Modern programming languages for HEP

Sébastien Ponce
sebastien.ponce@cern.ch

CERN

Thematic CERN School of Computing 2021

Goal of this course

- Make a tour of latest improvements in HEP programming languages
  - C++ and python
- Understand
  - the use cases of each language
  - the evolution of C++
  - how this impacts performances
- Make a quick tour of python 3 changes
  - and help migrating

Outline

Why python and C^{++}

Why python and C^{++}

1. Why python and C^{++}
   - Pros and Cons of each language
   - Respective usecases
2. C^{++}getting usable
3. Performant C^{++}
4. Migrating from Python 2 to python 3
5. Conclusion

C^{++} pros and cons

Adapted to large projects

- strongly typed, object oriented
- widely used (and taught) with many available libraries

C^{++} pros and cons

Adapted to large projects

- strongly typed, object oriented
- widely used (and taught) with many available libraries

Fast

- compiled (unlike Java or C#)
- allows to go close to hardware when needed

C^{++} pros and cons

Adapted to large projects

- strongly typed, object oriented
- widely used (and taught) with many available libraries

Fast

- compiled (unlike Java or C#)
- allows to go close to hardware when needed

What we get

- the most powerful language
- the most complicated one
- the most error prone ?

python pros and cons

Adapted to large projects

- multi-paradigm language (object oriented, functional ...)
- widely used (and taught) with many available libraries

python pros and cons

Adapted to large projects

- multi-paradigm language (object oriented, functional ...)
- widely used (and taught) with many available libraries

Easy to use and ubiquitous

- interpreted, supported on all platforms
- versatile : usages from ML to web dev or numeric code
- smooth learning curve, integrated with online tools (SWAN)
- compatible with C++, critical code can be written in C++ in the back

python pros and cons

Adapted to large projects

- multi-paradigm language (object oriented, functional ...)
- widely used (and taught) with many available libraries

Easy to use and ubiquitous

- interpreted, supported on all platforms
- versatile : usages from ML to web dev or numeric code
- smooth learning curve, integrated with online tools (SWAN)
- compatible with C^{++}, critical code can be written in C^{++}in the back

The price to pay

- not suitable for performance
- error prone (no strong typing)

Evolving languages

C^{++} got 4 major releases in 10 years

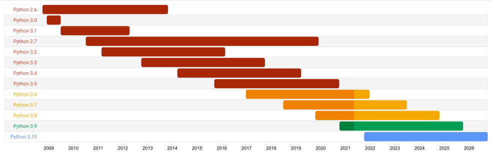
- one every 3 years
- major changes and improvements
- almost a new language

C++ standards

Year	C++ Standard	Informal name
1998	ISO/IEC 14882:1998[29]	C++98
2003	ISO/IEC 14882:2003[30]	C++03
2011	ISO/IEC 14882:2011[31]	C++11, C++0x
2014	ISO/IEC 14882:2014[32]	C++14, C++1y
2017	ISO/IEC 14882:2017[33]	C++17, C++1z
2020	ISO/IEC 14882:2020[12]	C++20, C++2a

python went to version 3

- major, backward incompatible changes
- initial release in 2008
- latest release 3.9
- widely adopted only in the last 5 years



Why python and C^{++}

A language for each task

C++

- The definite winner for performance critical code
- Also to be used for large, complex frameworks

python

- The definite winner for configuration
- Also to be used for “glue code”
- In general end-user facing code

C++ getting usable

C++ getting usable

C^{++} is becoming "simpler"

With the C^{++} conception of "simpler"

- new and much nicer ways to write code
- backward compatibility insured
  - so the language is overall (much) more complex

Most noticable features

- range based loops
- auto keyword
- lambdas
- ranges
- <=>

Range based loops

Reason of being

- simplifies loops tremendously
- especially with STL containers

Syntax

```
for ( type iteration_variable : container ) {
  // body using iteration_variable
}
```

Example code

```
std::vector<int> v{1,2,3,4};
int prod = 1;
for (int a : v) { prod *= a; } // pls use std::accumulate
```

