Introduction to Python Programming

Tran Giang Son, tran-giang.son@usth.edu.vn

ICT Department, USTH

Python

What

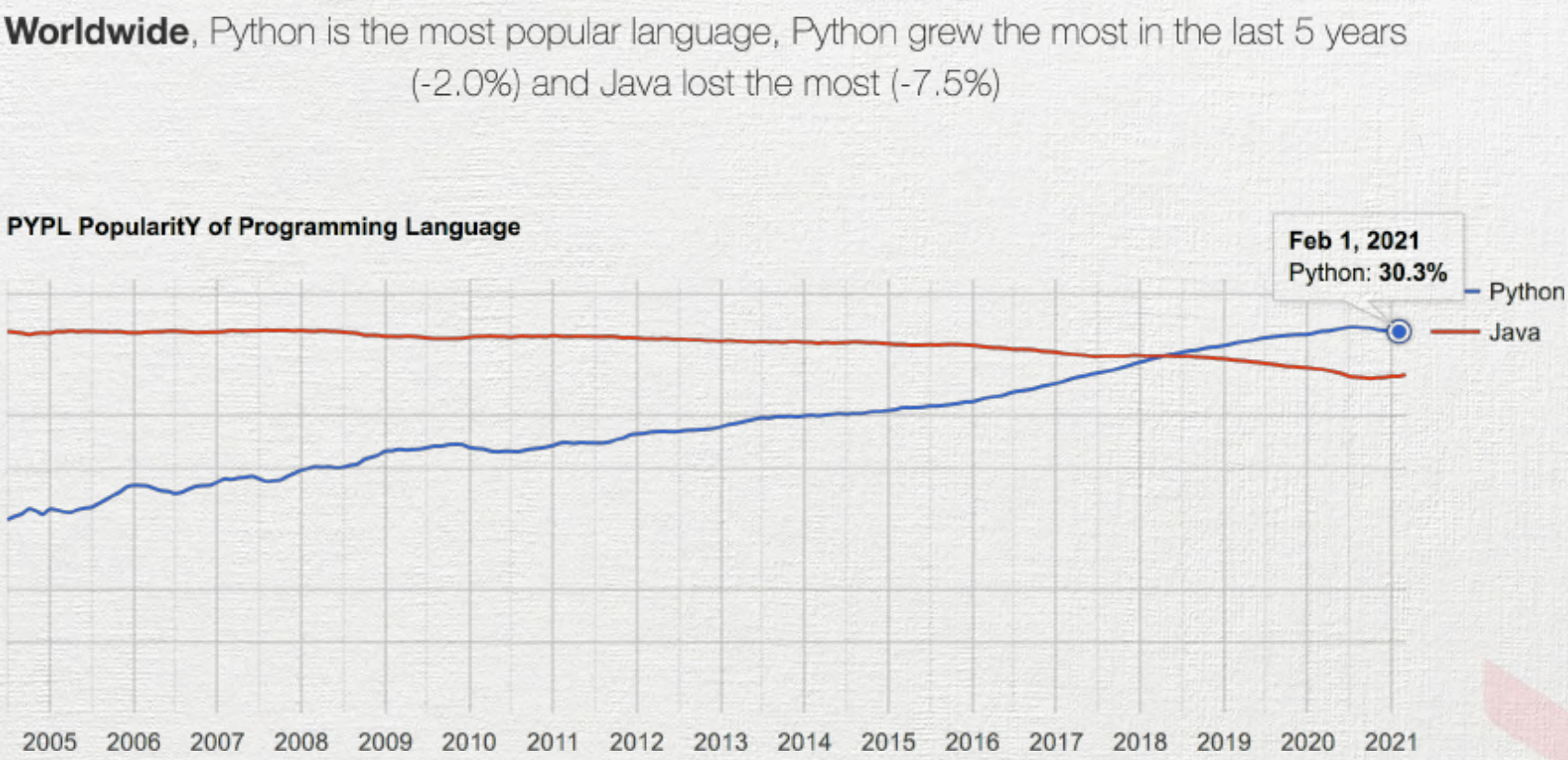
- Programming language
  - Open-source
  - Interpreted
  - High-level
  - General-purpose
- Code readability
- Object Oriented Programming

What

Worldwide, Mar 2021 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	30.17 %	-0.2 %
2		Java	17.18 %	-1.2 %
3		JavaScript	8.21 %	+0.2 %
4		C#	6.76 %	-0.6 %
5	↑	C/C++	6.71 %	+0.8 %
6	↓	PHP	6.13 %	+0.0 %
7		R	3.81 %	+0.0 %
8		Objective-C	3.56 %	+1.1 %
9		Swift	1.82 %	-0.4 %

What


Worldwide, Python is the most popular language, Python grew the most in the last 5 years (-2.0%) and Java lost the most (-7.5%)
PYPL PopularitY of Programming Language
Feb 1, 2021
Python: 30.3%
Python
Java
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021

Source

Why

- Easy
- Flexible
- Readability
- Projects
- Jobs

Why: Flexible

- Script
- Backend
- Machine Learning, Deep Learning
- Apps
  - Mobile
  - Desktop
  - Web

Why: Readability

- Indents ftw!
- Line breaks ftw!

Why: Projects

- Full list
  - Browser
  - Youtube-dl
  - Music Player
  - Video Editor
  - Bittorent client
  - Text Editor

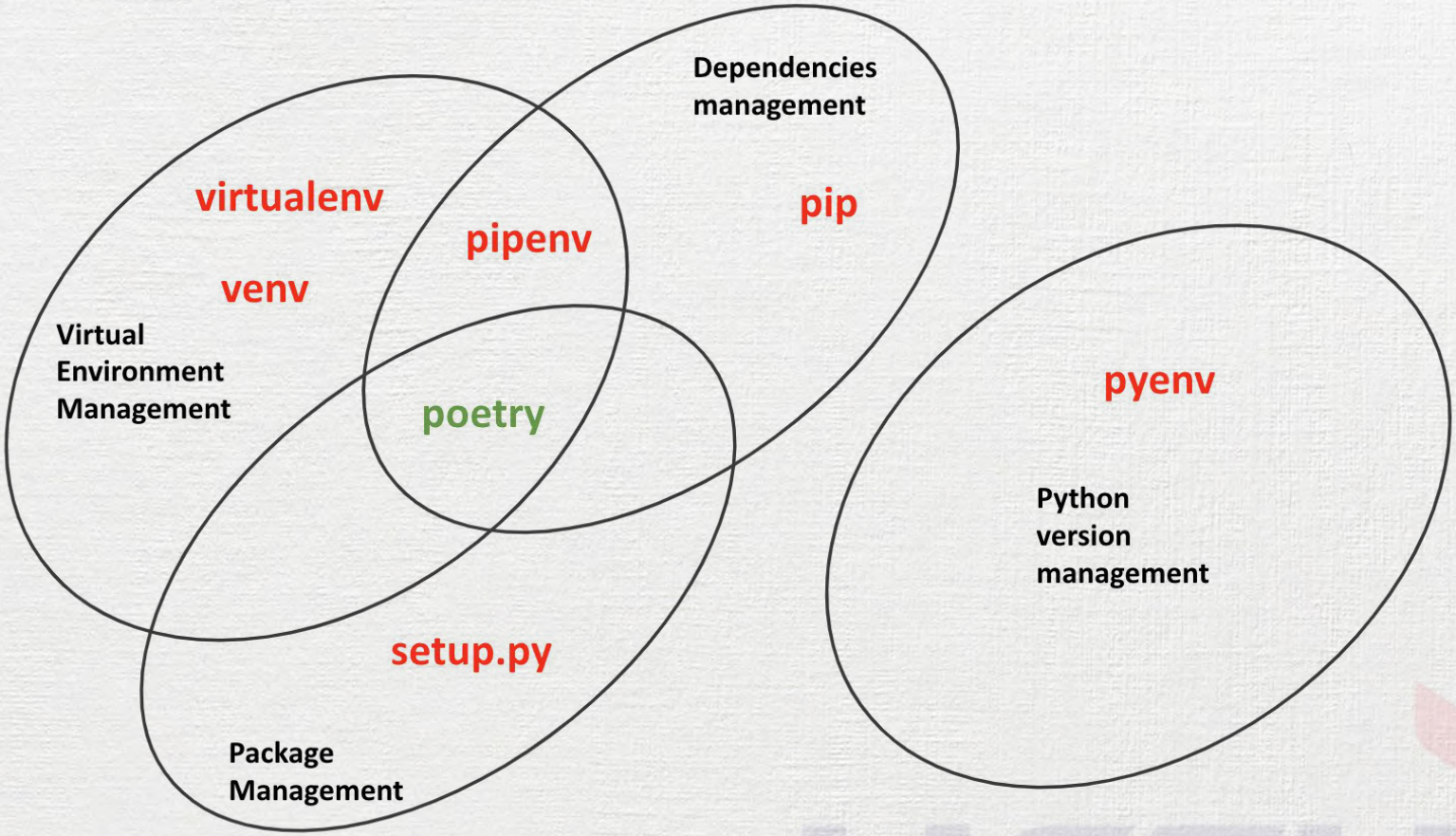
Why NOT?

- Intepretation, not compilation
- Slow
  - GIL...
- Not optimized for
  - Mobile
  - Web client

Python

- Python
  - Python 2.x: discontinued
  - Python 3.x
- Download

Versions, Environments, Packages



Versions, Environments, Packages

- pyenv: manage Python versions
- virtualenv: isolate Python environments
- pip:
  - install Python packages
  - from Python Package Index
- conda:
  - isolate environments
  - install packages
  - miniconda: minimal installer for conda
  - anaconda: miniconda + bunch of pre-installed packages

IDE

- Full fledged IDEs
  - PyCharm
  - Visual Studio Code
- Better live code
  - Jupyter notebooks
  - Spyder
- Simplicity: any text editor
  - Atom
  - Sublime Text
  - Notepad
  - vi/vim/nano...

Jupyter Notebook

- Written in Python
- Fork of IPython
- Open-source *Web app*
  - live code
  - equations
  - Computational output - visualizations
  - explanatory text
- Popular for Data Science
- JupyterLab

Jupyter Notebook

- Main components
  - IPython
  - ØMQ
  - Tornado (web server)
  - jQuery
  - Bootstrap (front-end framework)
  - MathJax

Jupyter Notebook

- Install
  - pip

    ```
    pip install notebook
    ```

  * conda

    ```
    conda install -c conda-forge notebook
    ```

- Launch

  ```
  jupyter notebook
  ```

The Python Language

Tran Giang Son, tran-giang.son@usth.edu.vn

ICT Department, USTH

Expressions

Interactive vs Script

- Interactive
  - Type command
  - Execute
  - Wait for response
- Script
  - All-in-one long sequences of statements
  - `python script.py`
  - Shebang `#!` works

Constants

- What
  - Fixed values
  - Value does not change over time
- Examples
  - Numeric constants
  - String constants
    - Single quotes '
    - Double quotes "
- Why: everywhere

Constants

- How

```
>>> print(123)
123
>>> print(98.6)
98.6
>>> print('Hello world')
Hello world
```

Variables

- What
  - Named place in the memory to store data
  - Access it later using name
  - Modifiable at runtime
- Why: store temporary changable values

Variables

- Variable name rules
  - Letters, numbers, or underscores
  - CaSe sEnSiTiVe
  - Not allowed: starting with number
- Examples
  - Good: `spam`, `eggs`, `spam23`, `_speed`
  - Bad: `23spam`, `#sign`, `var.12`
  - Different: `spam`, `Spam`, `SPAM`

Variables

- Reserved words

```
and del for is raise assert elif
from lambda return break else
global not try class except if or while
continue exec import pass
yield def finally in print
```

Statements

- What: combination of operator and its operand(s)
  - Operator: symbol indicating a calculation
  - One or more operands
- Numeric expression
  - `+` Addition
  - `-` Subtraction
  - `*` Multiplication
  - `/` Division
  - `**` Power
  - `%` Remainder

Statements

- Numeric expression

```
>>> x = 2
>>> x = x + 2
>>> print(x)
4
>>> y = 440 * 12
>>> print(y)
5280
>>> z = y / 1000
>>> print(z)
5
```

```
>>> j = 23
>>> k = j % 5
>>> print(k)
3
>>> print(4 ** 3)
64
```

Statements

- Mixing Integer and Floats: convert everything to float.

```
>>> print(99 / 100)
0
>>> print(99 / 100.0)
0.99
>>> print(99.0 / 100)
0.99
>>> print(1 + 2 * 3 / 4.0 - 5)
-2.5
>>>
```

Data Types

What

- Variables, literals, and constants have a “data type”

Type	Examples
Integer	0, 12, 5, -5
Float	4.5, 3.99, 0.1
String	“Hi”, “Hello”, “Hi there!”"
Boolean	True, False
List	[“hi”, “there”, “you”]
Tuple	(4, 2, 7, 3)

What: Boolean

- `bool`
- 2 possible values: `True`, `False`

What: Integer

- `int`
- Unbounded.

```
>>> i=10**100
>>> type(i)
<class 'int'>
>>> i
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

What: Float

- `float`
- Digits and Exponents

```
>>> 2.5
>>> 2e4
>>> 0.00001
>>> 1000020000300004
```

What: Strings

- `str`
- Series of Unicode characters
- Character: String of length 1
- Enclosed by a pair of single or double quotes
- Multiline: triple quote
  - `'''`
  - `"""`

```
>>> s="""This is
... a Multiline string
... for example"""
>>> s
'This is\na Multiline string\nfor example'
```

Dynamically typing

- Dynamically typed variables
- Types are automatically managed

C, Java

```
int a;
float b;
a = 5;
b = 0.43;
```

Python

```
a = 5
a = 0.43
a = "Hello"
```

Number Conversion

- int()
- float()

```
>>> print(float(99) / 100)
0.99
>>> i = 42
>>> type(i)
<class 'int'>
>>> f = float(i)
>>> print(f)
42.0
>>> type(f)
<class 'float'>
>>> print(1 + 2 * float(3) / 4 - 5)
-2.5
>>>
```

Number Conversion

- Also works with strings!

