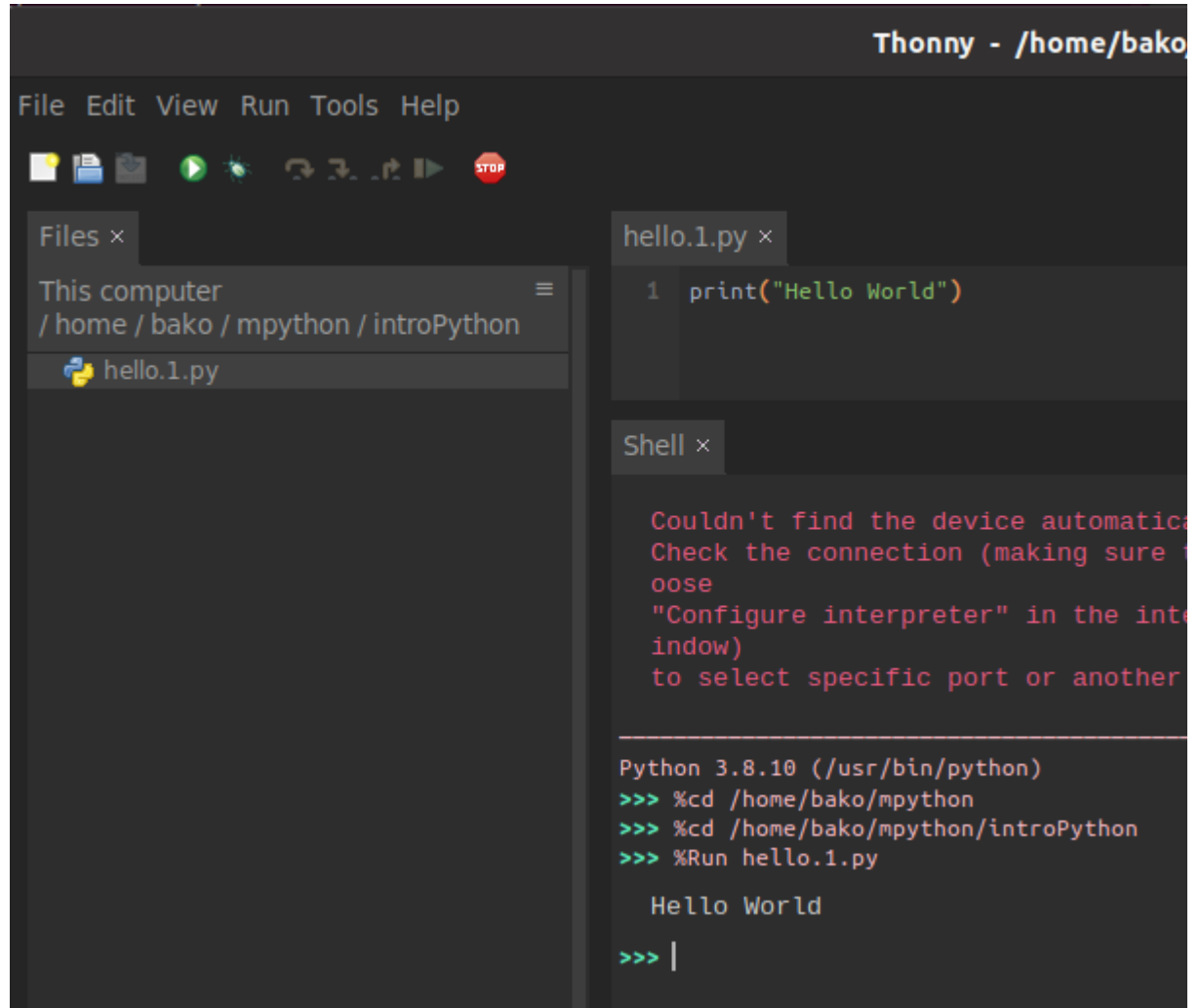
Python – first code

3 windows:

→ Files

→ Editor

→ Shell - terminal



```
Thonny - /home/bako...
File Edit View Run Tools Help
[Icons]
Files x
This computer
/home/bako/mpython/introPython
hello.1.py
hello.1.py x
1 print("Hello World")
Shell x
Couldn't find the device automatically.
Check the connection (making sure the
device is connected and the port is cor
rect).
To select specific port or another de
vice, please use the "Configure interpr
eter" dialog in the "Tools" menu.
Python 3.8.10 (/usr/bin/python)
>>> %cd /home/bako/mpython
>>> %cd /home/bako/mpython/introPython
>>> %Run hello.1.py
Hello World
>>> |
```