Auto keyword

Reason of being

- many type declarations are redundant
- and lead to compiler error if you mess up

```
std::vector<int> v;
int a = v[3];
int b = v.size();  // bug ? unsigned to signed
```

Auto keyword

Reason of being

- many type declarations are redundant
- and lead to compiler error if you mess up

```
std::vector<int> v;
int a = v[3];
int b = v.size();  // bug ? unsigned to signed
```

Practical usage

```
std::vector<int> v;
auto a = v[3];
auto b = v.size();
int sum{0};
for (auto n : v) { sum += n; }
```

CERN School of Computing

Lambdas

Definition

a lambda is a function with no name

Syntax

```
[captures] (args) -> type { code; }
```

The type specification is optional

Usage example

```
int sum = 0, offset = 1;
std::vector<int> data{1,9,3,8,3,7,4,6,5};
for_each(begin(data), end(data),
         [&sum, offset](int x) {
             sum += x + offset;
         });
```

Ranges ($C^{++}20$)

Reason of being

- provide easy manipulation of sets of data via views
- simplify the horrible iterator syntax

Syntax

Based on Unix like pipes, and used in range based loops

Example code - godbolt

```
std::vector<int> numbers{...};
auto results =
  numbers | filter([](int n){ return n % 2 == 0; })
          | transform([](int n){ return n * 2; });
for (auto v: results) std::cout << v << " ";
```

So far essentially syntactic sugar

Range based loops

```
for (int a : v) { sum *= a; }
```

Translate to iterators

```
for (auto it = begin(v); it != end(v); it++) {
  sum *= *it;
}
```

So far essentially syntactic sugar

Lambdas

```
[&sum, offset](int x) { sum += x + offset; }
```

Are just functors

```
struct MyFunc {
  int& m_sum;
  int m_offset;
  MyFunc(int& s, int o) : m_sum(s), m_offset(o) {}
  int operator(int x) { m_sum += x + m_offset; }
};
MyFunc(sum, offset)
```

By the way, as lambdas are functors, they can inherit from each other !
And this can be super useful.

C^{++}getting usable

What makes C^{++} hard ?

The many pitfalls you can fall in

- ugly C syntax, inherited
- pointers, memory management
- thread safety issues
- and locking
- horrible metaprogramming
- lack of modularity

All this has been corrected

Each pitfall is being "solved"

- ugly C syntax → enum class, std::variant, std::any
- pointers, memory management → "smart" pointers
- thread safety issues → constness
- dead locks → "smart" locks
- horrible metaprogramming → concepts
- bad code modularity → modules

Notes :

- constness is covered in next talk
- I won't cover concepts and modules
  - we would need (much) more time

enum class, aka scoped enum

Same syntax as enum, with scope

```
enum class VehicleType { Bus, Car };
VehicleType t = VehicleType::Car;
```

enum class, aka scoped enum

Same syntax as enum, with scope

```
enum class VehicleType { Bus, Car };
VehicleType t = VehicleType::Car;
```

Only advantages over enums

- scoping avoids name clashes
- strong typing, no automatic conversion to int

```
enum VType { Bus, Car }; enum Color { Red, Blue };
VType t = Bus;
if (t == Red) { // We do enter !  }
int a = 5 * Car; // Ok, a = 5

enum class VT { Bus, Car }; enum class Col { Red, Blue };
VT t = VT::Bus;
if (t == Col::Red) { // Compiler error }
int a = t * 5;       // Compiler error
```

std::variant, std::any

Purpose

- type safe union and “void*”
- with visitor pattern

std::variant, std::any

Purpose

- type safe union and “void*”
- with visitor pattern

Example code - godbolt

```
using Message = std::variant<int, std::string>;
Message createMessage(bool error) {
  if (error) return "Error"; else return 42;
}
struct Visitor {
  void operator()(int n) const {
    std::cout << "Int " << n << std::endl;
  }
  void operator()(const std::string &s) const {
    std::cout << "String \"" << s << "\"" << std::endl;
  }
};
std::visit(Visitor{}, createMessage(true));
```

thematic CSC CERN School of Computing

std::variant, std::any

Or you use lambdas and their inheritance - godbolt

```
template <class ... P> struct Combine : P... {
  using P::operator()...;
};
template <class ... F> Combine<F...> combine(F... fs) {
  return { fs ... };
}
using Message = std::variant<int, std::string>;
Message createMessage(bool error) {
  if (error) return "Error"; else return 42;
}
auto f = combine(
    [](int n) { std::cout << "Int " << n << std::endl; },
    [](string const &s) {
      std::cout << "String \"" << s << "\"" << std::endl;
    });
std::visit(f, createMessage(true));
```