```
>>> sval = '123'
>>> type(sval)
<class 'str'>
>>> print(sval + 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module
TypeError: can only concatenate str
>>> ival = int(sval)
>>> type(ival)
<class 'int'>
>>> print(ival + 1)
124
>>> nsv = 'hello bob'
>>> niv = int(nsv)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module
```

String Operators

- Some operators apply to strings
  - `+` concatenation
  - `*` multiple concatenation
  - `in`, `not in` contains/not contains

```
>>> print('abc' + '123')
abc123
>>> print('Hi' * 5)
HiHiHiHiHi
>>> "US" in "AmongUS"
True
>>> "us" not in "AmongUS"
True
```

String Operators

- Substring: `string[index:end:step]`
- `index`, `end`
  - `>=0`: start from beginning of string
  - `<`: start from end of string
  - Can be omitted

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+
|-6 |-5 |-4 |-3 |-2 |-1 |
+---+---+---+---+---+---+
```

- `step`: How many letters to skip

String Operators

- `string[index:end:step]`

```
>>> s = "Advanced Programming with Python"
```

```
>>> s[9]
'P'
>>> s[9:20]
'Programming'
>>> s[9:20:2]
'Pormig'
```

```
>>> s[:20]
'Advanced Programming'
>>> s[9:]
'Programming with Python'
>>> s[-6:-4]
'Py'
>>> s[-6:]
'Python'
```

String Formats

- Similar to C's `printf()`
- Previously, in pre-3.6 Python

```
print("Greeting, {}. You are {}".format(name, age))
```

- From Python 3.6 onward: f-string, or formatted string literals

```
print(f"Greeting, {name}. You are {age}")
```

Conditions

Indentation Rules

- *Increase* indent after an if statement or for statement (after :)
  - Equivalent to C, Java's {
- *Maintain* indent to indicate the scope of the block
  - Which lines are affected by the if/for
- *Reduce* indent to *back* to the level of the if statement or for statement to indicate the end of the block
  - Equivalent to C, Java's }
- Blank lines are ignored - they do not affect indentation
- Comments on a line by themselves are ignored w.r.t. indentation

if - else - if - else

```
x = 21
if x < 10:
    print('Smaller than 10')
elif x < 20:
    print('Smaller than 20')
else:
    print('Bigger than 20')
print('End')
```

Functions

What & Why

- Group of related statements performing a specific task
- Break programs into small chunks
- Better code organization
- Code reusable

How

- Definition
  - Function Name
  - Parentheses
  - Arguments

```
def function_name(arguments):
    """docstring"""
    statement1
    statement2
    ...
```

- Call

```
function_name("a value")
```

Examples

```
def greet(name):
    """
    This function greets to
    the person passed in as
    a parameter
    """
    print("Hello, " + name + ". Good morning!")

greet("Emmanuel Macron")
```

Examples

- `len(arg)`: number of elements in `arg`
- `print(args)`: write `args` to `stdout`
- `input(prompt)`: `print(prompt)`, wait and read user input from `stdin`, return the entered string

Collections

What

- Multiple objects are grouped together
- Main types
  - Sets
  - Sequences
  - Maps
  - Streams

Set

- Unordered collection of items
- No duplication
- Operators
  - `s1.isdisjoint(s2)`: no common element
  - `s1 <= s2`, `s1.issubset(s2)`: $s1 \subseteq s2$
  - `s1 >= s2`, `s1.issuperset(s2)`: $s1 \supseteq s2$
  - `s3 = s1 | s2`, `s3 = s1.union(s2)`: $s3 = s1 \cup s2$
  - `s3 = s1 & s2`, `s3 = s1.intersection(s2)`: $s3 = s1 \cap s2$
  - `s3 = s1 - s2`, `s3 = s1.difference(s2)`: $s3 = s1 \setminus s2$

Sequences

- Ordered collection of items
- Can have duplications
- Positioned access
- Slicing similar to strings
  - `seq[start:end:step]`
- Implementations
  - `list`
  - `tuple`
  - `range`
- Others:
  - `str`

Lists

- Mutable sequence
  - Values can be changed later
- Flexible, widely used
- Comma separated declaration

```
>>> names = [ "ICT", "ict" ]
```

Lists

```
>>> names = [ "ICT", "ict" ]
```

- `+` append elements at the end, same or `.extend()`

```
>>> names += ["Ict"]
>>> names
['ICT', 'ict', 'Ict']
```

- `=` replaces single value

```
>>> names[1] = "I See Tea"
>>> names
['ICT', 'I See Tea', 'Ict']
```

Lists

```
>>> names
['ICT', 'I See Tea', 'Ict']
```

- = replaces bunch of values

```
>>> names[1:3] = [ "Icy Tea", "I See Tea" ]
>>> names
['ICT', 'Icy Tea', 'I See Tea']
```

- += append elements at middle, same as `.insert()`

```
>>> names[1:1] += [ "Ice City" ]
>>> names
['ICT', 'Ice City', 'Icy Tea', 'I See Tea']
```

Lists

```
>>> names
['ICT', 'Ice City', 'Icy Tea', 'I See Tea']
```

- `sort()` elements

```
>>> names.sort()
>>> names
['I See Tea', 'ICT', 'Ice City', 'Icy Tea']
```

- `del` delete elements

```
>>> del names[1]
>>> names
['I See Tea', 'Ice City', 'Icy Tea']
```

Lists

```
>>> names
['I See Tea', 'Ice City', 'Icy Tea']
```

- `.remove()` occurrences

```
>>> names.remove("Icy Tea")
>>> names
['Ice City', 'I See Tea']
```

Range

- Generates a series of integers
- Very popular, widely used
- `range(end)` $ = [0..end-1]$.
- `range(start, end)` $ = [start..end-1]$.
- `range(start, end, step)` $ = {x | x = start + k * step, x < end}$

Range

```
>>> nums = range(10,15)
>>> print(nums)
range(10, 15)
>>> [x for x in nums]
[10, 11, 12, 13, 14]
```

Tuples

- Immutable sequence
- Contain any type of element.
- A very common use of tuples is a simple representation of pairs
  - Positition (x, y)
  - Size (w, h)
  - ...

Tuples

- Comma generated expression

```
>>> p = 10, 20
>>> p
(10, 20)
>>> p = (20, 40)
>>> p
(20, 40)
>>> type(p)
<class 'tuple'>
>>> p[1]=1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignm
```

Maps

- Key/value pairs
  - Key must be unique
  - Similar to JSON objects
- Unordered, mutable
- Implemented by `dict`

Maps

- Initialization

```
info = {"name": "USTH", "age": 10, \
        "depts": [ "ict", "ged"] }
```

- Key operations
  - `in`, `not in`: check key presence

```
>>> "name" in info
True
```

  - `max`, `min` of key

```
>>> max(info)
'name'
```

Maps: Operations

- Value operations
  - `d[k]`: get value by key
  - `d[k] = v`: set value to key
  - `del d[k]` remove key from dict

```
>>> info["name"]
'USTH'
>>> info["age"] = 11
>>> info["age"]
11
>>> del info["depts"]
>>> info
>>> info
{'name': 'USTH', 'age': 11
```

Maps: Methods

- `d.get(k[, default])`: same as `d[k]`, fallback to `default` if key not found
- `d.pop(k[, default])`: `del d[k]` and return previously deleted `d[k]`, fallback to `default` if key not found
- `d1.update(d2)`: for each key in `d2`, sets `d1[key]` to `d2[key]`, replacing the existing value if there was one
- `d.keys()`: returns list of keys
- `d.values()`: returns list of values
- `d.items()`: returns list of `(key,value)` tuples.

Maps: Methods

```
>>> info = {"name": "USTH", "age": 10, \
            "depts": [ "ict", "ged"] }
>>> info.get("name")
'USTH'
>>> info.get("address", "Earth")
'Earth'
>>> info.pop("depts")
['ict', 'ged']
>>> info.keys()
dict_keys(['name', 'age'])
>>> info.values()
dict_values(['USTH', 10])
>>> info.items()
dict_items([('name', 'USTH'), ('age', 10)])
```

Loops

What

- Loops (repeated steps) have iteration variables
- Iteration variable changes each time through a loop
- Often these iteration variables go through a sequence of numbers.

What

```
n = 5
while n > 0 :
    print(n)
    n = n - 1
print('Blastoff!')
```

```
5
4
3
2
1
Blastoff!
```

break

- The `break` statement ends the current loop
- Jumps to the statement immediately following the loop

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)
print('Done!')
```

continue

- The `continue` statement ends the current iteration
- Jumps to the top of the loop and starts the next iteration

```
while True:
    line = input('> ')
    if line[0] == '#' :
        continue
    if line == 'done' :
        break
    print(line)
print('Done!')
```

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

range()

- range()
  - built-in function
  - returns sequence of numbers in a range
- Very useful in "for" loops
- 1, 2, or 3 arguments

```
x = range(5)
print(x)
[0, 1, 2, 3, 4]

x = range(3, 7)
print(x)
[3, 4, 5, 6]

x = range(10, 1, -2)
print(x)
[10, 8, 6, 4, 2]
```

range()

- `for` statement
  - Iterates over the members of a sequence in order
  - Executes the block each time

```
for i in <collection>
    <loop body>
```

- Examples

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('Blastoff!')
```

```
for n in range(5, 0, -1):
    print(n)
print('Blastoff!')
```

Practical work 1: student mark management

- Functions
  - Input functions:
    - Input number of students in a class
    - Input student information: id, name, DoB
    - Input number of courses
    - Input course information: id, name
    - Select a course, input marks for student in this course
  - Listing functions:
    - List courses
    - List students
    - Show student marks for a given course
- Push your work to corresponding forked Github repository

OOP in Python

Tran Giang Son, tran-giang.son@usth.edu.vn

ICT Department, USTH

Questions!?!!![1]

- What is a class? What is an object?
- What is the difference between an object and a class?
- Where do objects come from (how do you create one)?
- How many objects of a given class can you have at a given time?
- Each data type in Java (and in many other languages) can be classified as one of two kinds. What are they, and how are they different?

[1] Or exam?!

Questions!?!!!

- What is a primitive type? What is a reference type (or object type)?
- What is a method?
- Can a class have more than one method with the same name? If so, are there any restrictions?
- What is a parameter? What is a return value?
- Can parameters and return values be primitive types? Reference types?

Questions!?!!!

- What is type matching or type conformance?
- What is a constructor?
- What is assignment? How is it different for reference types versus primitive types?
- What are accessor methods and mutator methods?
- What is abstraction and why is it a good thing?

Questions!?!!!

- What is inheritance?
- What is an inheritance hierarchy?
- What is a subclass? A superclass?
- What are the advantages of using inheritance?
- What is the difference between an is-a and a has-a relationships?

Questions!?!!!

- What is polymorphism?
- What are overriding and overloading? Are they the same? Give examples.
- What does the keyword super mean? When is it used?
- What does the keyword protected mean? When is it used, and what does it do?
- What is meant by the static and dynamic types of a variable?

Questions!?!!!

- What is an abstract class? When are they useful? Give an example.
- What is multiple inheritance? Why is it useful? Can it be done in Java? Is there a substitute for it?
- What is an interface? Why are they useful?

Object and Class

Why

Procedural Programming

- Variables and related functions are separated
- Programs is divided into functions

Object-Oriented Programming

- Variables and related functions are bound together
- Programs are divided into objects

How

- Define a class

```
class <ClassName>
```

- Define a method

```
def <methodName>([args])
```

- Define a constructor

```
def __init__([args])
```

- Create an object from class

```
<obj> = <ClassName>([args])
```

- `self`: current object

How

```
class Person:
    def print(self):
        print("Name:", self.name)
        print("Age:", self.age)

    def __init__(self, n, a):
        self.name = n
        self.age = a

macron = Person("Emmanuel Macron")
```

How

- Call an object's method

  ```
  <obj>.<methodName>([args])
  ```

- Accessing an object's attribute

  ```
  <obj>.<attr> = "values"
  ```

How

- Object comparison: `__lt__` method
  - Compares current object with another instance
  - Return `True` if less than[2] the other instance

```
def __lt__(self, other):
    return self.age < other.age
```

[2] Hence its name is `__lt__`

How

- Object's string representation: `__str__` method
  - Defines how an object will be stringified
  - Mostly when using with `print()`

```
def __str__(self):
    return f"My name is {self.name}. I am {self.age}."
```

How: complete example clazz.py

```
#!/usr/bin/env python3
class Person:
    def __init__(self, n, a):
        self.name = n
        self.age = a

    def describe(self):
        print("Name:", self.name)
        print("Age:", self.age)

    def __lt__(self, other):
        return self.age < other.age

    def __str__(self):
        return f"My name is {self.name}. I am {self.age}."

macron = Person("Emmanuel Macron", 43)
macron.describe()
print(macron)

biden = Person("Joe Biden", 78)
print(f"Macron is younger: {macron < biden}")
```

Inheritance

Defining Inheritance

- Define a child class with a superclass in parentheses
- All methods and attributes from superclass will be inherited to subclass

```
class Person:
    # already defined before...

class President(Person):
    def set_term(self, term):
        print(f"Setting term to {term}")
        self.term = term
```

Defining Inheritance

- Using the newly defined class and method

```
print("Macron is now President")
macron = President("Emmanuel Macron", 43)
macron.set_term(25)       # from President
macron.describe()         # from Person
```

```
$ ./clazz.py
Macron is now President
Setting term to 25
Name: Emmanuel Macron
Age: 43
```

Checking inheritance

- Built-in functions `isinstance()` and `issubclass()`
  - `isinstance()` returns True if the object is an instance of the class or other classes derived from it
  - `issubclass()` is used to check for class inheritance.

Checking inheritance

```
print(f"Macron is President: {isinstance(macron, President)}");
print(f"Macron is Person: {isinstance(macron, Person)}");
print(f"President is Person: {issubclass(President, Person)}")
print(f"Person is President: {issubclass(Person, President)}")
```

```
$ ./clazz.py

Macron is President: True
Macron is Person: True
President is Person: True
Person is President: False
```

Multiple Inheritance

- Python supports multiple inheritance
- Simply by adding more base class into the parentheses

```
class Person:
    # already defined before...

class Employee:
    def work(self):
        print("I should be paid...")

class President(Person, Employee):
    # already defined before...
```

Multiple Inheritance

```
print("Macron is now President")
macron = President("Emmanuel Macron", 43)
macron.set_term(25)     # from President
macron.describe()       # from Person
macron.work()           # from Employee
```

```
$ ./clazz.py
Setting term to 25
Name: Emmanuel Macron
Age: 43
I should be paid...
```

Polymorphism

Method overrides

- A superclass's method can be overridden, simply by `def`fing the same method name in the subclass
- A superclass instance can be accessed using `super()` in the subclass

```
class Person:
    # already defined before...

class President(Person, Employee):
    # something before
    def describe(self):
        super().describe()
        print("Term:", self.term)

    def work(self):
        super().work()  # from Employee
```

Method overrides

- Using the overridden method

```
print("Macron is now President")
macron = President("Emmanuel Macron", 43)
macron.set_term(25)       # from President
macron.describe()         # from Person
macron.work()             # from President
```

```
$ ./clazz.py
Setting term to 25
Name: Emmanuel Macron
Age: 43
I am well paid!
President is well paid!
```

Encapsulation

Private / Public access

- `public` by default
- No specified keyword
- Use underscore prefixes
  - `name`: public
  - `_name`: protected
  - `__name`: private

Private / Public access

- Accessor methods / Mutator methods
- Getter / Setter

```
class Employee:
    def __init__(self):
        self.__salary = 0

    def _get_salary(self):
        return self.__salary

    def set_salary(self, salary):
        self.__salary = salary

    def work(self):
        if self.__salary == 0:
            print("I should be paid...")
        else:
            print("I am well paid!")
```

Private / Public access

```
print("Macron is now President")
macron = President("Emmanuel Macron", 43)
macron.set_term(25)      # from President
macron.describe()        # from Person
macron.set_salary(1000) # from Employee
macron.work()            # from President
print(f"Macron salary is {macron._get_salary()}")
print(f"Macron salary is {macron.__salary}")
```

```
$ ./clazz.py
Macron is now President
Setting term to 25
Name: Emmanuel Macron
Age: 43
President is well paid!
Macron salary is 1000
Traceback (most recent call last):
  File "../clazz.py", line 59, in <module>
    print(f"Macron's salary is {macron.__salary}")
AttributeError: 'President' object has no attribute '__salary'
```