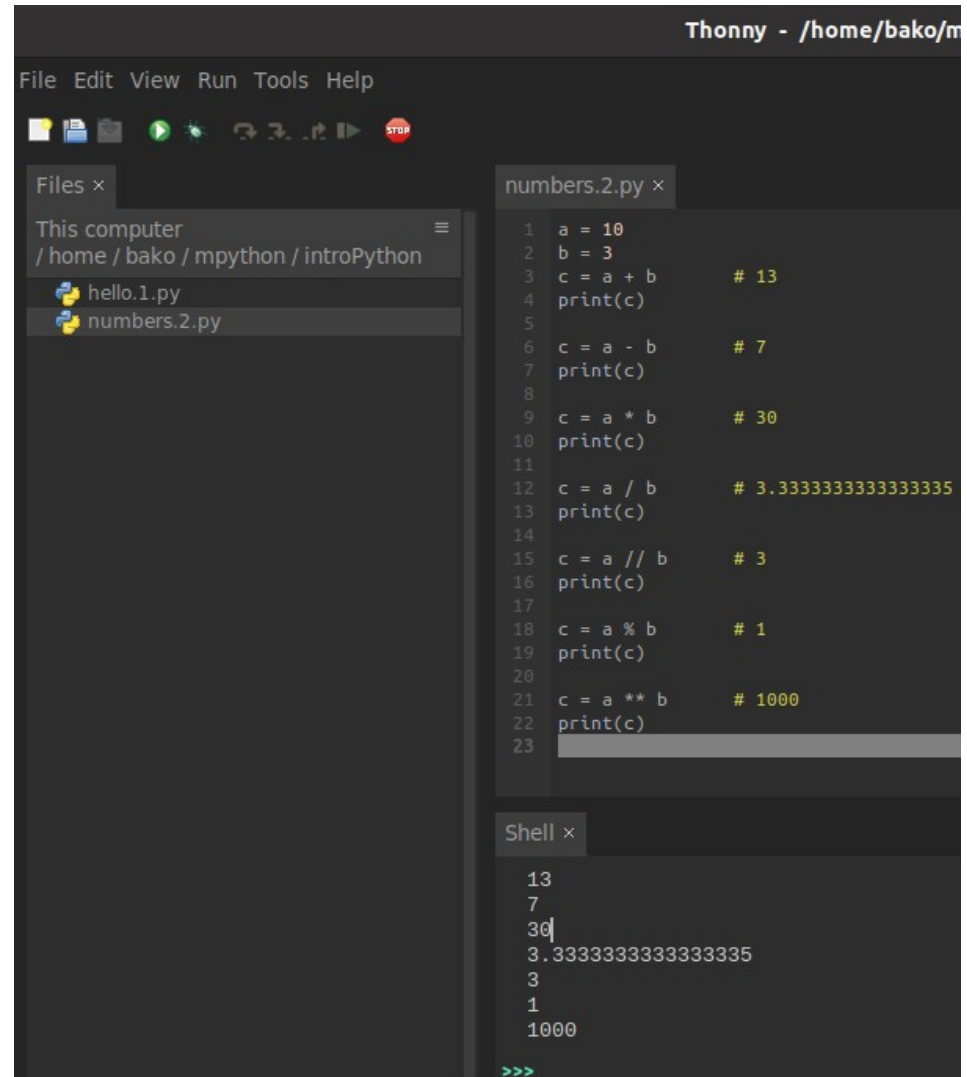
Python – numbers

Python provides for different types of numbers.

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

We have arithmetical operators:

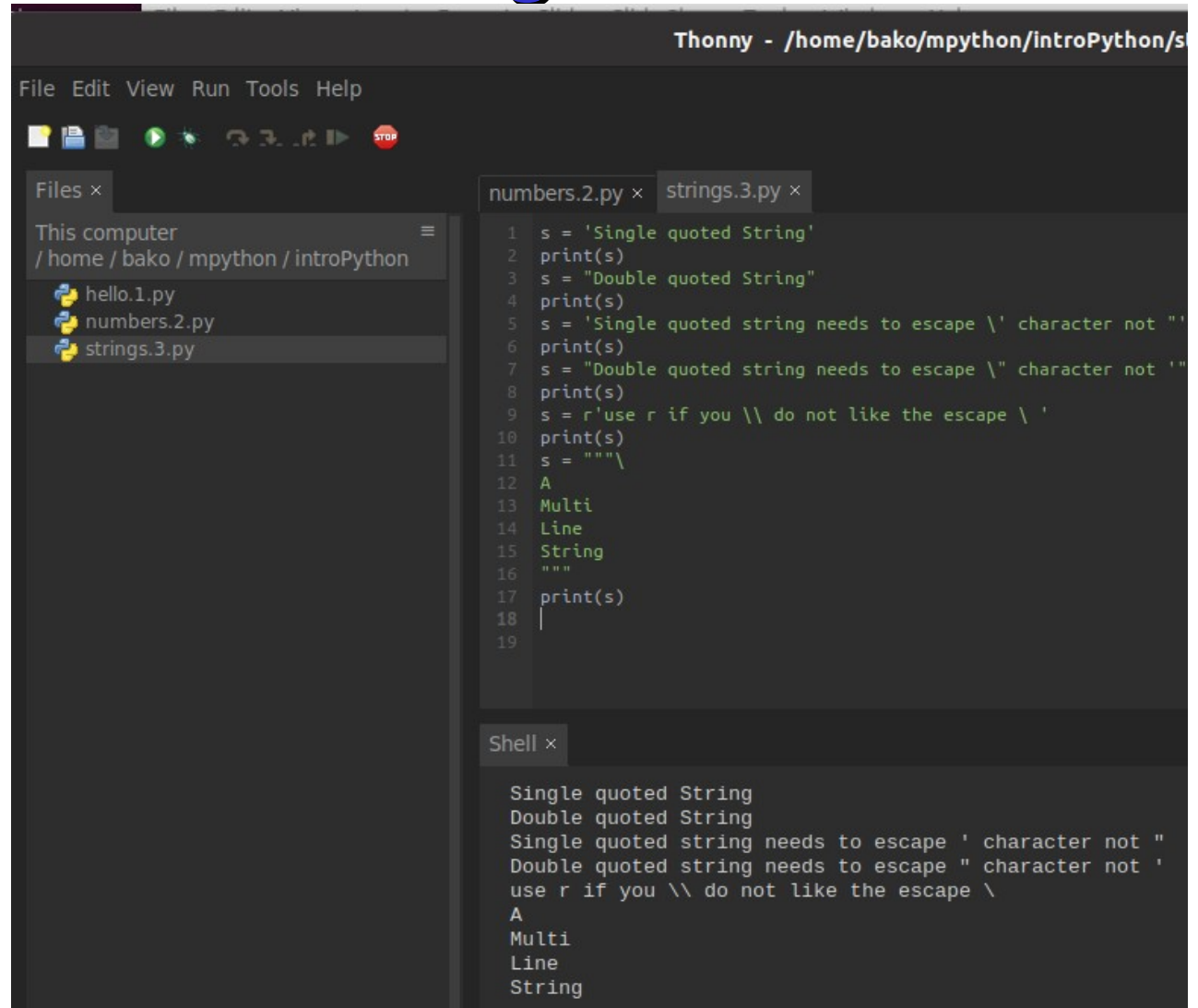
+, **-**, *****, **/**, **//**, **%**, ******



```
Thonny - /home/bako/m
File Edit View Run Tools Help
Files x
This computer
/home/bako/mpython/introPython
hello.1.py
numbers.2.py
numbers.2.py x
1 a = 10
2 b = 3
3 c = a + b # 13
4 print(c)
5
6 c = a - b # 7
7 print(c)
8
9 c = a * b # 30
10 print(c)
11
12 c = a / b # 3.3333333333333335
13 print(c)
14
15 c = a // b # 3
16 print(c)
17
18 c = a % b # 1
19 print(c)
20
21 c = a ** b # 1000
22 print(c)
23
Shell x
13
7
30
3.3333333333333335
3
1
1000
>>>
```

Python – Strings 1

Python Strings

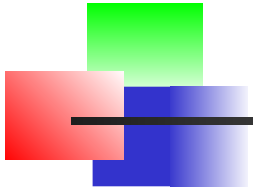


The screenshot shows the Thonny Python IDE interface. The title bar reads "Thonny - /home/bako/mpython/introPython/s". The menu bar includes "File", "Edit", "View", "Run", "Tools", and "Help". The toolbar contains icons for file operations and execution. The "Files" panel on the left shows the current directory: "/home/bako/mpython/introPython", with files "hello.1.py", "numbers.2.py", and "strings.3.py". The "strings.3.py" file is open in the editor, showing the following code:

```
1 s = 'Single quoted String'
2 print(s)
3 s = "Double quoted String"
4 print(s)
5 s = 'Single quoted string needs to escape \' character not "'
6 print(s)
7 s = "Double quoted string needs to escape \" character not '"
8 print(s)
9 s = r'use r if you \\ do not like the escape \' '
10 print(s)
11 s = """\
12 A
13 Multi
14 Line
15 String
16 """
17 print(s)
18 |
19
```

The "Shell" panel at the bottom shows the output of the script:

```
Single quoted String
Double quoted String
Single quoted string needs to escape ' character not "
Double quoted string needs to escape " character not '
use r if you \\ do not like the escape \
A
Multi
Line
String
```