Pointer management : RAII

Resource Acquisition Is Initialization

Practically

Use object semantic to acquire/release resources (e.g. memory)

- wrap the resource inside an object (e.g. a smart pointer)
- acquire resource via object constructor (call to new)
- release resource in destructor (call to delete)
- create this object on the stack so that it is automatically destructed when leaving the scope, including in case of exception

RAII in practice

File class

```
class File {
public:
  File(const char* filename) :
    m_file_handle(std::fopen(filename, "w+")) {
    if (m_file_handle == NULL) { throw ... }
  }
  ~File() { std::fclose(m_file_handle); }
  }
private:
  FILE* m_file_handle;
};
void foo() {
  // file opening, aka resource acquisition
  File logfile("logfile.txt") ;
  ...
  // file is automatically closed by the call to
  // its destructor, even in case of exception !
}
```

std::unique_ptr

an RAII pointer

- wraps a regular pointer
- has move only semantic
  - the pointer is only owned once
- in <memory> header

std::unique_ptr

an RAII pointer

- wraps a regular pointer
- has move only semantic
  - the pointer is only owned once
- in <memory> header

Usage

```
void f(std::unique_ptr<Foo> ptr);
{
  auto uptr = make_unique<Foo>();  // calling constructor
  std::cout << uptr->someMember << std::endl;
  std::cout << "Points to : " << uptr.get() << std::endl;
  f(std::move(uptr)); // transfer of ownership
  // memory is deallocated when f exits
}
```

std::shared_ptr

shared_ptr : a reference counting pointers

- wraps a regular pointer like unique_ptr
- has move and copy semantic
- uses internally reference counting
  - "Would the last person out, please turn off the lights ?"
- is thread safe, thus the reference counting is costly

make_shared : creates a shared_ptr

```
{
  auto sp = std::make_shared<Foo>(); // #ref = 1
  vector.push_back(sp);              // #ref = 2
  set.insert(sp);                    // #ref = 3
} // #ref 2
```

Modern C^{++} and pointers

Main rules

- use references rather than pointers
- no more calls to new or delete
  - only make_unique
  - exceptionally make_shared

```
void f(Foo const& arg);
auto p = std::make_unique<Foo>();
f(*p);
```

Consequences

- Forget seg faults due to null pointers
- Forget memory leaks

RAII applied to locking

Wrappers around `std::mutex`

std::scoped_lock for any number of locks

std::lock_guard for a single regular lock

- lock taken on construction
- released on destruction
- `scoped_lock` includes deadlock management

std::unique_lock same as `lock_guard` and can be released/relocked

Practically

```
int a = 0;
std::mutex m;
void inc() {
  std::scoped_lock guard{m};
  a++;
}; // Horribly inefficient code !!!
```

Performant C^{++}

1. Why python and C^{++}
2. C^{++}getting usable
3. Performant C^{++}
   - New performance related features
   - Templates
   - Avoiding virtuality when possible
4. Migrating from Python 2 to python 3
5. Conclusion

Performant C^{++}

Features related to performance

Main improvements in $C^{++}11$ and later

- noexcept
- around memory allocation
  - reserve, emplace, ... See next talk
- move semantic and copy elision
- templating and variadic templating

C^{++}exception support

After a lot of thinking and experiencing, the conclusions of the community on exception handling are :

- Never write an exception specification
- Except possibly an empty one

C++ exception support

After a lot of thinking and experiencing, the conclusions of the community on exception handling are :

- Never write an exception specification
- Except possibly an empty one

one of the reasons : performance

- does not allow compiler optimizations
- on the contrary forces extra checks

C^{++}exception support

After a lot of thinking and experiencing, the conclusions of the community on exception handling are :