Practical work 2: OOP'ed student mark management

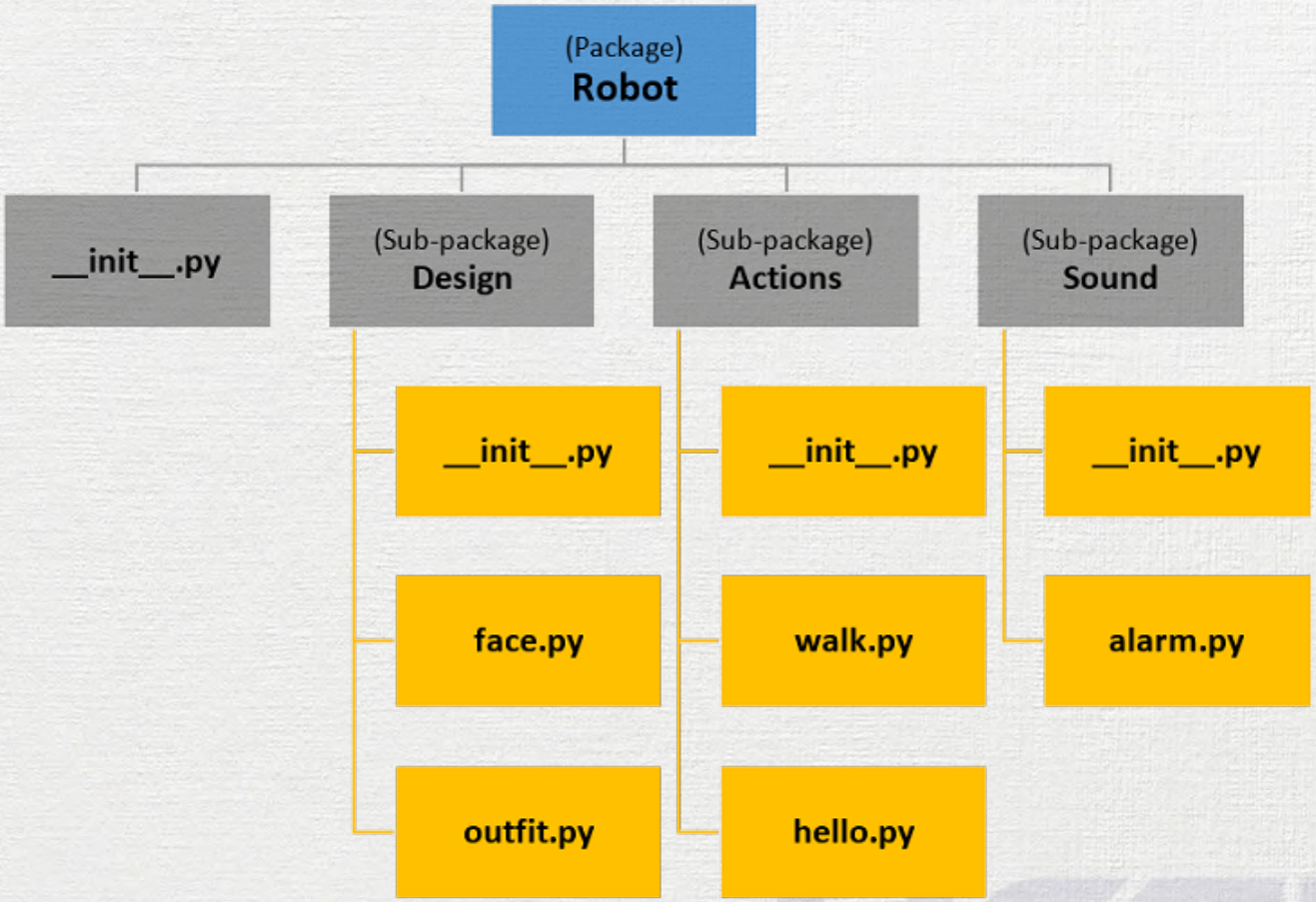
- Copy your practical work 1 to 2.student.mark.oop.py
- Make it OOP'ed
- Same functions
  - Proper attributes and methods
  - Proper encapsulation
  - Proper polymorphism
    - e.g. `.input()`, `.list()` methods
- Push your work to corresponding forked Github repository

Modules and Packages

Tran Giang Son, tran-giang.son@usth.edu.vn

ICT Department, USTH

Intro


(Package)
Robot
__init__.py
(Sub-package)
Design
(Sub-package)
Actions
(Sub-package)
Sound
__init__.py
face.py
outfit.py
__init__.py
walk.py
hello.py
__init__.py
alarm.py

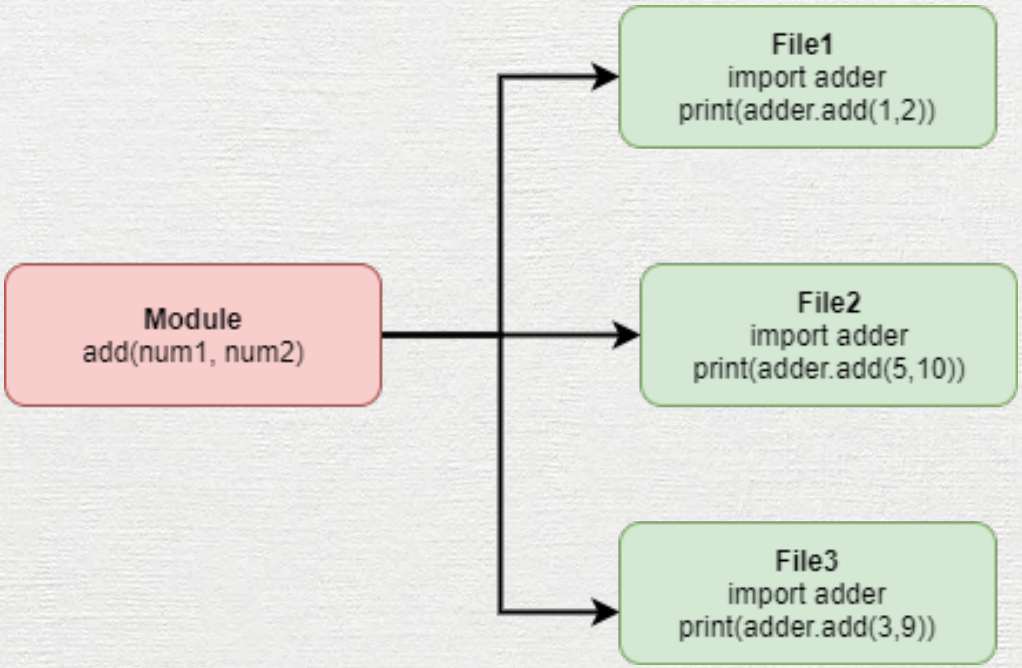
Modules

What

- Module: reusable piece of code
  - Can be from external sources
  - Regular `.py` file with `def`s and `class`es

What

- Similar to `.ko`, `.so`, `.dll`, ...



Why

- Modularity
- Reusability
- Shareibility
- Maintainability

How: Making a module

- Create a separated `.py` file
  - `define` functions, classes as usual
  - `define` `__init__()`, as needed

How: Using a module

- `import` a module to the global namespace
  - No path, no extension
- Use functions, classes, constants provided by module
  - `<module>.<method/class/const>`

```
>>> import math
>>> print(math.pi)
3.141592653589793
```

How: Using a module

- Shortcut to the global namespace
  - `from <module> import <func/class/const>`
  - Use `<func/class/const>` directly

```
>>> from math import pi
>>> print(pi)
3.141592653589793
>>> from math import *
>>> print(e)
2.718281828459045
```

How: Using a module

- Aliasing
  - `from <module> import <func/class/const> as <alias>`
  - Use `<alias>`

```
>>> import numpy as np
>>> np.pi
3.141592653589793
```

Packages

What

- A bunch of related modules
- [optional] A bunch of sub-packages
- [optional] A bunch of sub-sub-packages

Why

- Higher level of modularity
- Less `import` modules from the same packages
- Module name A.B designates a submodule named B in a package named A.

How

- Install from Python Package Index (PyPI)
  - `pip install <packageName>`
- Write your own package
  - Add your modules
  - Add a dedicated file `__init__.py` for initialization

How

- `import` a module in a specific package

```
import Package1.PackageModule1
```

- `import` a whole package

```
import Package1
```

- Should also import modules in package `__init__.py` for automatic module imports

```
import Module1
import Module2
import Module3
```

How

- Package can be nested
  - Sub-package inside a package
    - Sub-sub-package inside a sub-package
- Just make an `__init__.py`, even empty one

Practice!

Practical work 3: some maths and decorations

- Copy your practical work 2 to 3.student.mark.oop.math.py
- Use `math` module to round-down student scores to 1-digit decimal upon input, `floor()`
- Use `numpy` module and its `array` to
  - Add function to calculate average GPA for a given student
    - Weighted sum of credits and marks
  - Sort student list by GPA descending
- Decorate your UI with `curses` module
- Push your work to corresponding forked Github repository

Practical work 4: modularization

- Split your program 3.student.mark.oop.math.py to modules and packages in a new `pw4` directory
  - `input.py`: module for input
  - `output.py`: module for `curses` output
  - `domains`: package for classes
  - `main.py`: main script for coordination
- Push your work to corresponding forked Github repository

Files and Directories

Tran Giang Son, tran-giang.son@usth.edu.vn

ICT Department, USTH

Reviews

- What is a file? What is a directory?
- What is a symlink?
- Shell commands:
  - How to list files in a directory?
  - How to show a file's content?
  - How to print all lines of a file containing a specific string?

Files

What

- Everything in UNIX is a file
- Named locations on disk to store information
  - Text file
  - Binary file

How: Open a file

1. Open a file
2. Read or write
3. Close the file

- Indicates that the program wants to work with a given file
  - What file?
  - What operation to work with
- `open(fileName, mode)`
  - `fileName`: what file
  - `mode`: what operations
  - returns a `File` object representing an opened file

How: Open a file

1. Open a file
2. Read or write
3. Close the file

```
f = open("test.txt", "r")
```

Mode	Meaning
`r`	Reading (default)
`w`	Writing. Creates or clears a file.
`x`	Exclusive creation. Fails if file exists.
`a`	Appending. Creates if file does not exist.
`t`	Opens in text mode. (default)
`b`	Opens in binary mode.
`+`	Opens a file for updating (`rw`)

How: Read/write

1. Open a file
2. Read or write
3. Close the file

- `f.read(size)` reads and returns `size` bytes
  - `size` is optional
  - Reads all file content by default
  - Updates current file pointer after `.read()`
  - Be careful for large files!
- `f.seek(offset)` sets current file pointer to a specific offset
- `f.write()` writes into file

How: Read/write

```
>>> f = open("test.txt", "r+")
>>> f.read(19)
"The language's core"
>>> f.seek(0)
>>> f.read()
"The language's core philosophy is summarized in \
    the document The Zen of Python:\n* Beautiful \
    is better than ugly.\n* Explicit is better \
    than implicit.\n* Simple is better than \
    complex.\n* Complex is better than \
    complicated.\n* Readability counts.\n"
>>> f.write("That's all\n")
```

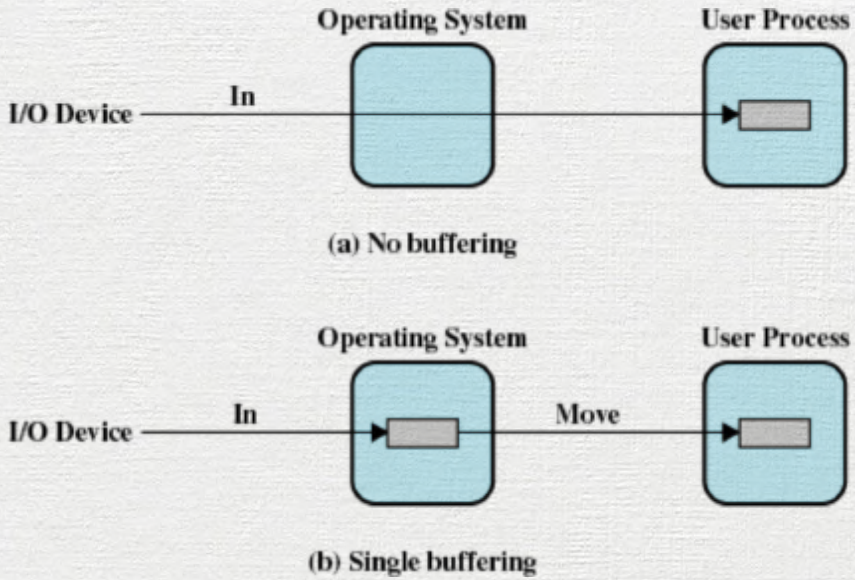
How: Read/write

- Text files
  - `.readline()`: reads until a new line.
    - There's a `\n` at the end of file
  - `.readlines()`: reads all lines

```
>>> f = open("test.txt", "r+")
>>> f.readline()
"The language's core philosophy is summarized in the documen
>>> f.readlines()
['* Beautiful is better than ugly.\n',
    '* Explicit is better than implicit.\n',
    '* Simple is better than complex.\n',
    '* Complex is better than complicated.\n',
    '* Readability counts.\n', "That's all\n"]
```

How: Buffering

- Buffer: in-memory cache of file content
  - Speeding up IO accesses[1]
  - Reading/writing blocks is faster than individual bytes



No buffering vs Single buffering

[1] Even `stdout`...

How: Buffering

- `open(fileName, mode, buffering = -1)`
- `buffering` is optional
  - `-1`, same as `io.DEFAULT_BUFFER_SIZE`
  - `0`: disable buffering
  - `1`: line buffering for text files
  - `>1`: fixed size buffer
- Flushing buffer: write buffer to disk, if any
  - Manually `f.flush()`

How: Close a file

1. Open a file
2. Read or write
3. Close the file

- Close a file after using
- Clean up OS caches, buffers
- Without closing, there may be data loss with power outage

```
f.close()
```

How: Close a file

- Automatically `.close()` using `with`

```
with open('test.txt', 'r+') as f:
    data = f.read()

# other stuffs here, f is closed.
```

How: Extras

- Exceptions
- Temporary files
- Compression
- Objects

How: Extras

1. Exceptions
2. Temporary files
3. Compression
4. Objects

- Exceptions? Remind.
- Exception:
  - Errors at runtime
  - Python: `try... except...`
- For handling IO errors:

```
filename = input("Enter file name: ")
try:
    f = open(filename, "r")
except IOError:
    print(f"Error missing file {filename}")
```

How: Extras

- Temporary files?
  - Don't care about name, location
  - Just somewhere to store temp contents
  - Automatically cleaned up after `close()`
- Module `tempfile`

```
import tempfile.TemporaryFile

# gimme a file, whenever it is
f = tempfile.TemporaryFile('w+t')
f.write("3.1415926...")
f.close()

# closed means deleted
```

1. Exceptions
2. Temporary files
3. Compression
4. Objects

How: Extras

1. Exceptions
2. Temporary files
3. Compression
4. Objects

- Compression?
  - What: Use less storage to represent data
  - Why:
    - Smaller disk storage
    - Easier for transmission over network
    - Encryption with passwords
- Plenty of existing modules
  - `zlib`, `gzip`, `bz2`, `lzma`, `tarfile`, `zipfile`
- Each module would have different advantages/disadvantages and usage.