Strings 2

Test the code and change some values:

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

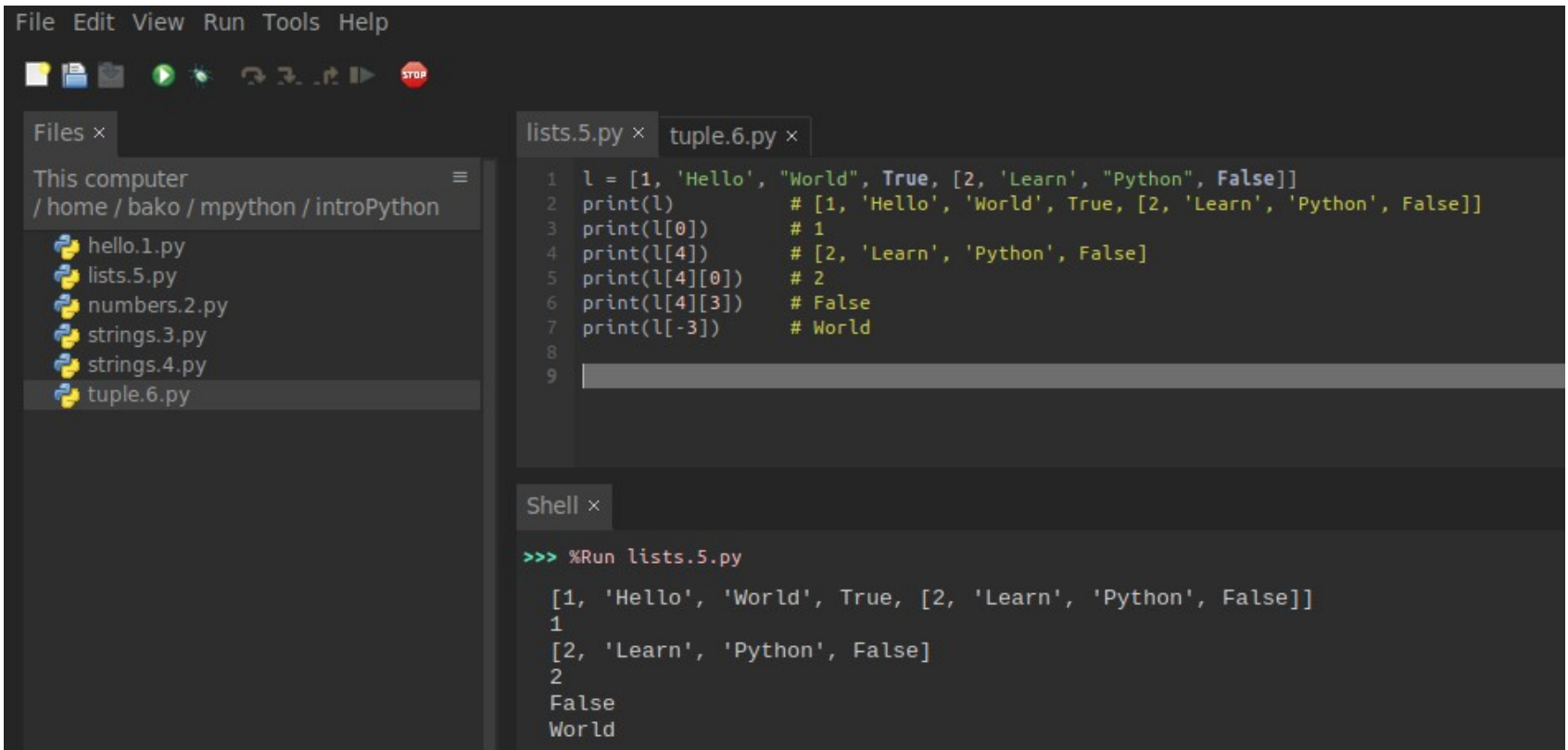
```
#s = s.__add__(" PyThOn")  
s = s + " PyThon"
```

```
File Edit View Run Tools Help  
Files x  
This computer  
/ home / bako / mpython / introPython  
hello.1.py  
numbers.2.py  
strings.3.py  
strings.4.py  
strings.3.py x strings.4.py x  
1 s = "lEaRnInG"  
2 # Append  
3 s = s.__add__(" PyThOn")  
4 #split  
5 print(s.split())  
6 print(s.split(sep="t"))  
7 # Splicing  
8 print(s[:])  
9 print(s[1:])  
10 print(s[1:-1])  
11 print(s[5:-5])  
12 print(s[16:0])  
13 # Casing  
14 print(s.lower())  
15 print(s.upper())  
16 print(s.title())  
17  
Shell x  
>>> %Run strings.4.py  
['lEaRnInG', 'PyThOn']  
['lEaRnInG PyThOn']  
lEaRnInG PyThOn  
EaRnInG PyThOn  
EaRnInG PyThO  
InG P  
  
learning python  
LEARNING PYTHON  
Learning 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.

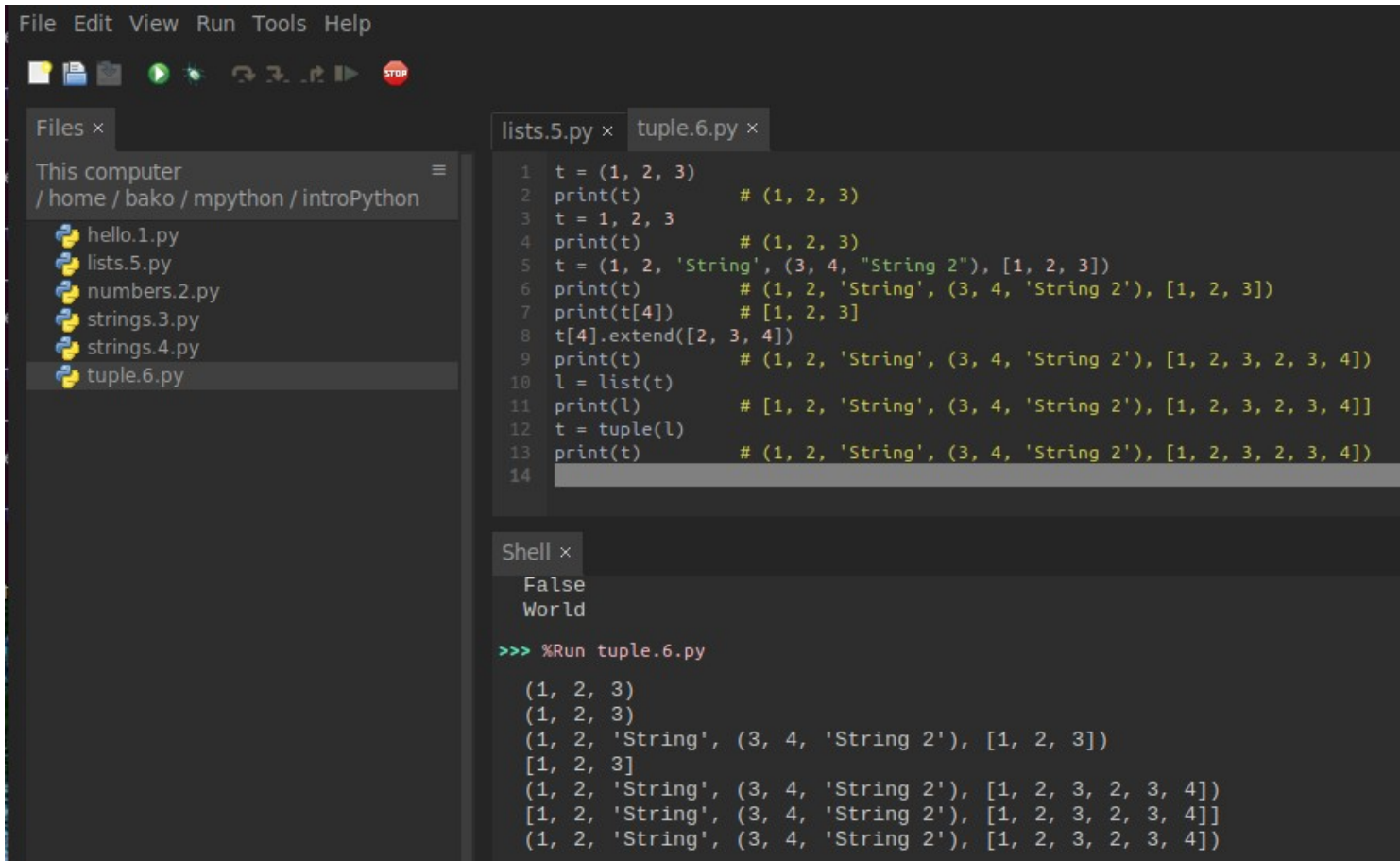