- Never write an exception specification
- Except possibly an empty one

one of the reasons : performance

- does not allow compiler optimizations
- on the contrary forces extra checks

Introducing noexcept

```
int f() noexcept;
```

- somehow equivalent to throw()
- meaning no exception can go out of the function
- but is checked at compile time
- thus allowing compiler optimizations

Impact on generated code - exceptions

```
struct MyExcept{};
int f(int a); // may throw

int foo() {
  try {
    int a = 23;
    return f(a) + f(-a);
  } catch (MyExcept& e) {
    return 0;
  }
}
```

Generated code (godbolt , gcc10, -O3)

```
foo():
        push    rbx
        mov     edi, 23
        call    f(int)
        mov     edi, -23
        mov     ebx, eax
        call    f(int)
        add     eax, ebx
.L1:
        pop     rbx
        ret
        mov     rdi, rax
        mov     rax, rdx
        jmp     .L2
foo() [clone .cold]:
.L2:
        sub     rax, 1
        jne     .L8
        call    __cxa_begin_catch
        call    __cxa_end_catch
        xor     eax, eax
        jmp     .L1
.L8:
        call    _Unwind_Resume
```

Impact on generated code - noexcept

```
struct MyExcept{};
int f(int a) noexcept;

int foo() {
  try {
    int a = 23;
    return f(a) + f(-a);
  } catch (MyExcept& e) {
    return 0;
  }
}
```

Generated code (godbolt , gcc10, -O3)

```
foo():
        push    rbx
        mov     edi, 23
        call    f(int)
        mov     edi, -23
        mov     ebx, eax
        call    f(int)
        add     eax, ebx
        pop     rbx
        ret
```

Move semantics

The idea

- a new type of reference : rvalue references
  - used for “moving” objects
  - denoted by &&
- 2 new members in every class, with move semantic :

  a move constructor similar to copy constructor

  a move assignment operator similar to assignment operator (now called copy assignment operator)

  - used when original object can be reused

CERN School of Computing

Move semantics

The idea

- a new type of reference : rvalue references
  - used for “moving” objects
  - denoted by &&
- 2 new members in every class, with move semantic :

  a move constructor similar to copy constructor

  a move assignment operator similar to assignment operator (now called copy assignment operator)

  - used when original object can be reused

Practically

```
T(const T&  other); // copy construction
T(       T&& other); // move construction
T& operator=(const T&  other); // copy assignment
T& operator=(       T&& other); // move assignment
```

Move semantics

A few important points concerning move semantic

- the whole STL can understand the move semantic
- move assignment operator is allowed to destroy source
  - so do not reuse source afterward
- if not implemented, move falls back to copy version
- move is called by the compiler whenever possible
  - e.g. when passing temporary

Move semantics

A few important points concerning move semantic

- the whole STL can understand the move semantic
- move assignment operator is allowed to destroy source
  - so do not reuse source afterward
- if not implemented, move falls back to copy version
- move is called by the compiler whenever possible
  - e.g. when passing temporary

Practically

```
T a;
T b = a;       // 1. Copy assign
T c = T(2);    // 2. Move assign
T d = func();  // 3. Move assign
```

Move semantics gains

Essentially targetting containers or fat classes

- "moving" the content of a vector avoids copying
- only copies the underlying pointer to the data
- and is thus essentially as efficient as copying an integer !

Move semantics gains

Essentially targetting containers or fat classes

- "moving" the content of a vector avoids copying
- only copies the underlying pointer to the data
- and is thus essentially as efficient as copying an integer !

Zero gain for plain structs

- all members still have to be "copied"
- move can only help if a member "points" to some other data

Transform
float x,y,z; float rot[9];

TransVec
Transform* trs;

```
Transform( Transform&& o ) :
  x(o.x), y(o.y), z(o.z),
  rot(o.rot) {}
TransVec( TransVec&& o ) :
  trs(o.trs) { o.trs = nullptr; }
```

Guaranteed copy elision

What is copy elision

```
struct Foo { ... };
Foo f() {
  return Foo();
}
int main() {
  // compiler was authorised to elude the copy
  Foo foo = f();
}
```

From C++17 on

The elision is guaranteed.