How: Extras

1. Exceptions
2. Temporary files
3. Compression
4. Objects

Module	Compression	In-memory	Extension	Files	Directory
`zlib`	Yes	Yes	No	No	No
`gzip`	Yes	Yes	`.gz`	No	No
`bz2`	Yes	Yes	`.bz2`	No	No
`lzma`	Yes	Yes	`.xz`	No	No
`tarfile`	No	No	`.tar`	Yes	Yes
`zipfile`	Yes	Yes	`.zip`	Yes	Yes

How: Extras

1. Exceptions
2. Temporary files
3. Compression
4. Objects

- *Serialize* objects into byte array
  - Save state to disk, optionally compressed (!)
  - Load state later
  - Transmit object to a remote machine

How: Extras

1. Exceptions
2. Temporary files
3. Compression
4. Objects

- `pickle` module
  - `pickle.dump(obj, f)`: save object `obj` into *already-opened-for-binary-write* file `f`
  - `obj = pickle.load(f)`: load object from *already-opened-for-binary-read* file `f`

How: Extras

1. Exceptions
2. Temporary files
3. Compression
4. Objects

- What can be pickled?
  - `None`, `True`, and `False`
  - Integers, floating point numbers, complex numbers
  - Strings, bytes, bytearrays
  - Tuples, lists, sets, and dictionaries containing only picklable objects
- Can also pickled behaviors:
  - Functions defined at the top level of a module
  - Built-in functions defined at the top level of a module
  - Classes that are defined at the top level of a module

How: Extras

1. Exceptions
2. Temporary files
3. Compression
4. Objects

- `pickle` vs `json`

Feature	`pickle`	`json`
Compatibility	Python-only	Open
Format	Binary	Text
Readability	Nah	Yay
Data types	Many	Limited

Directories

What

- Hierachical structure
  - A bunch of files
  - A bunch of sub-directories
- Looks like a tree
  - Path indicates a location inside a directory

Why

- For organization of data
- Easier traversing and browsing

How

- Listing: `os.scandir()`, `os.walk()` (recursive)
- Creating: `os.mkdir()`, `os.mkdirs()`
- Deleting: `os.rmdir()`, `shutil.rmtree()` (recursive)

How

```
>>> import os
>>> e=os.scandir(".")
>>> [f for f in e]
[<DirEntry '0. pre-intro.md'>, \
    <DirEntry '1. course intro.md'>, \
    <DirEntry '2. introz.md'>, \
    <DirEntry '3. language.md'>, \
    <DirEntry '4. oop.md'>, \
    <DirEntry '5. modules.md'>]
>>> os.mkdirs("figs/intro")
```

Practical work 5: persistent info

- Copy your `pw4` directory into `pw5` directory
- Update your input functions
  - Write student info to `students.txt` after finishing input
  - Write course info to `courses.txt` after finishing input
  - Write marks to `marks.txt` after finishing input

Practical work 5: persistent info

- Before closing your program
  - Select a compression method
  - Compress all files aboves into `students.dat`
- Upon starting your program,
  - Check if `students.dat` exists
  - If yes, decompress and load data from it
- Push your work to corresponding forked Github repository

Practical work 6: `pickled` management system

- Copy your `pw5` directory into `pw6` directory
- Upgrade the persistence feature of your system to use `pickle` instead, still with compression
- Push your work to corresponding forked Github repository

Multi Processing

Tran Giang Son, tran-giang.son@usth.edu.vn

ICT Department, USTH

Review

- Process
- Scheduling
- IO Redirection

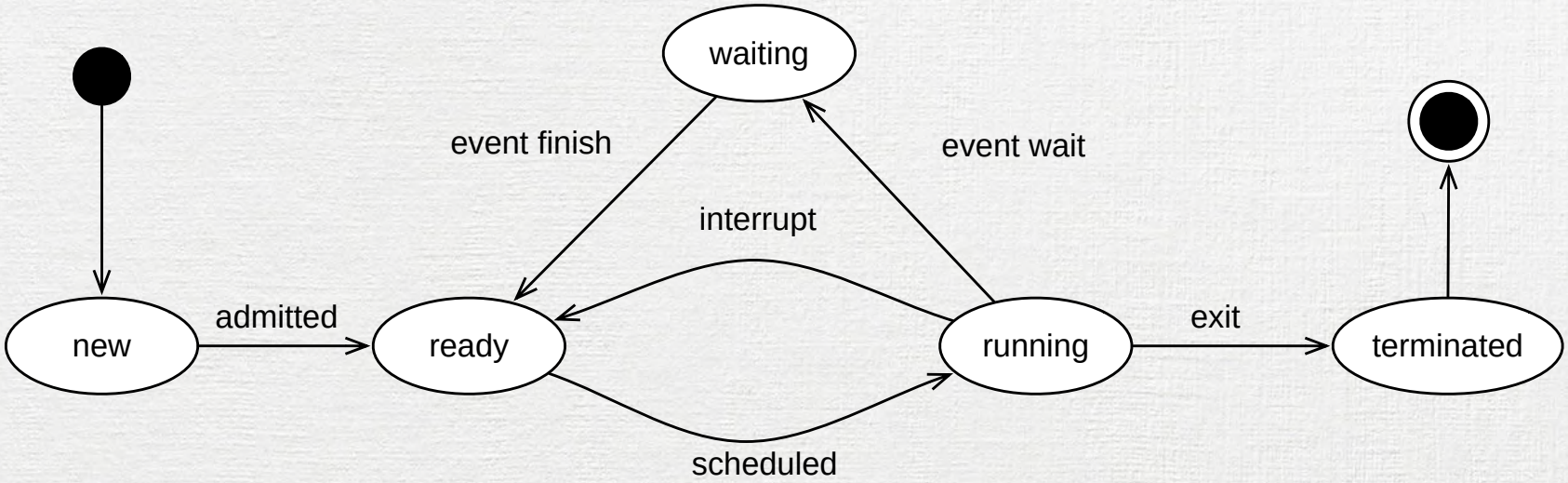
Process

- What is process?
- Process vs program?

Process

- Process is a program in execution state (**active**)
- Why process?
  - Program is passive
  - No execution → what's running?
- A process execution state contains
  - Processor state (context)
  - File descriptors
  - Memory allocation
    - Process stack
    - Data section
    - Heap

Process States



- **new**: process has just been created
- **ready**: waiting to be assigned (scheduled) to a processor
- **running**: it's executing instructions
- **waiting**: waiting for some events to occur
- **terminated**: finished execution

Process Creation

- Start a new process == Create a new process
  - Create new child process
    - Can create child process → grand child process
  - Dependent on OS, parent and child can share
    - **All** resources: opened files, devices, etc...
    - **Some** resources: opened files only
    - **No** resource
- A fully loaded system will have a process tree

Process Creation

```
$ pstree -A
init-+-acpid
     |-cron
     |-daemon---mpt-statusd---sleep
     |-dbus-daemon
     |-dovecot-+-anvil
     |         |-config
     |         `-log
     |-master-+-pickup
     |        |-qmgr
     |        `-tlsmgr
     |-mysqld_safe---mysqld---23*[{mysqld}]
     |-php5-fpm---2*[php5-fpm]
     |-proftpd
     |-screen---bash---python2---{python2}
     |-sshd-+-sshd---sshd---bash---pstree
     |      `-sshd---sshd
     |-udevd---2*[udevd]
     `-znc---{znc}
```

Process Creation on Windows

```
BOOL WINAPI CreateProcess(
  _In_opt_    LPCTSTR                lpApplicationName,
  _Inout_opt_ LPTSTR                 lpCommandLine,
  _In_opt_    LPSECURITY_ATTRIBUTES  lpProcessAttributes,
  _In_opt_    LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  _In_        BOOL                   bInheritHandles,
  _In_        DWORD                  dwCreationFlags,
  _In_opt_    LPVOID                 lpEnvironment,
  _In_opt_    LPCTSTR                lpCurrentDirectory,
  _In_        LPSTARTUPINFO          lpStartupInfo,
  _Out_       LPPROCESS_INFORMATION  lpProcessInformation
);
```

Source: MSDN

Process Creation on Windows

- A simplified WinAPI function:

```
UINT WINAPI WinExec(
  _In_ LPCSTR lpCmdLine,
  _In_ UINT   uCmdShow
);
```

Process Creation on Windows

- A simplified WinAPI function:

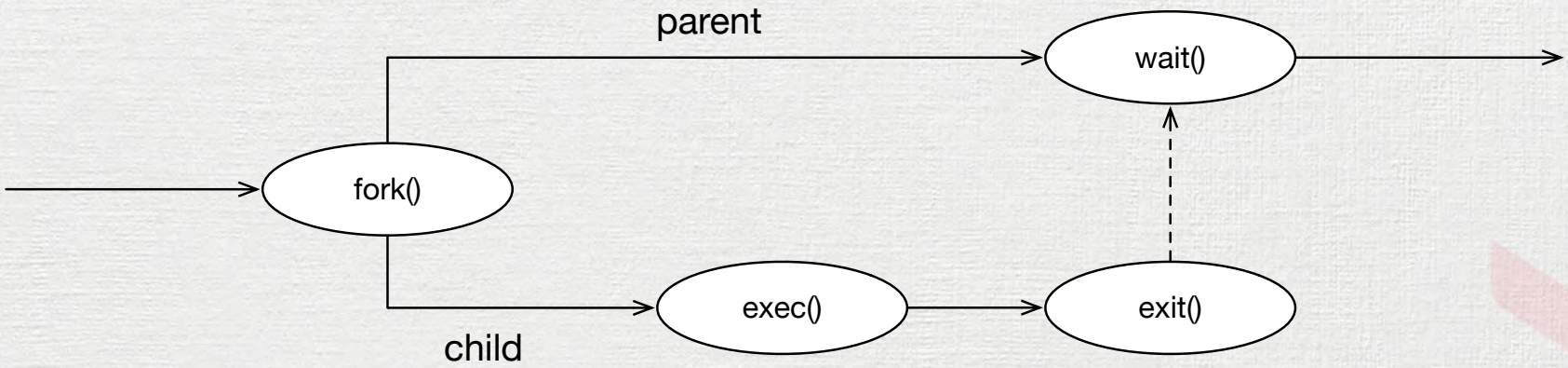
```
UINT WINAPI WinExec(
  _In_ LPCSTR lpCmdLine,
  _In_ UINT   uCmdShow
);
```

- It's deprecated.

Source: MSDN

Process Creation on UNIX/Linux

- New processes are not created from scratch
- Two steps
  - `fork()`
  - `exec()`



Process Creation on UNIX/Linux

- `fork()`
  - Perfectly «clone» current process to a new process
    - Open files
    - Register states
    - Memory allocations
    - **Except** process id
  - Who's who?
    - Parent?
    - Child?

```
pid_t fork(void);
```

Process Creation on UNIX/Linux

- Parent: `fork()` returns process id of child
- Child: `fork()` returns 0
- Example

```c
#include <unistd.h>
#include <stdio.h>
int main() {
    printf("Main before fork()\n");
    int pid = fork();
    if (pid == 0) printf("I am child after fork()\n");
    else printf("I am parent after fork(), child is %d\n", pid);
    return 0;
}
```

```
$ ./dofork
Main before fork()
I am parent after fork(), child is 2378
I am child after fork()
```

Process Creation on UNIX/Linux

- `exec()`
  - Load an executable binary to replace current process image
  - A family of functions.
  - Ask `man`

```
int execl(...);
int execle(...);
int execlp(...);
int execv(...);
int execvp(const char *file, char *const argv[]);
int execvP(...);
```

Process Creation on UNIX/Linux

- exec() example

```
#include <stdio.h>
#include <unistd.h>
int main() {
    printf("Going to launch ps -ef\n");
    char *args[]= { "/bin/ps", "-ef" , NULL};
    execvp("/bin/ps", args);
    return 0;
}
```

Scheduling

- Multiple processes running at the same time
- Process scheduler is a part that decides which processes to be executed at a certain time.

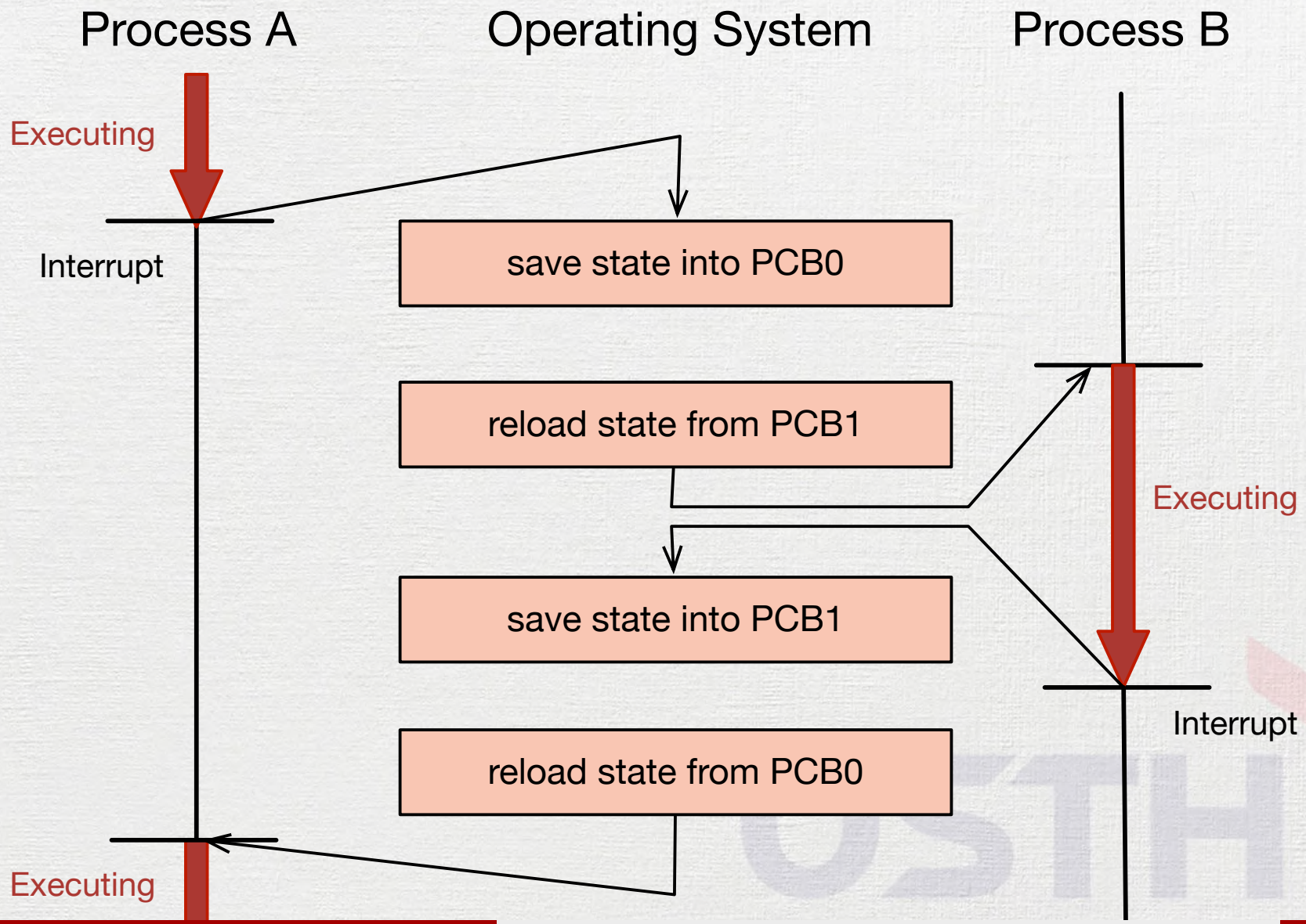
Scheduling

- Maximize CPU usage
- Responsiveness for User interface
- Provide computational power for heavy-workload processes
- «Multitasking»
- Different characteristics of processes
  - CPU bound: spends more time on computation
  - I/O bound: spends more time on I/O devices (reading/writing disk, printing...)