```
File Edit View Run Tools Help
Files x
This computer
/home / bako / mpython / introPython
hello.1.py
lists.5.py
numbers.2.py
strings.3.py
strings.4.py
tuple.6.py
lists.5.py x tuple.6.py x
1 l = [1, 'Hello', "World", True, [2, 'Learn', "Python", False]]
2 print(l) # [1, 'Hello', 'World', True, [2, 'Learn', 'Python', False]]
3 print(l[0]) # 1
4 print(l[4]) # [2, 'Learn', 'Python', False]
5 print(l[4][0]) # 2
6 print(l[4][3]) # False
7 print(l[-3]) # World
8
9
Shell x
>>> %Run lists.5.py
[1, 'Hello', 'World', True, [2, 'Learn', 'Python', False]]
1
[2, 'Learn', 'Python', False]
2
False
World
```



Tuples

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



```
File Edit View Run Tools Help
Files x
This computer
/home / bako / mpython / introPython
hello.1.py
lists.5.py
numbers.2.py
strings.3.py
strings.4.py
tuple.6.py

lists.5.py x tuple.6.py x
1 t = (1, 2, 3)
2 print(t)          # (1, 2, 3)
3 t = 1, 2, 3
4 print(t)          # (1, 2, 3)
5 t = (1, 2, 'String', (3, 4, "String 2"), [1, 2, 3])
6 print(t)          # (1, 2, 'String', (3, 4, 'String 2'), [1, 2, 3])
7 print(t[4])       # [1, 2, 3]
8 t[4].extend([2, 3, 4])
9 print(t)          # (1, 2, 'String', (3, 4, 'String 2'), [1, 2, 3, 2, 3, 4])
10 l = list(t)
11 print(l)          # [1, 2, 'String', (3, 4, 'String 2'), [1, 2, 3, 2, 3, 4]]
12 t = tuple(l)
13 print(t)         # (1, 2, 'String', (3, 4, 'String 2'), [1, 2, 3, 2, 3, 4])
14

Shell x
False
World

>>> %Run tuple.6.py

(1, 2, 3)
(1, 2, 3)
(1, 2, 'String', (3, 4, 'String 2'), [1, 2, 3])
[1, 2, 3]
(1, 2, 'String', (3, 4, 'String 2'), [1, 2, 3, 2, 3, 4])
[1, 2, 'String', (3, 4, 'String 2'), [1, 2, 3, 2, 3, 4]]
(1, 2, 'String', (3, 4, 'String 2'), [1, 2, 3, 2, 3, 4])
```



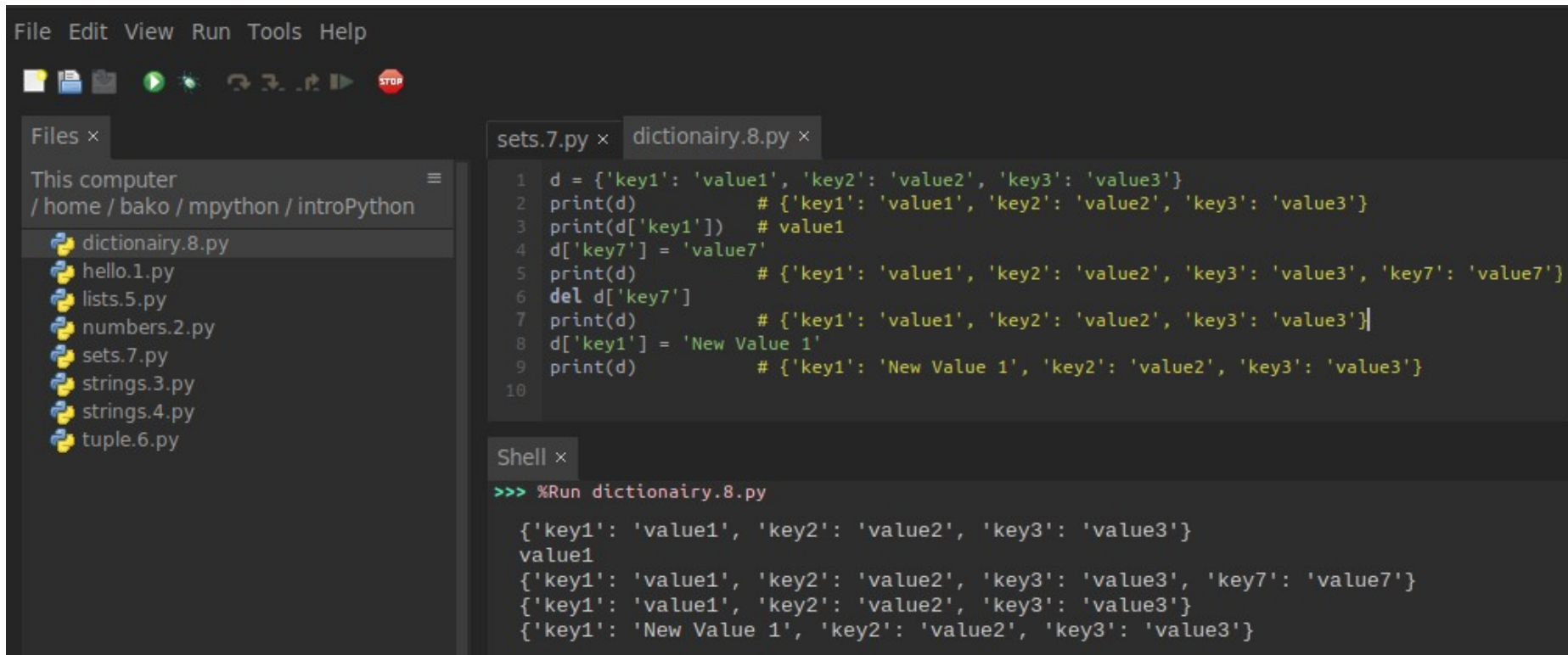
Sets

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