- superseeds move semantic in some cases
- so do not hesitate anymore to return plain objects in generators
  - and ban pointers for good

Performant C^{++}

Templates

Concept

- The C++ way to write reusable code
  - aka macros on steroids
- Applicable to functions and objects

```
template<typename T>
const T & max(const T &A, const T &B) {
  return A > B ? A : B;
}

template<typename T>
struct Vector {
  int m_len;
  T* m_data;
};
```

Templates

Warning

These are really like macros

- they need to be defined before used
  - so all templated code has to be in headers
- they are compiled n times
- and thus each version is optimized individually !

```
template<typename T>
T func(T a) {
  return a;
}
```

func(3) →

```
int func(int a) {
  return a;
}
```

func(5.2) →

```
double func(double a) {
  return a;
}
```

Templates

Specialization

templates can be specialized for given values of their parameter

```
template<typename F, unsigned int N> struct Polygon {
  Polygon(F radius) : m_radius(radius) {}
  F perimeter() {return 2*N*sin(PI/N)*m_radius;}
  F m_radius;
};

template<typename F>
struct Polygon<F, 6> {
  Polygon(F radius) : m_radius(radius) {}
  F perimeter() {return 6*m_radius;}
  F m_radius;
};
```

The Standard Template Library

What it is

- A library of standard templates
- Everything you need, or ever dreamed of
  - strings, containers, iterators
  - algorithms, functions, sorters
  - functors, allocators
  - ...
- Portable
- Reusable
- Efficient

The Standard Template Library

What it is

- A library of standard templates
- Everything you need, or ever dreamed of
  - strings, containers, iterators
  - algorithms, functions, sorters
  - functors, allocators
  - ...
- Portable
- Reusable
- Efficient

Just use it

and adapt it to your needs, thanks to templates

Performant C^{++}

Virtuality in a nutshell

Principle

- a base class (aka interface) declares some method virtual
- children can overload these methods (as any other)
- for these method, late binding is applied
- that is most precise type is used

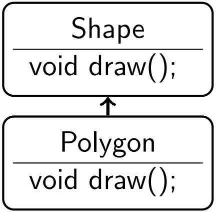
Virtuality in a nutshell

Principle

- a base class (aka interface) declares some method virtual
- children can overload these methods (as any other)
- for these method, late binding is applied
- that is most precise type is used

```
Polygon p;
p.draw(); // Polygon.draw

Shape & s = p;
s.draw(); // Shape.draw
```



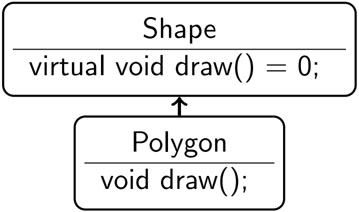
Virtuality in a nutshell

Principle

- a base class (aka interface) declares some method virtual
- children can overload these methods (as any other)
- for these method, late binding is applied
- that is most precise type is used

```
Polygon p;
p.draw(); // Polygon.draw

Shape & s = p;
s.draw(); // Polygon.draw
```



The price of virtuality

Actual implementation

- each object has an extra pointer
- to a "virtual table" object in memory
- where each virtual function points to the right overload

Cost

- extra virtual table in memory, per type
- each virtual call does
  - retrieve virtual table pointer
  - load virtual table into memory
  - lookup right call
  - effectively call
- and is thus much more costful than standard function call
- up to 20% difference in terms of nb of instructions

Actual price of virtuality

Comparison with templates - godbolt / godbolt

```
struct Interface {
  virtual void tick(float n) = 0;
};
struct Counter : Interface {
  float sum{0};
  void tick(float v) override
    { sum += v; }
};
void foo(Interface& c) {
  for (int i = 0; i < 80000; ++i) {
    for (int j = 0; j < i; ++j) {
      c.tick(j);
    }
  }
}
int main() {
  auto obj = std::make_unique<Counter>();
  foo(*obj);
  // ... print ...
}
```

```
struct Counter {
  float sum{0};
  void tick(float v) { sum += v; }
};
template<typename CounterType>
void foo(CounterType& c) {
  for (int i = 0; i < 80000; ++i) {
    for (int j = 0; j < i; ++j) {
      c.tick(j);
    }
  }
}
```