Scheduling

- By the ability to pause running processes
  - Preemption: OS forcely pauses running processes
  - Non-preemption (also cooperation): processes willing to pause itself
- By duration between each «switch»
  - Short term scheduler: milliseconds (fast, responsive)
  - Long term scheduler: seconds/minutes (batch jobs)

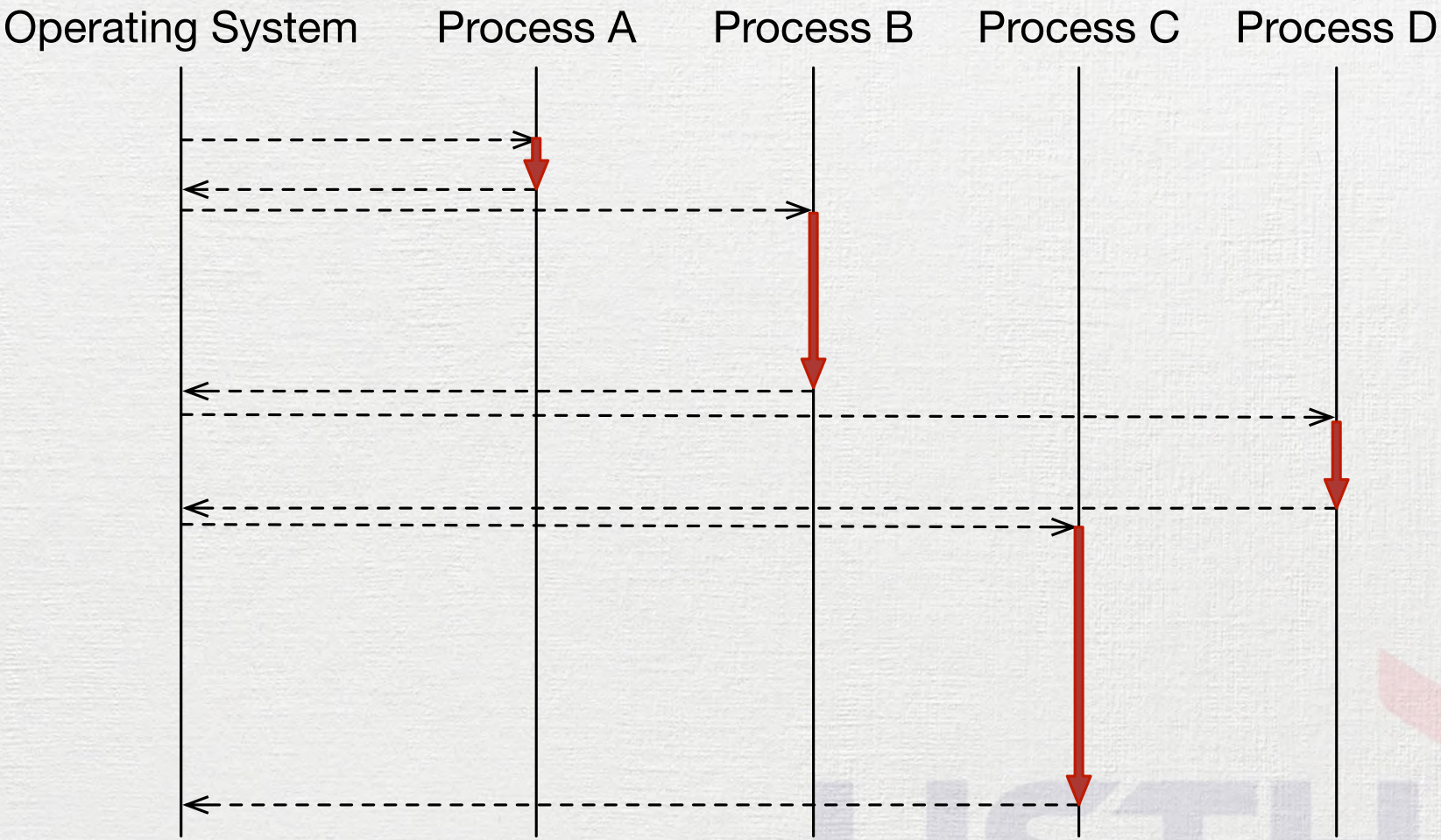
Scheduling with Context Switch



Scheduling with Context Switch

- Switch between processes
  - Save data of old process
  - Load previously saved data of new process
- Context switch is overhead
  - No work done for processes during context switch
  - Time slice (time between each switch) is hardware-limited

Scheduling with Context Switch



Scheduler

- Knowns
  - List of processes
  - Process states
  - Accounting information
- Constraints
  - Process priority (if any)
    - Processes have scheduling **priority**
    - Indicates the importance of each process
    - Higher priority: more likely to be scheduled

Scheduler

- Problem 1: What processes to run next?
  - Job queue - set of all processes **entering** the system, stored on disk
  - Ready queue - set of all processes residing in **main memory**, ready and waiting to execute
  - Device queues - set of processes waiting for **an I/O device**
  - Lists of PCBs
  - Processes change state $\rightarrow$ they migrate among the various queues

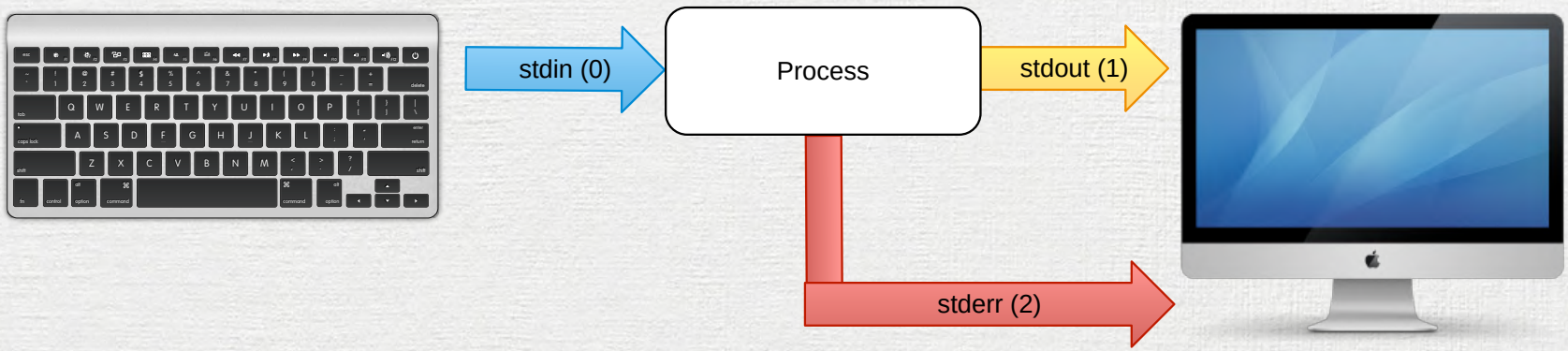
Scheduler

- Problem 2: How long should it run?
  - First In First Served
  - Earliest Deadline First
  - Shortest Remaining Time
  - Round Robin
  - ...

Scheduler

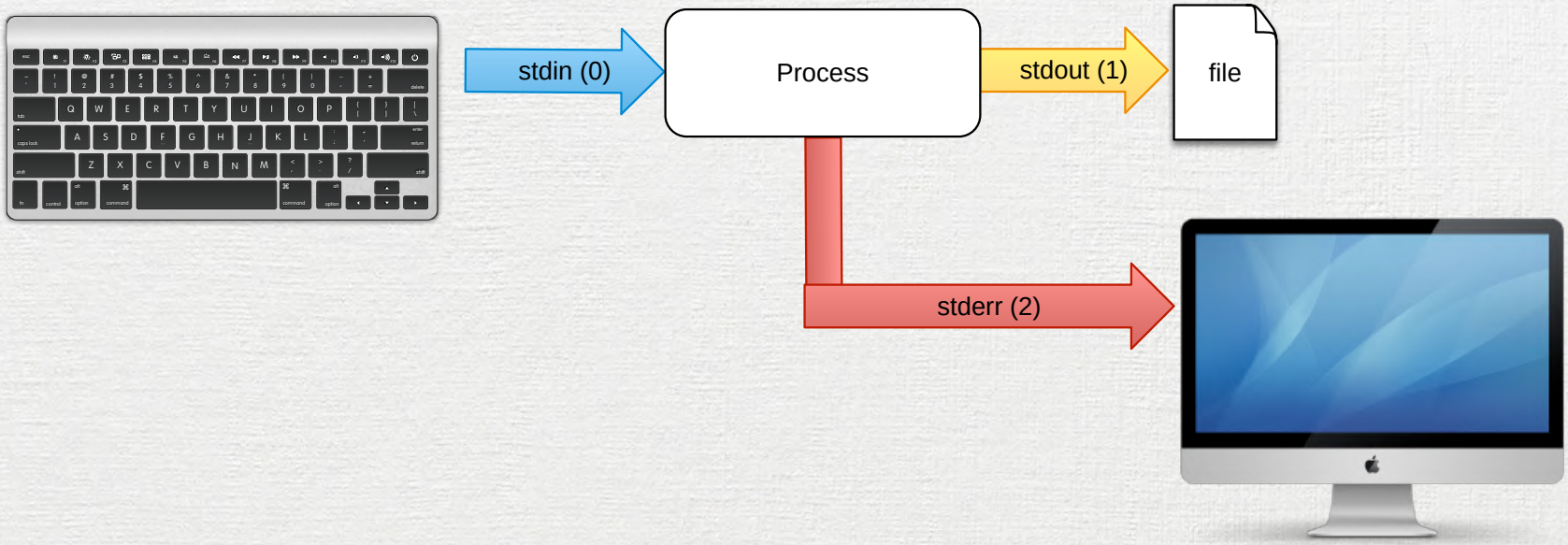
Algorithm	Preempt?	Priority?	Note
First Come, First Served	No	No	Depends on arrival time
Shortest-Job-First	No	Yes	Low waiting time ω
Shortest-Remaining-Time-First	Yes	Yes	Preemptive SJF, low ω
Round Robin	Yes	No	Low response time ρ
Multilevel Queue	Depends	Depends	Several subqueues, permanent
Multilevel Feedback Queue	Depends	Depends	Several subqueues, migrate

IO Redirection



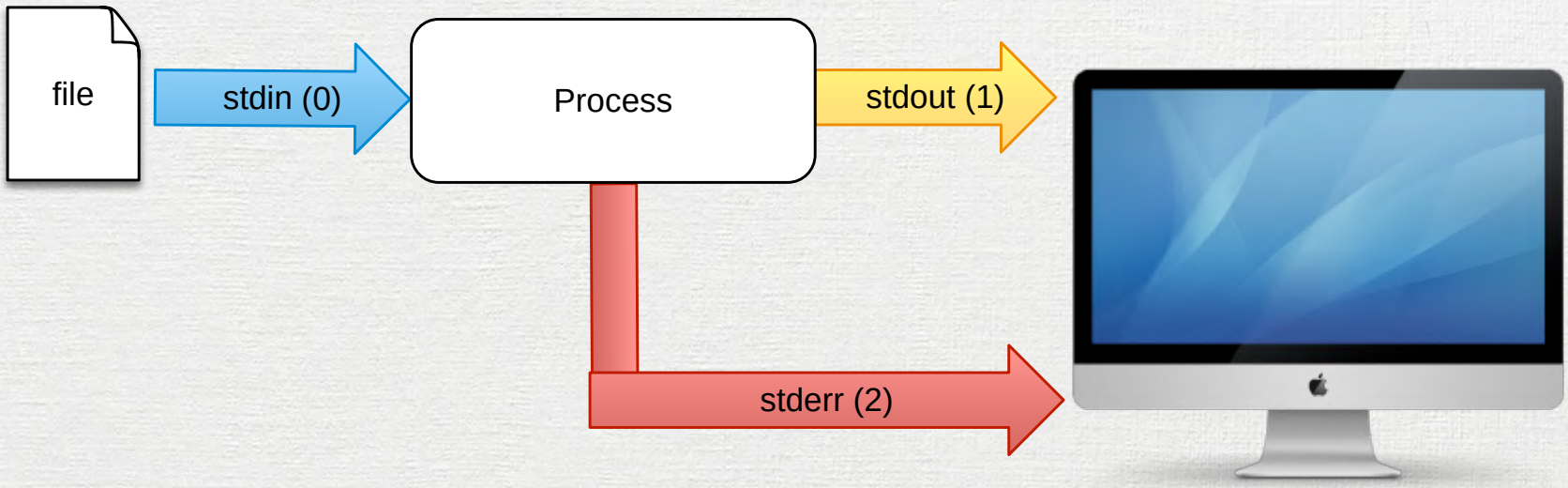
Default: input from keyboard and output to terminal