```
File Edit View Run Tools Help
Files x
This computer
/home / bako / mpython / introPython
hello.1.py
lists.5.py
numbers.2.py
sets.7.py
strings.3.py
strings.4.py
tuple.6.py
lists.5.py x tuple.6.py x sets.7.py x
1 s = {1, "String", ('1', 'Tuple'), 1, 2}
2 print(s) # {1, 'String', 2, ('1', 'Tuple')}
3 s.add(1)
4 print(s) # {1, 'String', 2, ('1', 'Tuple')}
5 s.add(3)
6 print(s) # {1, 'String', 3, 2, ('1', 'Tuple')}
7 s.remove(1)
8 print(s) # {'String', 3, 2, ('1', 'Tuple')}
9 # remove throws an exception and discard just ignores any attempt
10 s.discard("Strings")
11 print(s) # {'String', 3, 2, ('1', 'Tuple')}
12 s.pop()
13 print(s) # {3, 2, ('1', 'Tuple')}
14 s.clear()
15 print(s) # set()
Shell x
>>> %Run sets.7.py
{1, 2, 'String', ('1', 'Tuple')}
{1, 2, 'String', ('1', 'Tuple')}
{1, 2, 3, 'String', ('1', 'Tuple')}
{2, 3, 'String', ('1', 'Tuple')}
{2, 3, 'String', ('1', 'Tuple')}
{3, 'String', ('1', 'Tuple')}
set()
```


Dictionaries

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

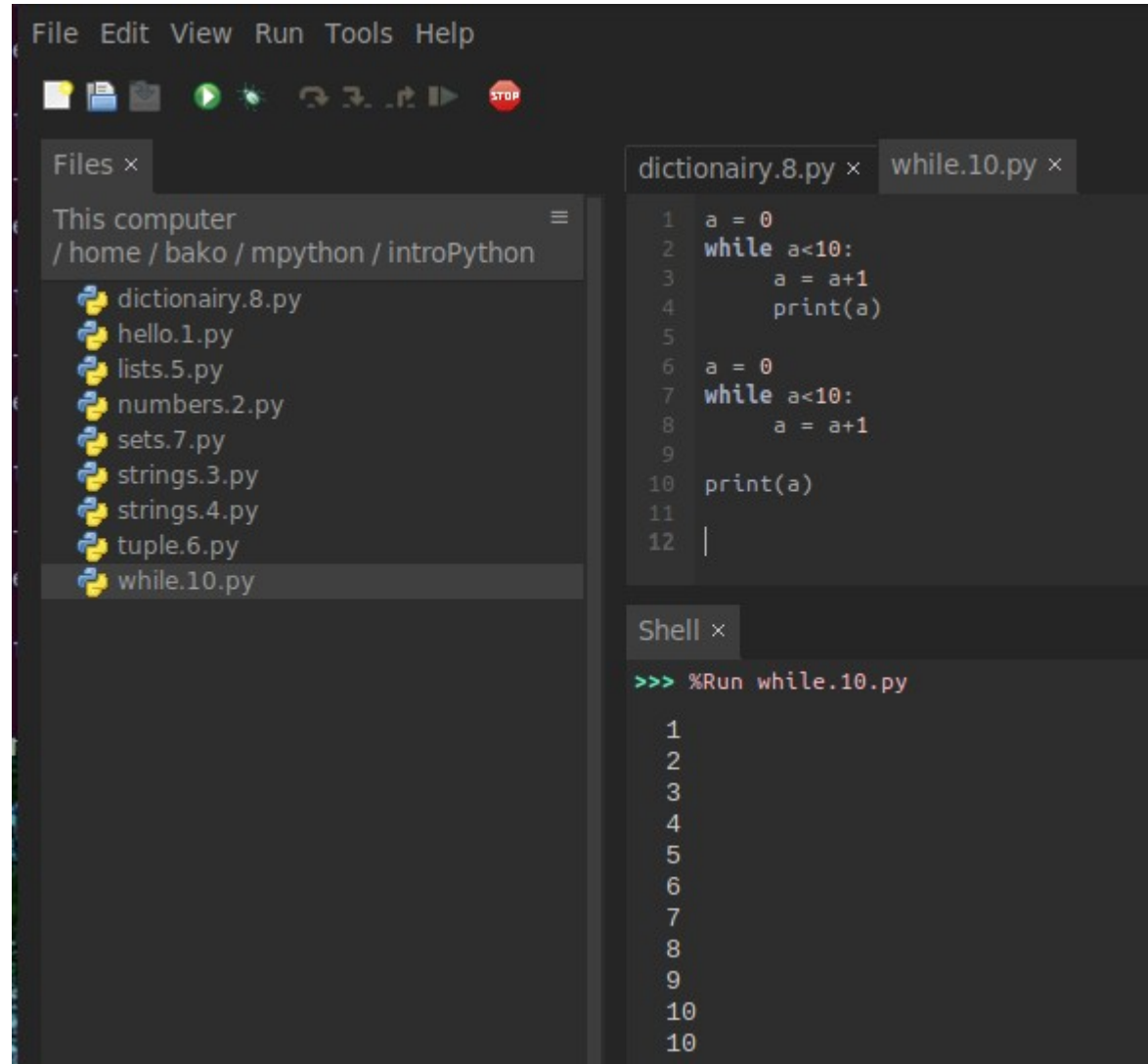