Actual price of virtuality

Comparison with templates - godbolt / godbolt

```
struct Interface {
  virtual void tick(float n) = 0;
};
struct Counter : Interface {
  float sum{0};
  void tick(float v) override
    { sum += v; }
};
void foo(Interface& c) {
  for (int i = 0; i < 80000; ++i) {
    for (int j = 0; j < i; ++j) {
      c.tick(j);
    }
  }
}
int main() {
  auto obj = std::make_unique<Counter>();
  foo(*obj);
  // ... print ...
}
```

```
struct Counter {
  float sum{0};
  void tick(float v) { sum += v; }
};
template<typename CounterType>
void foo(CounterType& c) {
  for (int i = 0; i < 80000; ++i) {
    for (int j = 0; j < i; ++j) {
      c.tick(j);
    }
  }
}
```

Timing	Time(s)	Nb instr(G)
virtual	10.8	35.2
templ	2.97	12.0

- measured on EPYC 7552, with gcc 9.1 and perf

A few explanations

Some consequences of virtuality

- more branching, killing the pipeline
  - here 6.4M vs 0.8M branches !
  - as virtual calls are branches
- lack of inlining possibilities
- lack of optimizations after inlining
  - e.g. auto vectorization

Note that the compiler is trying hard to help

- when it can, when it knows so give it all the knowledge !
- typical on my example

A few explanations

Some consequences of virtuality

- more branching, killing the pipeline
  - here 6.4M vs 0.8M branches !
  - as virtual calls are branches
- lack of inlining possibilities
- lack of optimizations after inlining
  - e.g. auto vectorization

Note that the compiler is trying hard to help

- when it can, when it knows so give it all the knowledge !
- typical on my example
  - declare `obj` on the stack and the compiler will “drop” virtuality
    - again : drop pointers !
  - gcc 10 does much better : 22G instructions and 3s

Should I use virtuality ?

Yes, when you cannot know anything at compile time

Typical cases

- you have no knowledge of the implementations of an interface
  - new ones may even be loaded dynamically via shared libraries
- you mix various implementations in a container
  - e.g. `std::vector<MyInterface>`
  - and there is no predefined set of implementations

Should I use virtuality ?

Yes, when you cannot know anything at compile time

Typical cases

- you have no knowledge of the implementations of an interface
  - new ones may even be loaded dynamically via shared libraries
- you mix various implementations in a container
  - e.g. `std::vector<MyInterface>`
  - and there is no predefined set of implementations

Typical alternatives

- templates when everything is compile time
  - allows full optimization of each case
  - and even static polymorphism through CRTP
    - Curiously recurring template pattern
- `std::variant`, `std::any` and visitor
  - when type definitions are known at compile type
  - but not necessary their usage

A Visitor example - godbolt

```
struct Point { virtual float getR() = 0; };
struct XYZPoint : Point {
  float x, y, z;
  float getR() override { return std::sqrt(x*x+y*y+z*z); }; };
struct RTPPoint : Point {
  float r, theta, phi;
  float getR() override { return r; } };
float sumR(std::vector<std::unique_ptr<Point>>& v) {
  return std::accumulate(begin(v), end(v), 0.0f,
  [&](float s, std::unique_ptr<Point>& p) { return s + p->getR();} );
}

struct XYZPoint { float x,y,z; };   struct RTPPoint { float r, theta, phi; };
using Point=std::variant<XYZPoint, RTPPoint>;
float sumR(std::vector<Point>& v) {
  auto getR = combine(
      [](XYZPoint& p) { return std::sqrt(p.x*p.x+p.y*p.y+p.z*p.z); },
      [](RTPPoint& p) { return p.r; });
  return std::accumulate(begin(v), end(v), 0.0f,
      [&](float s, Point& p) { return s + std::visit(getR, p);} );
}
```