IO Redirection



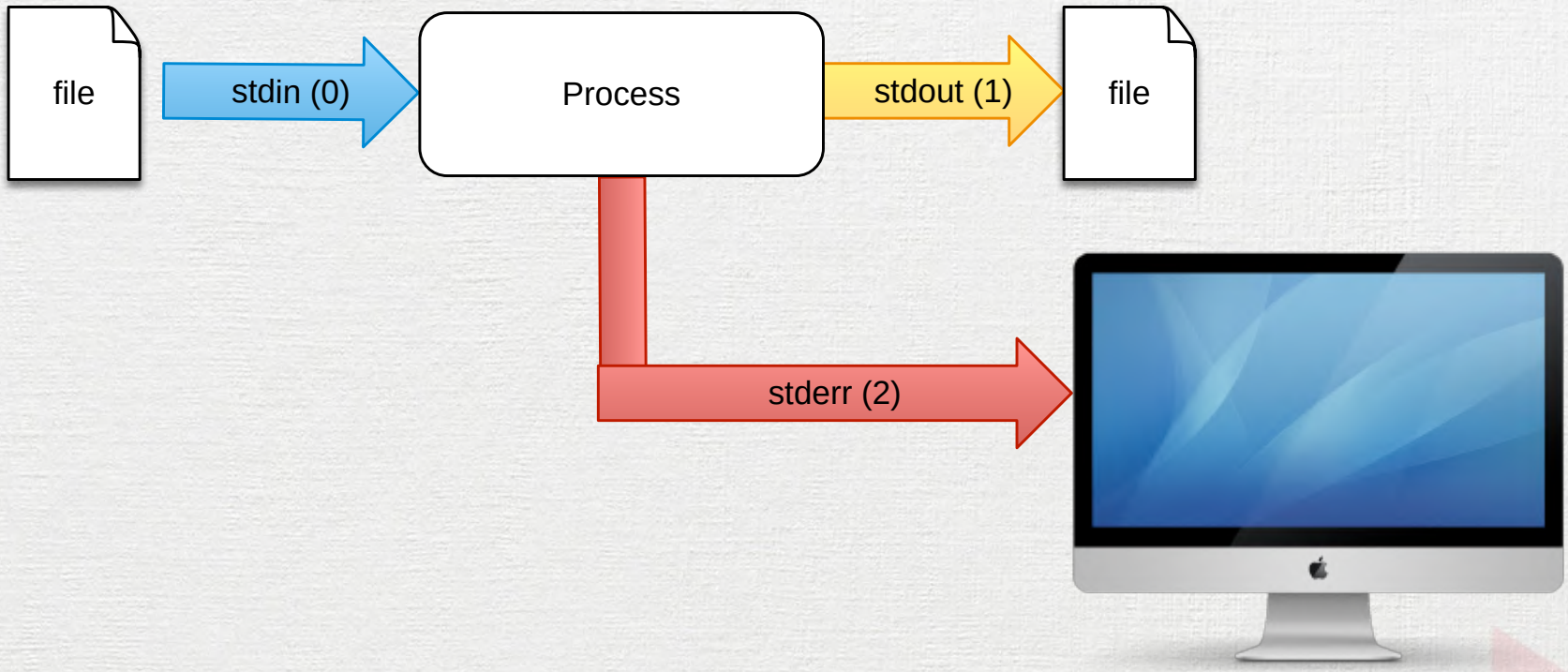
Input from keyboard and output to file

IO Redirection



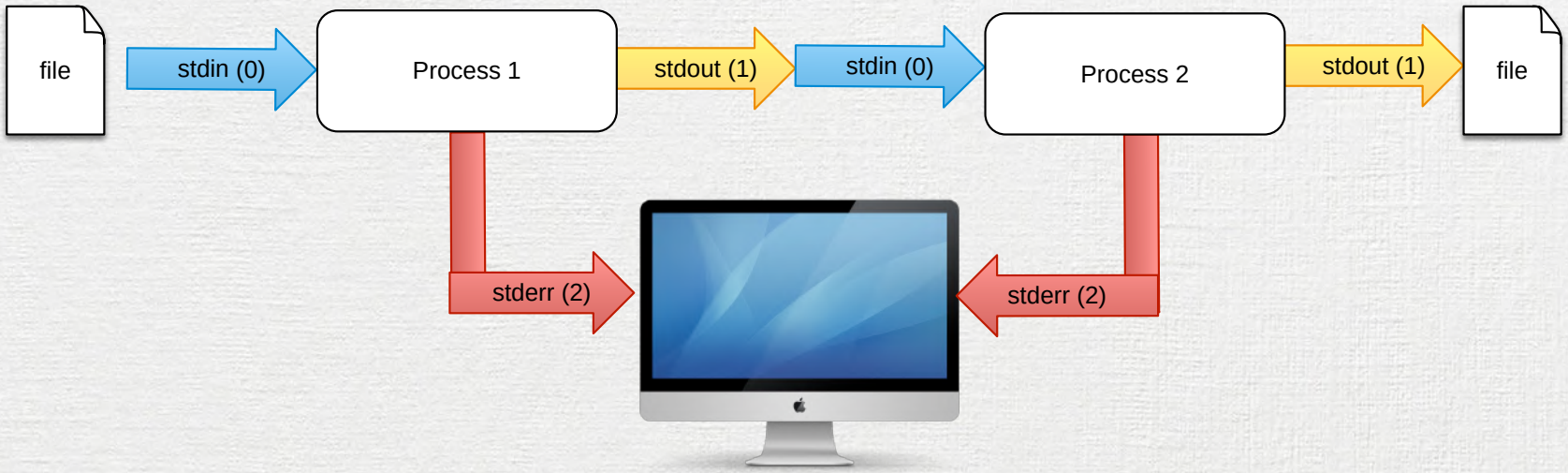
Input from file and output to terminal

IO Redirection



Input from file and output to another file

IO Redirection



Input from file, pipe output of Process 1 to Process 2, output to another file

Task

- Create a process
  - Run and wait for finish
  - Run in background
  - Run with timeout
- IO redirection
  - Redirect input
  - Redirect output
  - Redirect with pipe
- Terminate
- Get return code

os module

- os module is deprecated in Python 3
- This is for references only.

Task	How
Run and wait	`os.system("ps aux")`
Run in background	`os.system("long_command.sh &")`
Timeout	N/A
Redirect input	`os.popen("bc", "w").write("1+2")`
Redirect output	`print(os.popen("ps aux", "r").readlines())`
Redirect with pipe	`os.pipe()`, `os.fork()`
Terminate	`os.kill(pid, signal.SIGTERM)`
Get return code	return value of `os.system()`

subprocess module

Task	How
Run and wait	`subprocess.run(["ps", "aux"])`
Run in background	`subprocess.Popen("long_command.sh")`
Timeout	`subprocess.run("long_command.sh", timeout = 10)`
Redirect input	`subprocess.Popen("bc", stdin=subprocess.PIPE).communicate(b"3+4\n")`
Redirect output	`subprocess.Popen(["ps", "aux"], stdout=subprocess.PIPE).communicate()`
Redirect with pipe	`subprocess.Popen("bc", stdin=anotherProcess.stdout)`
Terminate	`anotherProcess.terminate()`, `anotherProcess.kill()`
Get return code	`subprocess.check_output()`, catch `CalledProcessError`

Practical work 7: Python shell

- Create a new python program, name it «7.shell.py»
- Make a shell
  - User inputs command
  - Shell executes the command, print output
  - Support IO redirection
    - input from file to process
    - output from process to file
    - e.g. input from one process being output of another

Practical work 7: Python shell

- Run it and test some commands
  - `ls -la`
  - `ls -la > out.txt`
  - `bc < input.txt`
  - `ps aux | grep term`
- Push your work to corresponding forked Github repository

Multithreading

Tran Giang Son, tran-giang.son@usth.edu.vn

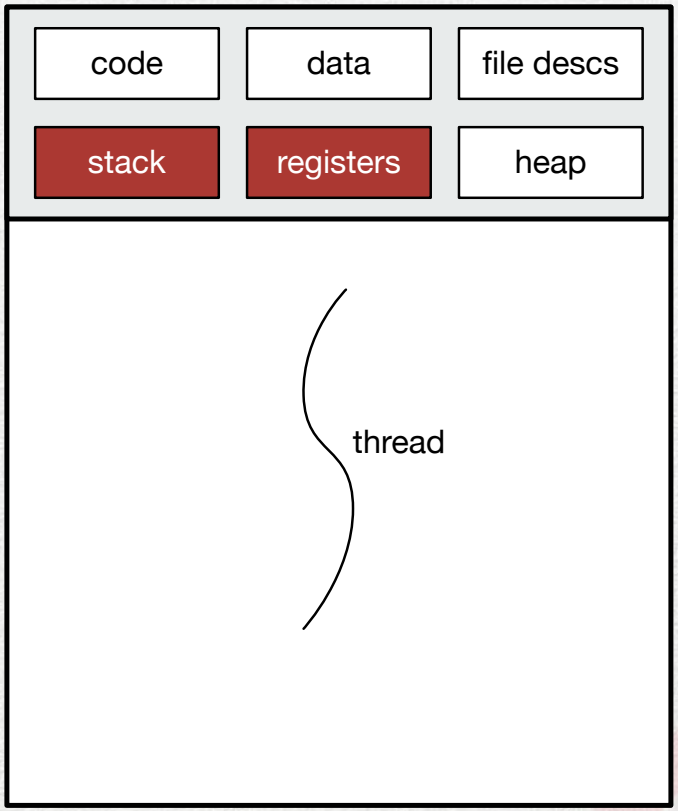
ICT Department, USTH

Remind PCB

- Process Control Block
- Contains
  - Process ID
  - Process state (new/ready/running/waiting/terminated)
  - **Processor state (program counter, registers)**
  - File descriptors
  - Scheduling information (next section)
  - Accounting information (limits)

Thread & Single-threaded process

- Thread
  - a single flow of execution
  - belongs to a process
  - can be considered as lightweight process
- Single-threaded process
  - Default
  - Only one thread per process



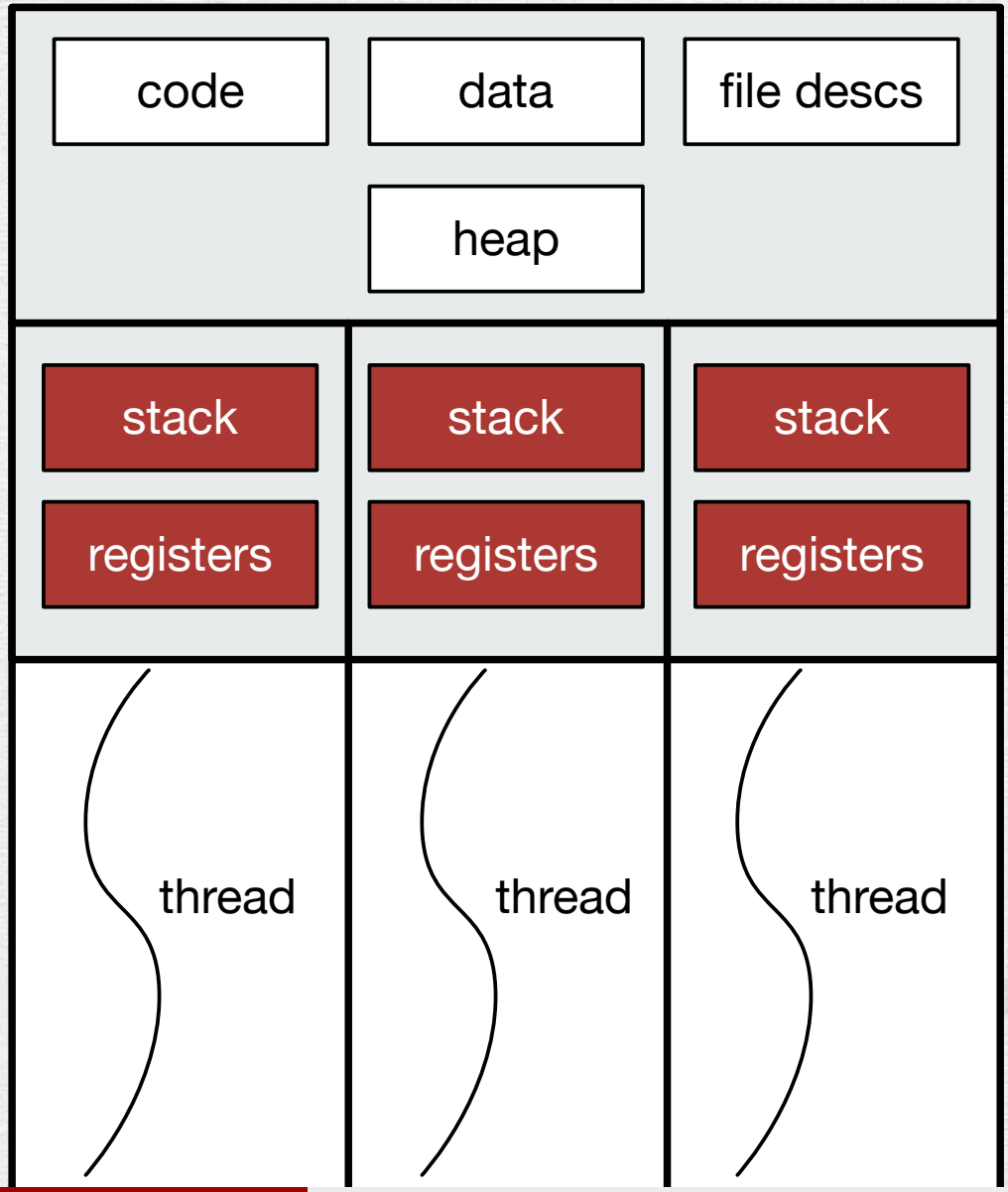
Single-threaded process

- Single stack
- Single text section (code)
- Single data section (global data)
- Single heap (dynamic allocation)

Multi-threaded process

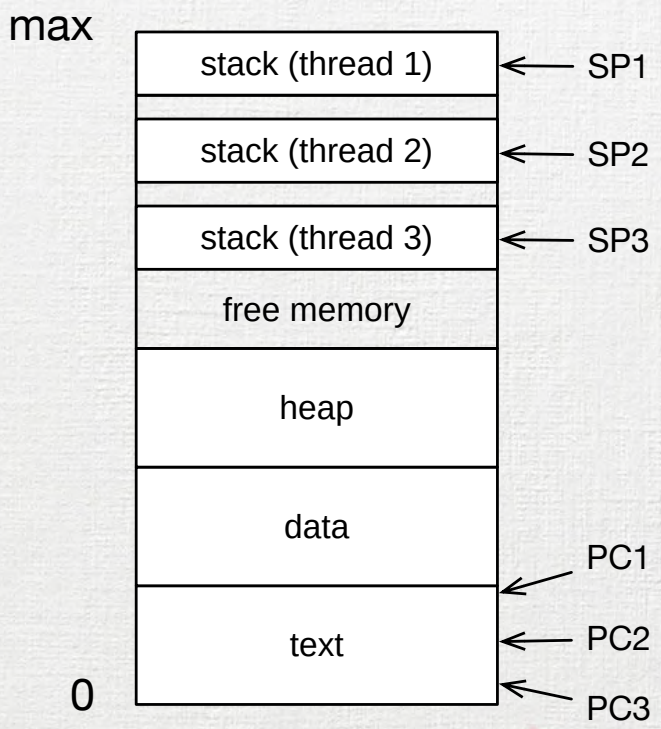
- More than one thread per process
- Share the same PCB among threads
  - Process state
  - Memory allocation (heap, global data)
  - File descriptors (files, sockets, etc.)
  - Scheduling information
  - Accounting information
- **Different** processor state (program counter, registers)
- **Different** stack

Multi-threaded process



Multi-threaded process

- Each thread has:
  - Private stack
  - Private stack pointer
  - Private program counter
  - Private register values
  - Private scheduling policies
- Share:
  - Common text section (code)
  - Common data section (global data)
  - Common heap (dynamic allocation)
  - File descriptors (opened files)
  - Signals...



Process memory space

Multi-threaded process vs Multi process

- Same goals
  - Do several things at the same time
  - Increase CPU utilization
  - Increase responsiveness
- What is the principal difference between these two types of process?
  - Multi-process with `fork()`: «resource cloning»
  - Multi-thread process: «resource sharing»

Why?

- Responsiveness
- Performance
- Resource Sharing
- Scalability

Responsiveness

- Perform different tasks **at the same time**
  - Several operations can block (e.g. network, disk I/O)
  - UI needs responsiveness

→ one thread for UI, other threads for background tasks

Performance

- Creating (`fork()`) a new process is slower than a thread
- Terminating a process is also slower than a thread
- Switching between processes is slower than between threads

Resource Sharing

- Memory is always shared
  - Heap
  - Global data
- All file descriptors are also shared
  - Open files
  - TCP sockets
  - UNIX sockets
  - Devices

integer identifiers OS uses to uniquely represent

- No need to use `shm*()`

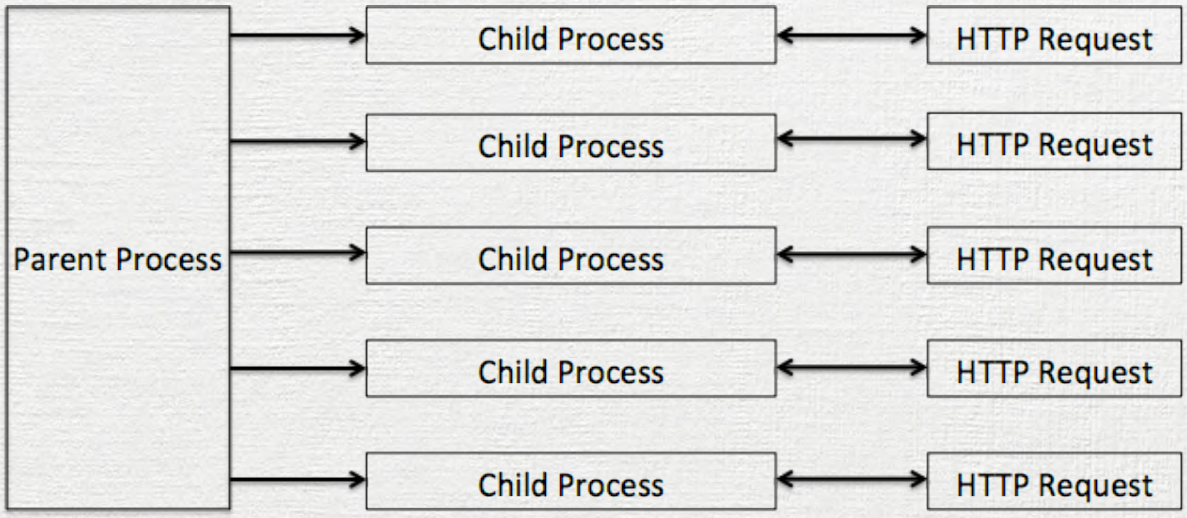
Scalability

- More CPU cores: simply increase number of threads
- Don't create too many threads
  - Overhead
  - Synchronization

Why **NOT** multi-thread?