```
File Edit View Run Tools Help
dictionary.8.py
sets.7.py x dictionary.8.py x
1 d = {'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
2 print(d) # {'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
3 print(d['key1']) # value1
4 d['key7'] = 'value7'
5 print(d) # {'key1': 'value1', 'key2': 'value2', 'key3': 'value3', 'key7': 'value7'}
6 del d['key7']
7 print(d) # {'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
8 d['key1'] = 'New Value 1'
9 print(d) # {'key1': 'New Value 1', 'key2': 'value2', 'key3': 'value3'}
10
Shell x
>>> %Run dictionary.8.py
{'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
value1
{'key1': 'value1', 'key2': 'value2', 'key3': 'value3', 'key7': 'value7'}
{'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
{'key1': 'New Value 1', 'key2': 'value2', 'key3': 'value3'}
```


Code flow – while loop

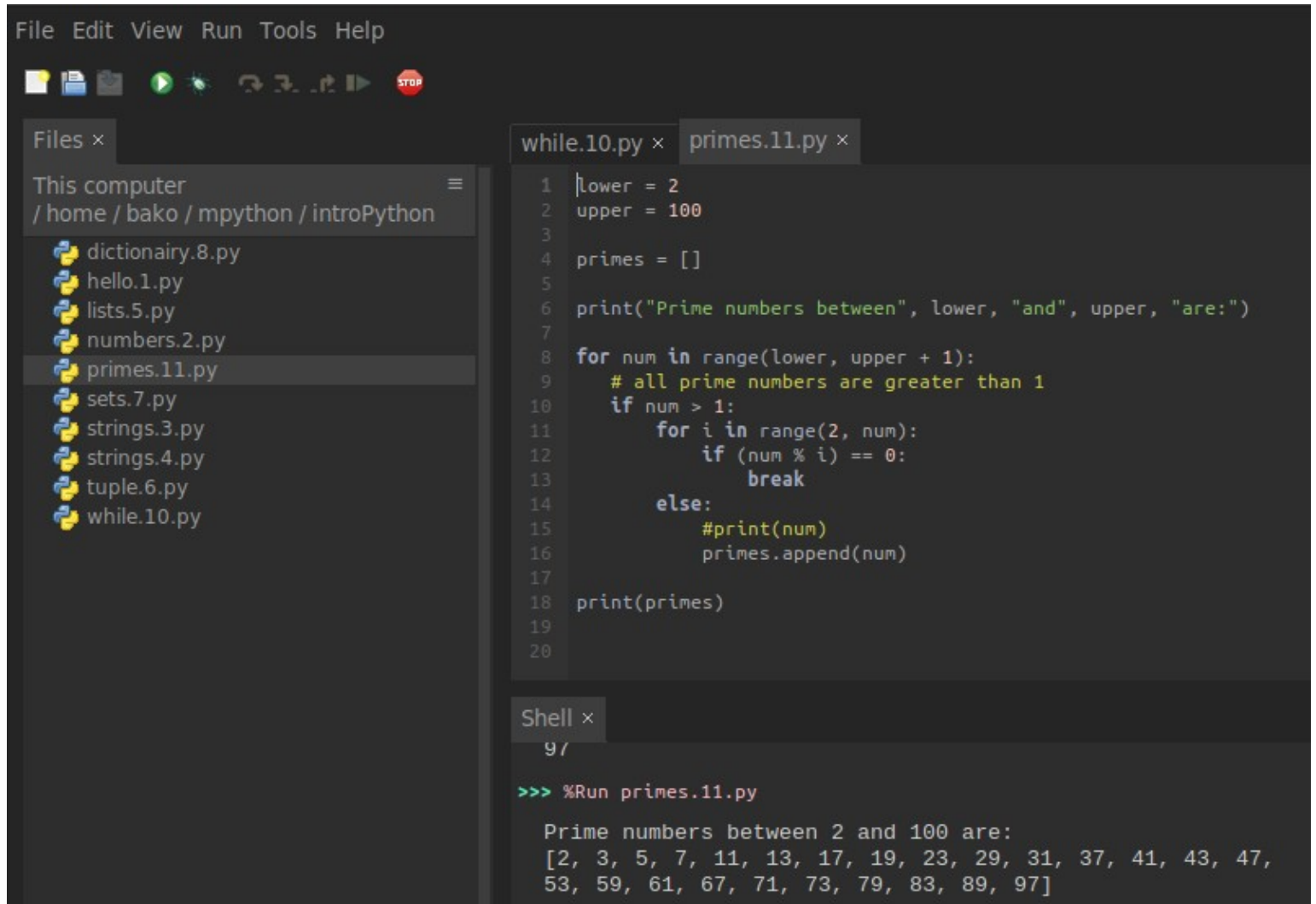
Programming is all about **data and decisions**.

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



```
File Edit View Run Tools Help
dictionary.8.py x while.10.py x
1 a = 0
2 while a<10:
3     a = a+1
4     print(a)
5
6 a = 0
7 while a<10:
8     a = a+1
9
10 print(a)
11
12 |
Shell x
>>> %Run while.10.py
1
2
3
4
5
6
7
8
9
10
10
```

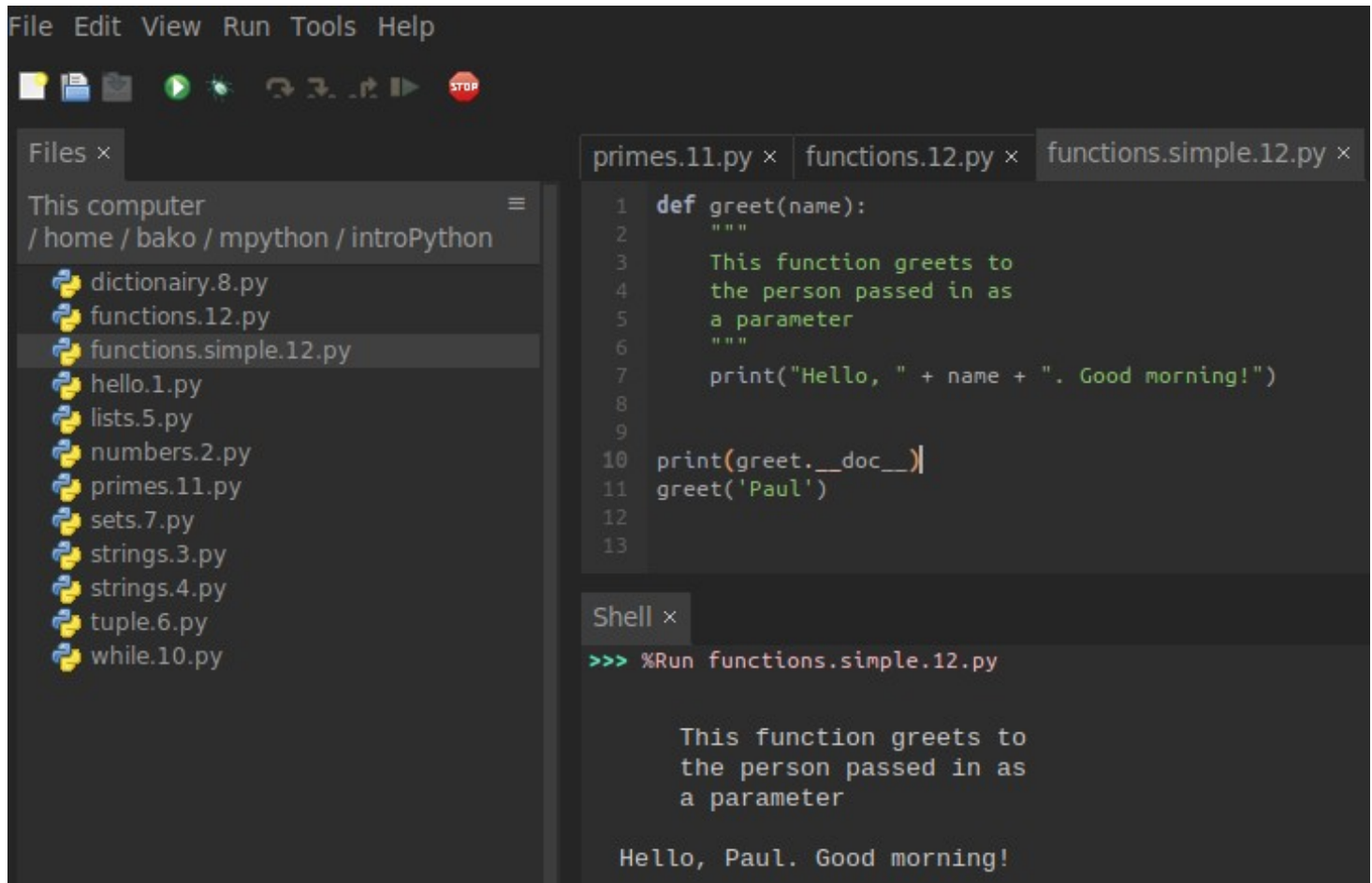
Primes (for .. in range())