CERN School of Computing

A Visitor example - godbolt

```
struct Point { virtual float getR() = 0; };
struct XYZPoint : Point {
  float x, y, z;
  float getR() override { retur                    y*y+z*z); }; };
struct RTPPoint : Point {
  float r, theta, phi;
  float getR() override {                   };
float sumR(std::vector<std          ptr<Point>>& v) {
  return std::accumulate(begin(v), end(v), 0.0f,
  [&](float s, std::unique_ptr<Point>& p) { return s + p->getR();} );
}
```

took 3500µs

```
struct XYZPoint { float x,y,z; };   struct RTP      t { float r, theta, phi; };
using Point=std::variant<XYZPoint, RTPPoi
float sumR(std::vector<Point>& v) {
  auto getR = combine(
      [](XYZPoint& p) { return                     p.y*p.y+p.z*p.z); },
      [](RTPPoint& p) { return
  return std::accumulate(begin(                 , 0.0f,
      [&](float s, Point& p) { re          s + std::visit(getR, p);} );
}
```

took 2050µs

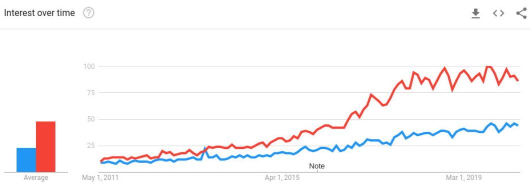
Migrating from Python 2 to python 3

Why python 3 ? Should we migrate ?

Reasons for python 3

- rectify fundamental design flaws in python2
- allow for non backward compatible changes

Reasons to migrate



- python3 has clearly taken over
- python 2 is no more maintained
  - official end of life : December 31^{st} 2019
- most libraries have dropped support for python2
  - pip, numpy, matplotlib, jupyter, pytorch, ...

Migrating from Python 2 to python 3

Backward incompatible changes

print statement became a function

```python
# python 2
print "this is python", 2
```

```python
# python 3
print("this is python", 3)
```

integer division has changed

```python
# python 2
assert( 3 / 2 == 1 )
assert( 3 // 2 == 1 )
```

```python
# python 3
assert( 3 / 2 == 1.5 )
assert( 3 // 2 == 1 )
```

strings are now unicode

```python
# python 2
s = 'string, aka str'
bs = b'string, aka str
us = u'unicode object'
```

```python
# python 3
s = 'unicode, aka str'
bs = b'bytes'
us = u'unicode, aka str'
```

Removed legacy syntax

Exceptions syntax has changed

```python
# python 2
try:
  raise ValueError, "msg"
except ValueError, e:
  ...
```

```python
# python 2 or 3
try:
  raise ValueError("msg")
except ValueError as e:
  ...
```

looping on dictionnary changed

```python
# python 2
d = {1:1, 2:2}
for k in d.keys(): ...
```

```python
# python 2 or 3
d = {1:1, 2:2}
for k in d: ...
```

Many other small points

- ranges, metaclasses, backticks, imports, input, ...

Migrating from Python 2 to python 3

CERN School of Computing

Supporting both python2 and python3

Best strategy

- migrate to python 3
- make python 3 code compatible with python 2, only if needed !
  - by modernizing code
    - "modern" python code is compatible with both 2 and 3
  - by extending python2 so that it understands python3 constructs
  - through the use of `__future__`

Practically

```
# valid both in python 2 and 3
from __future__ import division, print_function
a = 3 / 2
print(a)
# outputs 1.5
```

Migrating from Python 2 to python 3

Migrating code

Use 2to3 or futurize tool

- provided in python3 distribution
- "turns code into valid Python 3 code, and then adds `__future__` and future package imports to re-enable compatibility with Python 2"

Revalidate every single line by hand...

- very often generated code is too verbose
- from time to time, it does not work
- and python lose type checking does not help

The essential point

Have a damn good test suite with high coverage

Conclusion

Conclusion

Key messages of the day

- $C^{++}$and python are complementary and compatible
  - together they allow for full performance and easiness of use
  - they are both evolving
- When looking for performance, $C^{++}$is a must
  - and some latest features are key
- python 3 is now the de factor standard
  - convert your code is not yet done