- Threads are evil
  - Nondeterministic
  - Synchronization
  - Deadlocks
- Complication

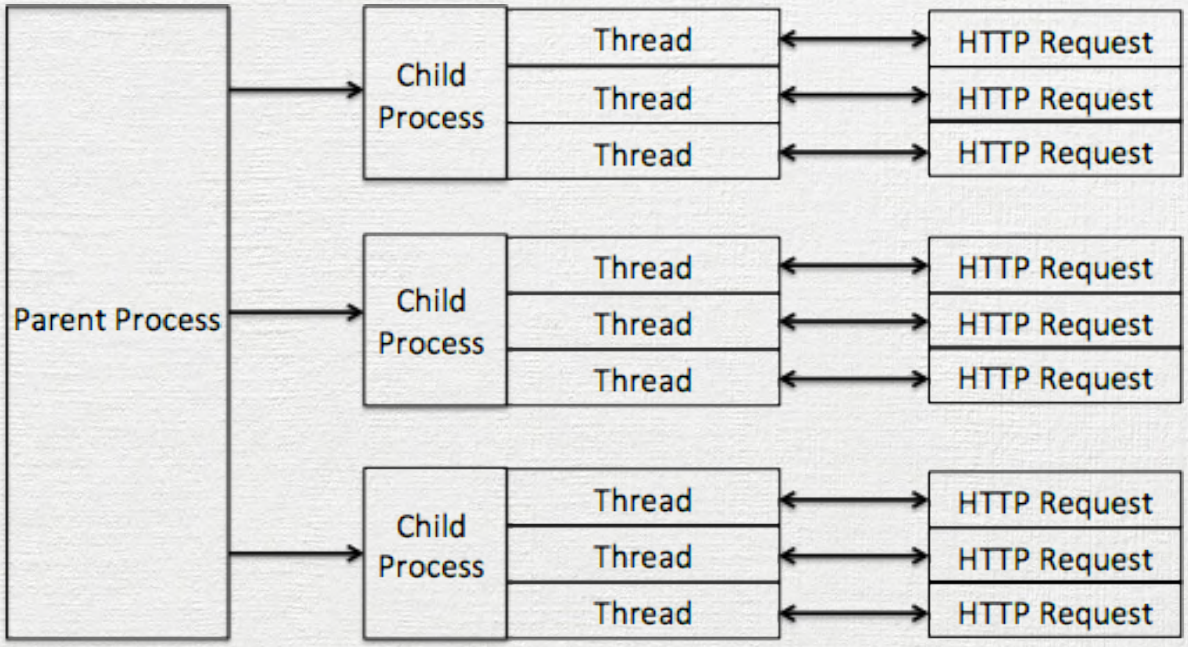
Multi-process real world app



Apache HTTPD Prefork Model[1]

[1] Image courtesy of Toni Miu's blog

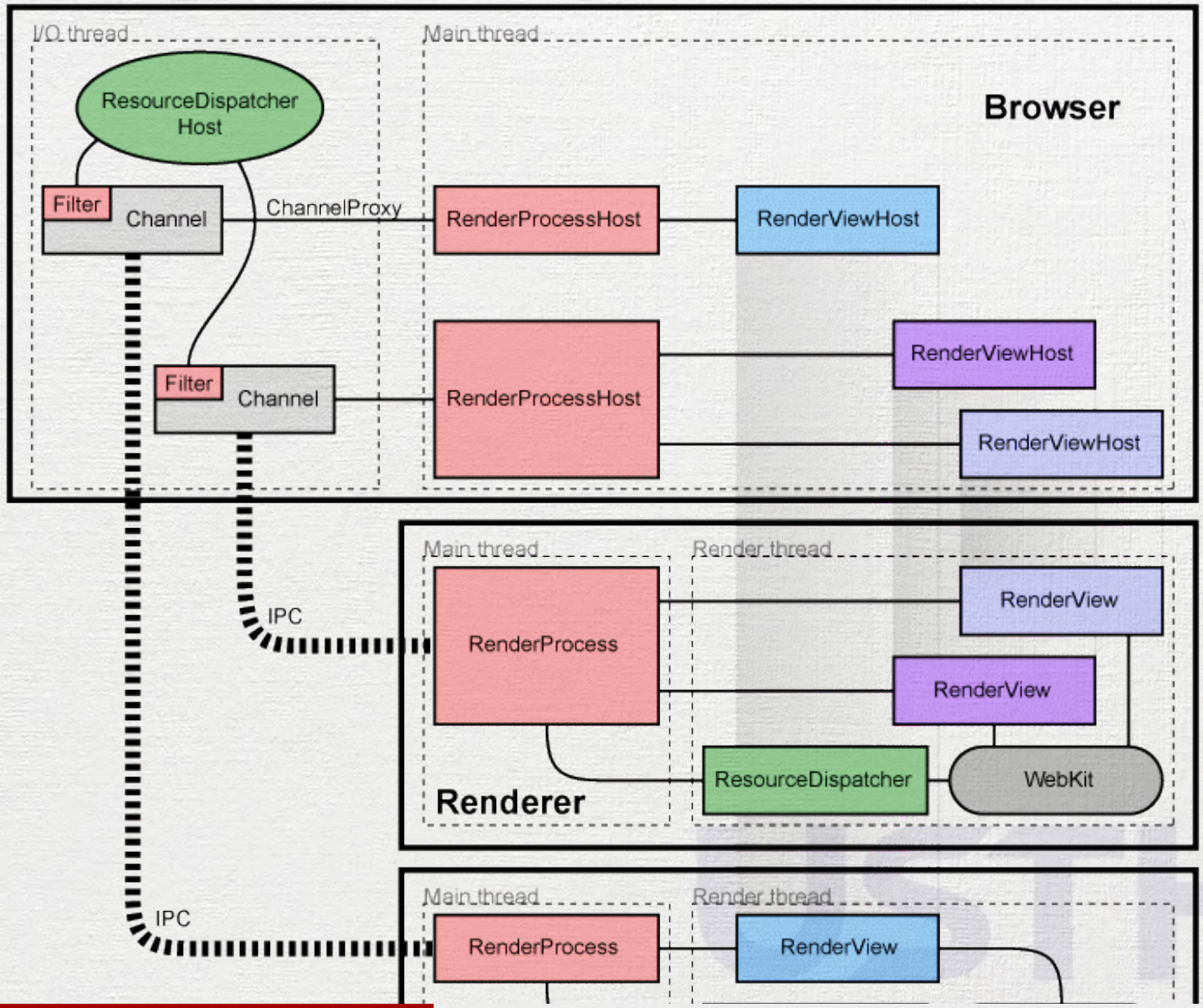
Multi-thread, multi-process, real world app



Apache HTTPD Worker Model[2]

[2]Image courtesy of Toni Miu's blog

Multi-thread, multi-process, real world app



Python threading

- Global Interpreter Lock
  - Implemented in CPython
  - Mutex
  - Only 1 thread can control the Python intepreter
  - Only one thread can be executed at any given time
    - Bottleneck in Python CPU-bound code
    - Not a problem in wrapper-to-native-code[3]
    - Not a problem in IO-bound programs

[3] e.g. numpy uses native libraries, so no GIL problem

Python threading

- Why GIL?
  - Memory management
    - Reference counting
    - Garbage collector
  - Simplification of thread-safety
    - Only 1 mutex on the intepreter
    - No multiple mutexes on each object
    - No deadlock
  - That's not a bug but a feature

Python threading

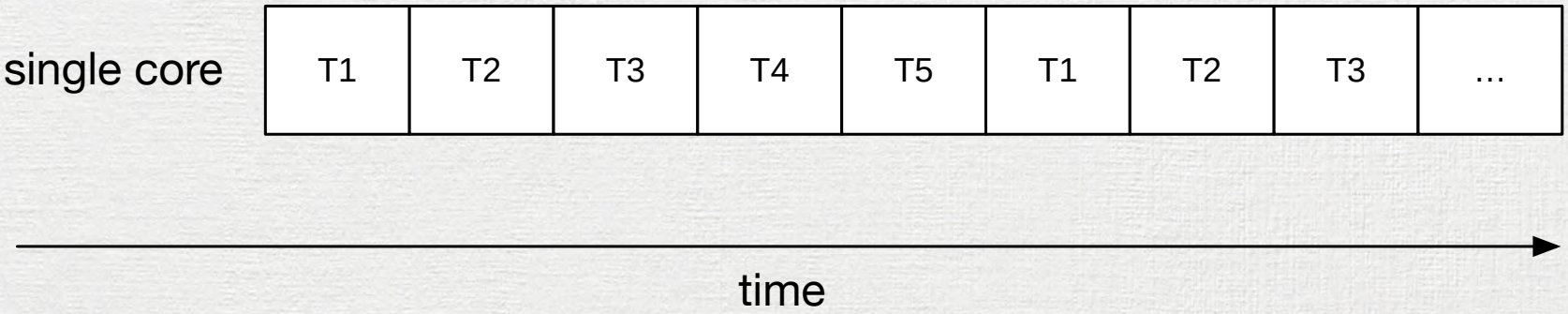
- Removing GIL?
  - Slower single-threaded performance
    - 1 mutex per object reference...
    - Potential deadlocks
  - Less compatbility

How?

- 2 «How» questions:
  - Q1: How does thread achieve concurrency?
  - Q2: How to use thread?

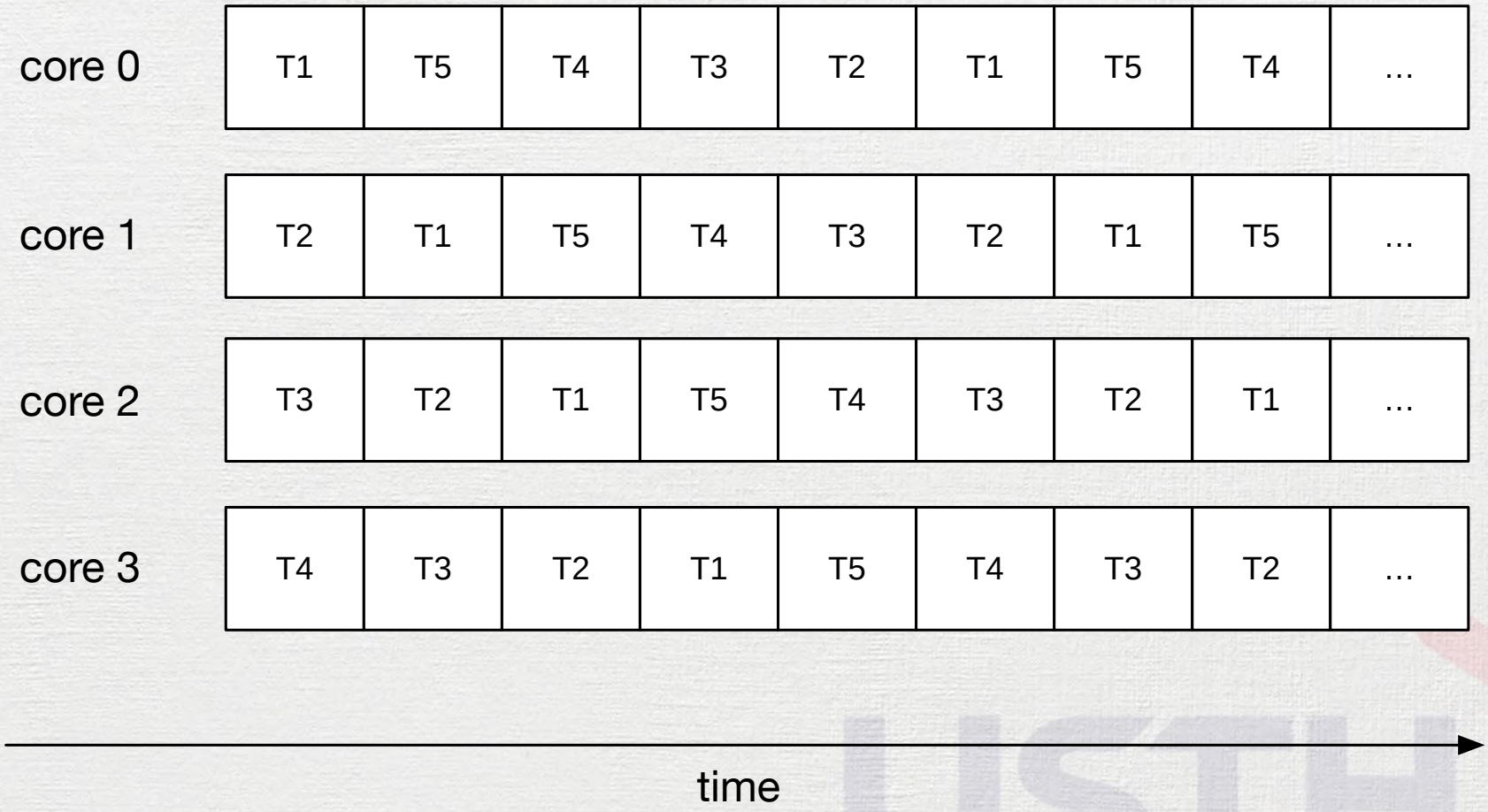
How (Q1): Concurrency on Single Core

- Q1: How does thread achieve concurrency?



How (Q1): Concurrency on Multi Cores

- Q1: How does thread achieve concurrency?