```
File Edit View Run Tools Help
[Icons]
Files x
This computer
/home / bako / mpython / introPython
dictionary.8.py
hello.1.py
lists.5.py
numbers.2.py
primes.11.py
sets.7.py
strings.3.py
strings.4.py
tuple.6.py
while.10.py
while.10.py x primes.11.py x
1 lower = 2
2 upper = 100
3
4 primes = []
5
6 print("Prime numbers between", lower, "and", upper, "are:")
7
8 for num in range(lower, upper + 1):
9     # all prime numbers are greater than 1
10    if num > 1:
11        for i in range(2, num):
12            if (num % i) == 0:
13                break
14        else:
15            #print(num)
16            primes.append(num)
17
18 print(primes)
19
20
Shell x
9/
>>> %Run primes.11.py
Prime numbers between 2 and 100 are:
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Functions, modules and packages

Simple function definition and call:



The screenshot shows a Python IDE with a dark theme. The menu bar includes File, Edit, View, Run, Tools, and Help. The toolbar contains icons for file operations and execution. The file explorer on the left shows a directory structure under /home/bako/mpython/introPython, listing various Python files. The main editor window displays the code for functions.12.py, which defines a greet function and calls it with the name 'Paul'. The shell window shows the execution of the script, displaying the function's docstring and the output of the greet function.

```
File Edit View Run Tools Help
Files x
This computer
/home/bako/mpython/introPython
dictionary.8.py
functions.12.py
functions.simple.12.py
hello.1.py
lists.5.py
numbers.2.py
primes.11.py
sets.7.py
strings.3.py
strings.4.py
tuple.6.py
while.10.py
primes.11.py x functions.12.py x functions.simple.12.py x
1 def greet(name):
2     """
3     This function greets to
4     the person passed in as
5     a parameter
6     """
7     print("Hello, " + name + ". Good morning!")
8
9
10 print(greet.__doc__)
11 greet('Paul')
12
13
Shell x
>>> %Run functions.simple.12.py

This function greets to
the person passed in as
a parameter

Hello, Paul. Good morning!
```

Complex function

```
primes.11.py × functions.12.py × functions.simple.12.py ×
1
2 def getFunction(full=True):
3     'Outer Function'
4     print(getFunction.__doc__)
5
6     def p(frm=0, to=1, step=1):
7         'Inner Function'
8         print(p.__doc__)
9         return (x ** 3 for x in range(frm, to, step))
10
11    if (full):
12        return p
13    else:
14        return lambda frm = 0, to = 1, step = 1: (x ** 3 \
15            for x in range(frm, to, step))
16
17    print(__doc__)
18    t = getFunction()
19    print("Check the elaborate function")
20    for v in t(step=1, to=10):
21        print(v)
22    t = getFunction(False)
23    print("Check the lambda function")
24    for v in t(1, 5):
25        print(v)
```

```
17 print(__doc__)
18 t = getFunction()
```

Shell ×

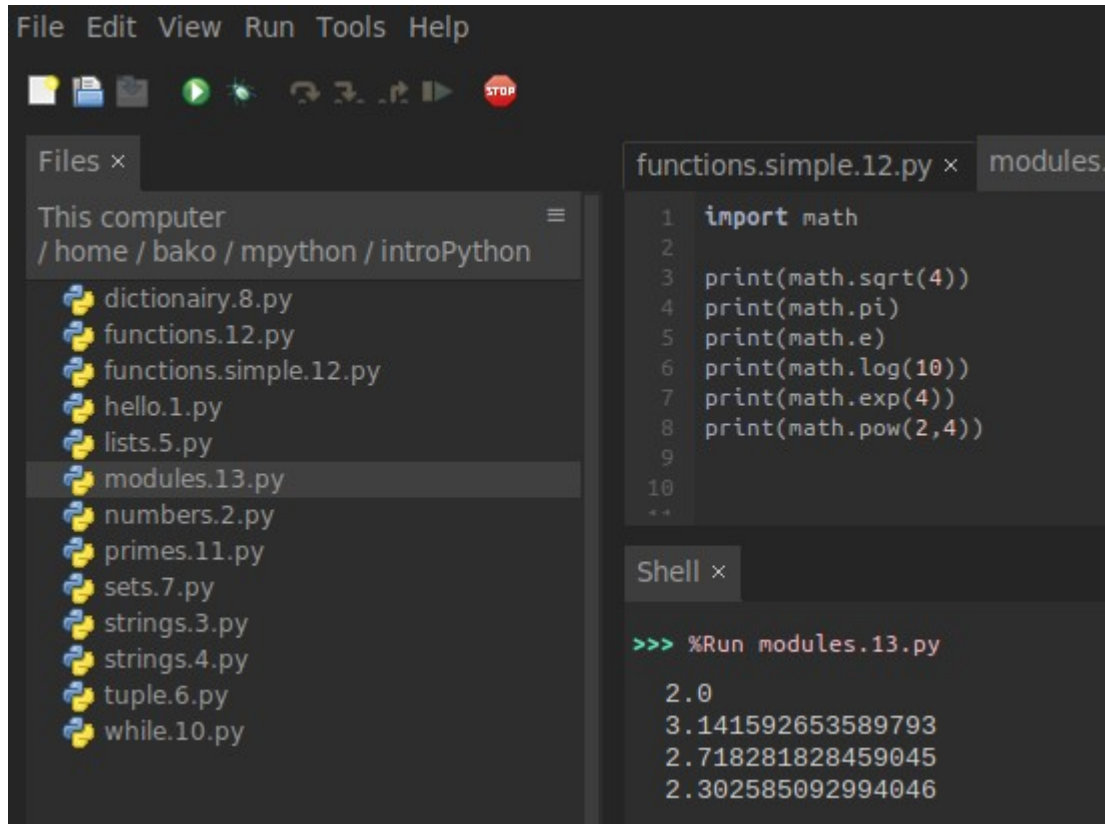
```
hello, world. good morning.
```

```
>>> %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



```
File Edit View Run Tools Help
```

Files x

This computer
/ home / bako / mpython / introPython

- dictionary.8.py
- functions.12.py
- functions.simple.12.py
- hello.1.py
- lists.5.py
- modules.13.py
- numbers.2.py
- primes.11.py
- sets.7.py
- strings.3.py
- strings.4.py
- tuple.6.py
- while.10.py

```
functions.simple.12.py x modules.
```

```
1 import math
2
3 print(math.sqrt(4))
4 print(math.pi)
5 print(math.e)
6 print(math.log(10))
7 print(math.exp(4))
8 print(math.pow(2,4))
9
10
**
```

Shell x

```
>>> %Run modules.13.py
2.0
3.141592653589793
2.718281828459045
2.302585092994046
```



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:

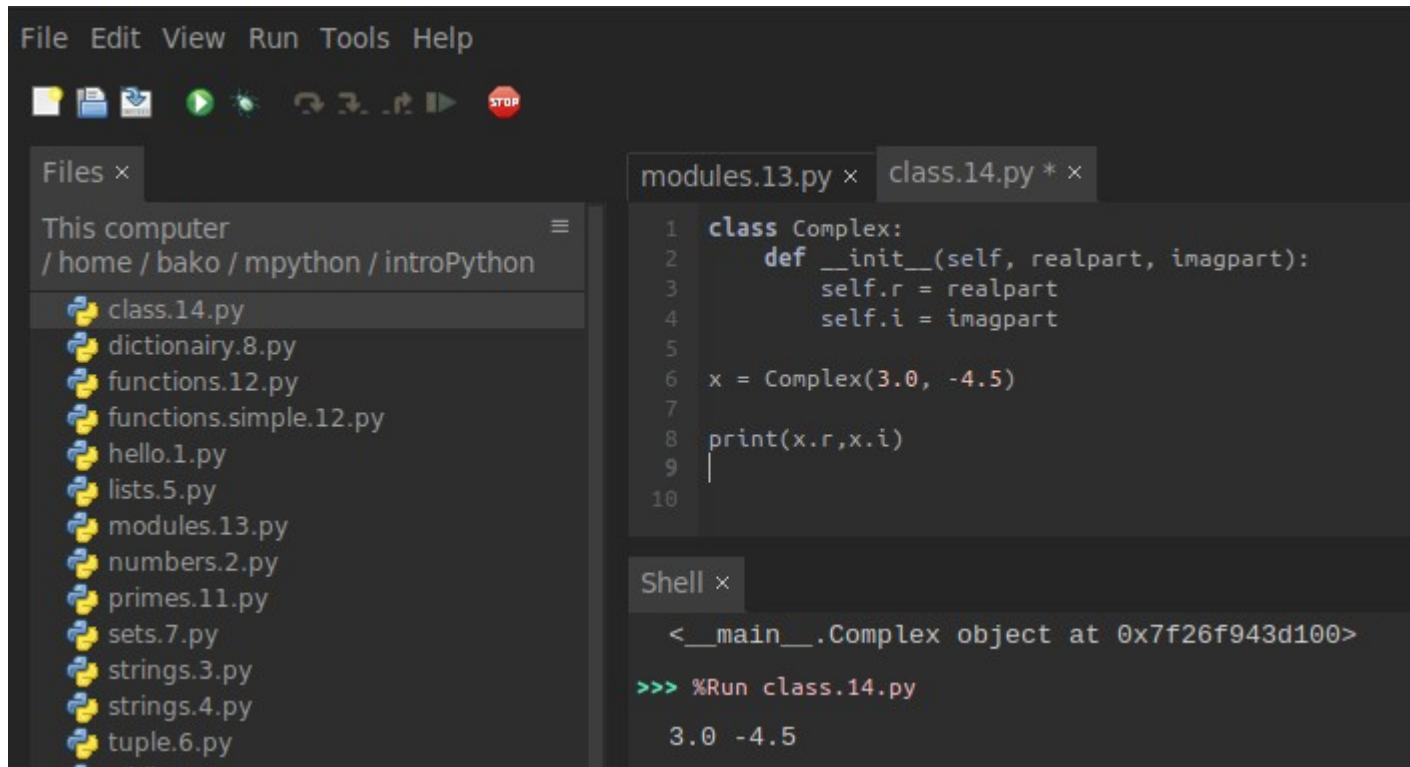
```
def __init__(self):  
    self.data = []
```

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,



The screenshot shows a Python IDE with a dark theme. The left sidebar displays a file explorer for the directory `/home/bako/mpython/introPython`, listing several Python files, with `class.14.py` selected. The main editor window shows the code in `class.14.py`:

```
1 class Complex:
2     def __init__(self, realpart, imagpart):
3         self.r = realpart
4         self.i = imagpart
5
6 x = Complex(3.0, -4.5)
7
8 print(x.r,x.i)
9
10
```

Below the editor is a Shell window showing the output of running the script:

```
<__main__.Complex object at 0x7f26f943d100>
>>> %Run class.14.py
3.0 -4.5
```



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 ×

```
1 class MyClass:
2     """A simple example class"""
3     i = 12345
4
5     def f(self):
6         return 'hello world'
7
8 x=MyClass()
9
10 x.counter = 1
11 x.toto =3
12
13 while x.counter < 10:
14     x.counter = x.counter * 2
15
16 print(x.counter)
17 print(x.f())
18 print(x.i)
19 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.15.py x class.method.16.py x
1 import time
2
3 class MyClass:
4     """A simple example class"""
5     i = 12345
6
7     def f(self):
8         return 'hello world'
9
10
11 x = MyClass()
12 xf = x.f
13 while True:
14     print(xf())
15     time.sleep(2)
16
17 |
```

```
Shell x
>>> %Run class.method.16.py
hello world
hello world
hello world
hello world
hello world
hello world
```



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 * ×
1 class Dog:
2     kind = 'canine'
3     # class variable shared by all instances
4
5     def __init__(self, name):
6         self.name = name
7         # instance variable unique to each instance
8
9     d = Dog('Fido')
10    e = Dog('Buddy')
11    print(d.kind)
12    print(e.kind)           # shared by all dogs
13    print(d.name)          # unique to d
14    print(e.name)          # unique to e
15
16
17
```

```
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 * x  class.inst.var.init.py * x
1  class Dog:
2
3      def __init__(self, name):
4          self.name = name
5          self.tricks = []
6          # creates a new empty list for each dog
7
8      def add_trick(self, trick):
9          self.tricks.append(trick)
10
11  d = Dog('Fido')
12  e = Dog('Buddy')
13  d.add_trick('roll over')
14  e.add_trick('play dead')
15  print(d.tricks)
16  print(e.tricks)

Shell x
>>> %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