How (Q2): Using thread

- Use the module
- Subclass Thread
- Create new instance
- Launch the new thread
- [optional] Wait for thread to finish

How (Q2): Using thread

1. Use the module
2. Subclass Thread
3. Create new instance
4. Launch the new thread
5. Wait for thread to finish

- The `threading` module

```
import threading
```

How (Q2): Using thread

1. Use the module
2. Subclass Thread
3. Create new instance
4. Launch the new thread
5. Wait for thread to finish

- Define a subclass of `threading.Thread`
  - Override `run()` method to run in background
  - [optional] Implement `__init__()` method for passing parameters

How (Q2): Using thread

1. Use the module
2. Subclass Thread
3. Create new instance
4. Launch the new thread
5. Wait for thread to finish

```
class BackgroundThread(threading.Thread):
    def __init__(self, sleepTime):
        threading.Thread.__init__(self)
        self.__sleepTime = sleepTime

    def run(self):
        time.sleep(self.__sleepTime)
        print(f"Finished sleeping {self.__sleepTime}s")
```

How (Q2): Using thread

1. Use the module
2. Subclass Thread
3. Create new instance
4. Launch the new thread
5. Wait for thread to finish

- Create new instance of the thread class

```
backgroundThread = BackgroundThread(10)
```

How (Q2): Using thread

1. Use the module
2. Subclass Thread
3. Create new instance
4. Launch the new thread
5. Wait for thread to finish

- Launch the new thread with `.start()`
  - **NOT** `.run()`

```
backgroundThread.start()      # note no args here
```

How (Q2): Using thread

1. Use the module
2. Subclass Thread
3. Create new instance
4. Launch the new thread
5. Wait for thread to finish

- [optional] Wait for thread to finish with `.join()`

```
backgroundThread.join()
```

How (Q2): Using thread

```
import threading
import time

class BackgroundThread(threading.Thread):
    def __init__(self, sleepTime):
        threading.Thread.__init__(self)
        self.__sleepTime = sleepTime
    def run(self):
        time.sleep(self.__sleepTime)
        print(f"Finished sleeping {self.__sleepTime}s")

backgroundThread = BackgroundThread(10)
backgroundThread.start()    # note no args here
backgroundThread.join()
print("Finished main thread")
```

How: Extras

- Simple threading without subclassing:

```
def threadFunction(sleepTime):
    time.sleep(sleepTime)
    print(f"Finished sleeping {sleepTime}s")

t = threading.Thread(target=threadFunction, args=(10,))
t.start()
```

How: Extras

- Synchronization between threads `threading.Lock` (also called mutex)
  - `.acquire()`
  - `.release()`
  - Automatic `.acquire()` and `.release()` using `with` statement
- Be careful with race conditions while using `Lock`

How: Extras

```
lock = threading.Lock()
lock.acquire()
# do something dangerous here
lock.release()

with lock:
    # do something dangerous here
    print("Dangerous function")

# lock is released automatically
```

Practical work 8: multithreaded management system

- Copy your `pw6` directory into `pw8` directory
- Upgrade the persistence feature of your system to use `pickle` **in background thread**, still with compression
- Push your work to corresponding forked Github repository

What

- GUI
  - Graphical User Interface
  - Interactive with graphical components
    - Windows, scrollbars, buttons, textboxes,
  - Sexy (?)
- CLI
  - Command Line Interface
  - Writing commands in terminal
  - Wait for response from system
  - Old school, boring (?)

CLI vs GUI

Feature	CLI	GUI
Learning curve	Steep	Easy
Flexibility	Very high	Limited
Memory	Low	Higher
Speed	Fast	Slower
Interact	Keyboard	Keyboard, Mouse
Theming	Limited	Easy

Why

- User friendly, intuitive for new comers
- Easy to learn, no need to remember comments
- Better multitasking

Toolkits

Feature	Tkinter	PyQt	Kivy	wxPython
Included?	Yes	No	No	No
Cross platform	Yes	Yes	Yes	Yes
Backend	Tcl/Tk	Qt	OpenGL	wxWidgets

Tkinter

- Simplicity
- Flexibility
- Focusing on new comers

- TODO: image of student management system here

Tkinter

- Window
- Widgets
- Layout
- Window event loop

Tkinter: Window

1. Window
2. Widgets
3. Layout
4. Window event loop

- Types: Main window and sub window
- Important attributes: title, size
- Main window

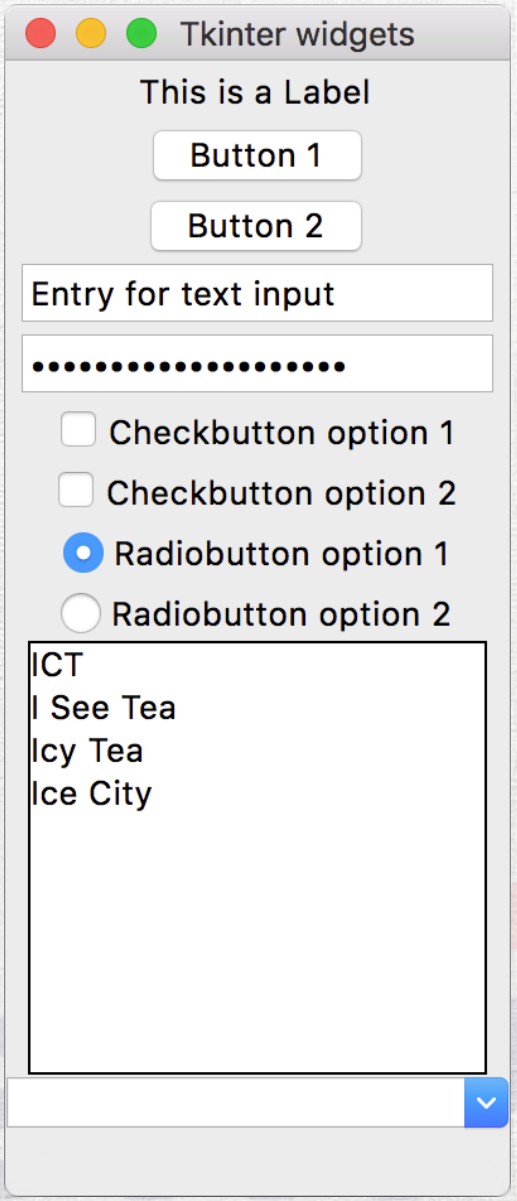
```
import tkinter as tk
window = tk.Tk()
window.title("Student Information System")
window.geometry("800x600")
```

- Sub window

```
sub = tk.Toplevel(window)
sub.title("Students")
sub.geometry("600x400")
```

Tkinter: Widgets

- Everything is widget
  - Frame
  - Label
  - Buttons
  - Entry
  - Check Button
  - Radio Button
  - List Box
  - ComboBox
  - Menu
  - ...
- Important attributes
  - Dimension: `width = 400`, `height = 300`
  - Background color: `bg = "green"`



Tkinter: Widgets

- Frame
  - A container for other widgets
  - `tk.Frame(window, width = 100, height = 100)`
- Label
  - Show texts
  - `tk.Label(window, text = "This is a Label")`

Tkinter: Widgets

1. Window
2. Widgets
3. Layout
4. Window event loop
5. Message box

- Button
  - Clickable
  - Handle click: `command = onClickFunc`

```
from tkinter import messagebox

def onClick():
    messagebox.showinfo(message="Button 1 clicked")

tk.Button(window, text = "Button 1", command = onClick)
```

Tkinter: Widgets

1. Window
2. Widgets
3. Layout
4. Window event loop
5. Message box

- Entry
  - Input texts
  - Can be used for password field (with `show = "*"`)

```
entry = tk.Entry(window)
entry.insert(-1, "Entry for text input")
```

- Checkbutton
  - Checkboxes
  - 2 states: check and uncheck
  - `tk.Checkbutton(window, text = "Checkbutton option 1")`

Tkinter: Widgets

1. Window
2. Widgets
3. Layout
4. Window event loop
5. Message box

- Radiobutton
  - Single choice among options
  - Text != Value

```
radioValue = tk.StringVar(value = "op1")
tk.Radiobutton(window, variable = radioValue,
    text = "Radiobutton option 1", value = "op1")
tk.Radiobutton(window, variable = radioValue,
    text = "Radiobutton option 2", value = "op2")
```

Tkinter: Widgets

1. Window
2. Widgets
3. Layout
4. Window event loop
5. Message box

- Listbox
  - A list of items
  - Selectable item
  - Get selected item: `listbox.get(tk.ACTIVE)`

```
icts = ["ICT", "I See Tea", "Icy Tea", "Ice City"]
listbox = tk.Listbox(window)
for i in icts:
    listbox.insert(icts.index(i), i)
```

Tkinter: Widgets

- Combobox
  - A list of selectable items
  - Initially collapsed, can be expanded
  - Get selected item: `combobox.get()`

```
from tkinter import ttk
icts = ["ICT", "I See Tea", "Icy Tea", "Ice City"]
ttk.Combobox(window, values = icts)
```

Tkinter: Layout

- Geometry manager
  - Handle placements (positions) of widgets on windows
  - Main container: `tk.Frame`
  - Widget methods for geometry management
    - `.pack()`
    - `.place()`
    - `.grid()`

Tkinter: Layout

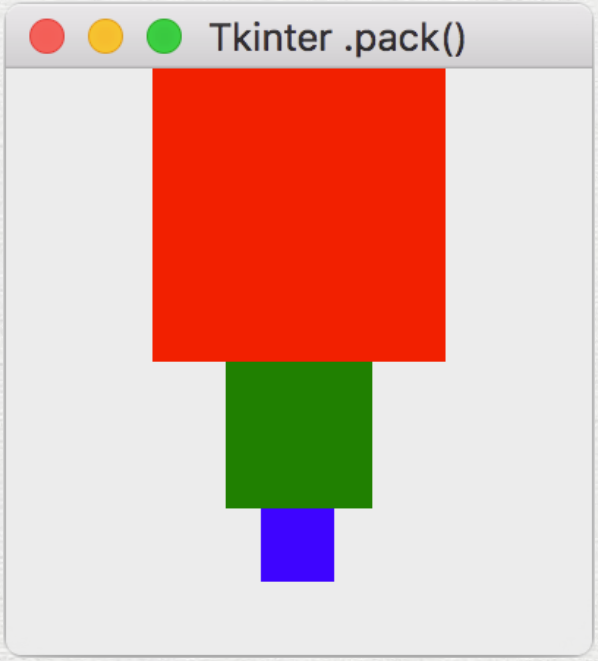
- `.pack()`
  - Packing algorithm
  - Similar to HTML `div`
  - Default
    - Vertically align
    - Horizontally centered
  - Alignment direction: `side = tk.LEFT`
  - Automatic expand: `fill = tk.X`, `fill = tk.Y`

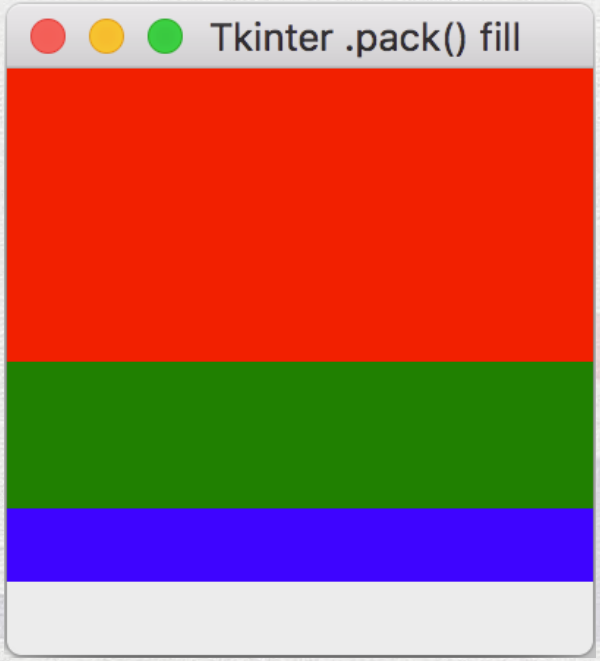
Tkinter: Layout

1. Window
2. Widgets
3. Layout
4. Window event loop
5. Message box

```
# default window
tk.Frame(window, width = 100, ..., bg="red").pack()
tk.Frame(window, width = 50, ..., bg="green").pack()
tk.Frame(window, width = 25, ..., bg="blue").pack()

# secondary window with fill
tk.Frame(sub, width = 100, ..., bg="red").pack(fill = tk.X)
tk.Frame(sub, width = 50, ..., bg="green").pack(fill = tk.X)
tk.Frame(sub, width = 25, ..., bg="blue").pack(fill = tk.X)
```





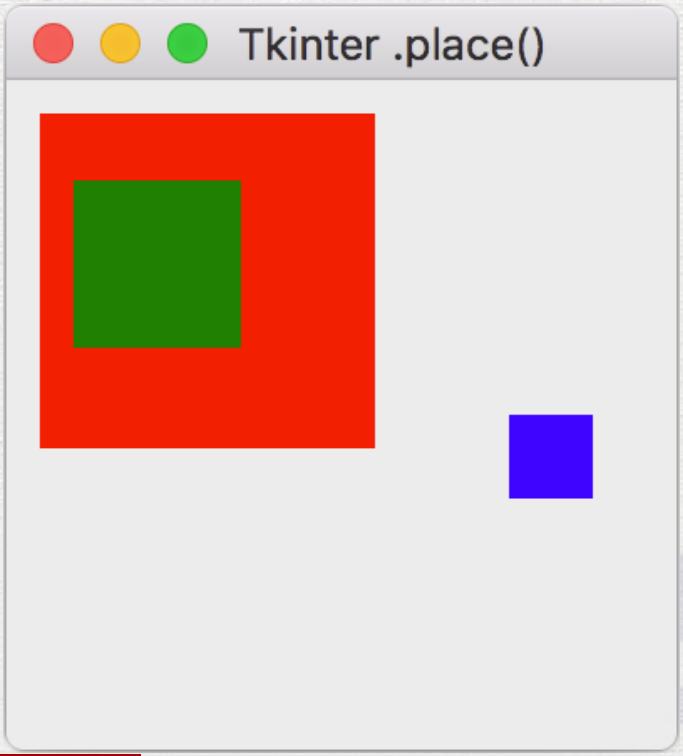
Tkinter: Layout

- `.place()`
  - Similar to HTML `position: absolute`
  - Unit: pixels
  - Absolute values
    - Position: `x = 10`, `y = 10`
    - Dimension: `width = 400`, `height = 300`
  - Relative values [0...1]
    - Position: `relx = 0.1`, `rely = 0.1`
    - Dimension: `relwidth = 0.5`, `relheight = 0.7`

Tkinter: Layout

```
tk.Frame(window, bg="red").place(
    x = 10, y = 10, width = 100, height = 100)
tk.Frame(window, bg="green").place(
    x = 20, y = 30, width = 50, height = 50)
tk.Frame(window, bg="blue").place(
    x = 150, y = 100, width = 25, height = 25)
```

Output:



1. Window
2. Widgets
3. Layout
4. Window event loop
5. Message box

Tkinter: Layout

- `.grid()`
  - Similar to HTML `table`
  - Position: `column = 0`, `row = 2`
  - Stretching: `sticky = tk.EW`
  - Padding: `padx = 3`, `pady = 3`
  - Spanning
    - `columnspan = 3`
    - `rowspan = 2`

Tkinter: Layout

1. Window
2. Widgets
3. Layout
4. Window event loop
5. Message box

```
tk.Label(window, text = "Username").grid(
    column = 0, row = 0, sticky = tk.EW, padx = 3, pady = 3)
tk.Label(window, text = "Password").grid(
    column = 0, row = 1, sticky = tk.EW, padx = 3, pady = 3)

tk.Entry(window).grid(
    column = 1, row = 0, sticky = tk.EW, padx = 3, pady = 3, columnspan = 4)
tk.Entry(window).grid(
    column = 1, row = 1, sticky = tk.EW, padx = 3, pady = 3, columnspan = 4)

tk.Button(window, text = "Login").grid(
    column = 0, row = 2, columnspan = 2)
tk.Button(window, text = "Exit").grid(
    column = 2, row = 2, columnspan = 2)
```

Tkinter: Layout

1. Window
2. Widgets
3. Layout
4. Window event loop
5. Message box

Tkinter: Window loop

- A **blocking** method
- Handles input, output events
- `window.mainloop()`

Practical work 9: GUI'ed management system

- Copy your `pw8` directory to `pw9` directory
- Upgrade your user interface to GUI using Tkinter
- Push your work to corresponding forked Github repository