Journal club 2018/8/10

Yang Tao

- **Data conversion between c++ types and python types for PyROOT (to Yuhan)**
- **Introduction of Regular Expressions in Python**

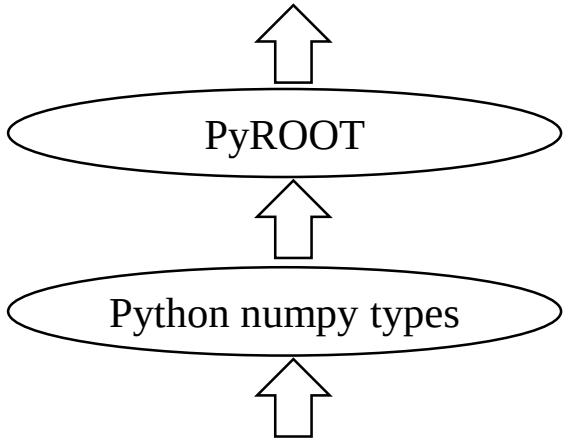
Data conversion between c++ types and python types for PyROOT

ctypes type	C type	Python type
c_bool	_Bool	bool (1)
c_char	char	1-character bytes object
c_wchar	wchar_t	1-character string
c_byte	char	int
c_ubyte	unsigned char	int
c_short	short	int
c_ushort	unsigned short	int
c_int	int	int
c_uint	unsigned int	int
c_long	long	int
c_ulong	unsigned long	int
c_longlong	__int64 or long long	int
c_ulonglong	unsigned __int64 or unsigned long long	int
c_size_t	size_t	int
c_ssize_t	ssize_t or Py_ssize_t	int
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char * (NUL terminated)	bytes object or None
c_wchar_p	wchar_t * (NUL terminated)	string or None
c_void_p	void *	int or None

Python 3

- int
- float

} Dynamic allocate depends on RAM
$(-\infty \sim +\infty)$



C++

- short / unsigned short 16 bit
- int / unsigned int 32 bit
- long int / unsigned long int 64 bit
- float 32 bit
- Double 64 bit

ROOT

PyROOT

Default python data types (int or float)is not advised for data safety if you have no enough confidence that there is no data overflow when you create a new branch.

```
Short_t  value_short
Int_t  value_int
Long64_t  value_long
Float_t  value_float
Double_t  value_double
```

Corresponding types

```
value_short = numpy.zeros(1, dtype='int16')
value_int = numpy.zeros(1, dtype='int32')
value_longint = numpy.zeros(1, dtype='int64')
value_float = numpy.zeros(1, dtype='float32')
value_double = numpy.zeros(1, dtype='float64')
value_vector = ROOT.std.vector(int)()
```

```
TBranch *Short_Branch = tree->Branch( "short", &value_short, "short/S")
TBranch *Int_Branch = tree->Branch("int", &value_int, "int/I")
TBranch *Long_Branch = tree->Branch("long", &value_long, "long/L")
TBranch *Float_Branch = tree->Branch("float", &value_float, "float/F")
TBranch *Double_Branch = tree->Branch("double", &value_double, "double/D")
```

```
tree.Branch('short', value_short, 'short/S')
tree.Branch('int', value_int, 'int/I')
tree.Branch('long', value_long, 'long/L')
tree.Branch('float', value_float, 'float/F')
tree.Branch('double', value_double, 'double/D')
tree.Branch('vector', value_vector)
```

Address : &

Substitute for C++ pointer in Python : empty list / array

Python container

Address : numpy arrary / list [ ]

Pattern Matching with Regular Expressions in Python

Regular expressions, called regexes for short, are descriptions for a pattern of text. For example, a \d in a regex stands for a digit character — that is, any single numeral 0 to 9. The regex \d\d\d-\d\d\d-\d\d\d\d is used by Python to match the same text about phonenumber : a string of three numbers, a hyphen, three more numbers, another hyphen, and four numbers. Any other string would not match the \d\d\d-\d\d\d-\d\d \d\d regex.

```
>>> import re
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> print('Phone number found: ' + mo.group())
Phone number found: 415-555-4242
```

Here, we pass our desired pattern to re.compile() and store the resulting Regex object inphoneNumRegex. Then we call search() on phoneNumRegex and pass search() the string we want to search for a match. Knowing that mo contains a Match object and not the nullvalue None, we can call group() on mo to return the match. Writing mo.group() inside our print statement displays the whole match, 415-555-4242.

- **Grouping with Parentheses**

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> mo.group(1)
'415'
>>> mo.group(2)
'555-4242'
>>> mo.group(0)
'415-555-4242'
>>> mo.group()
'415-555-4242'
```

If you would like to retrieve all the groups at once, use the groups() method — note the plural form for the name.

```
>>> mo.groups()
('415', '555-4242')
>>> areaCode, mainNumber = mo.groups()
>>> print(areaCode)
415
>>> print(mainNumber)
555-4242
```

- **Matching Multiple Groups with the Pipe**

The “ | ” character is called a pipe. You can use it anywhere you want to match one of many expressions.

```
>>> heroRegex = re.compile (r'Batman|Spiderman')
>>> mo1 = heroRegex.search('Batman and Spiderman.')
>>> mo1.group()
'Batman'
>>> mo2 = heroRegex.search(Spiderman and Batman.')
>>> mo2.group()
' Spiderman '
```

You can also use the pipe to match one of several patterns as part of your regex

```
>>> batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
>>> mo = batRegex.search('Batmobile lost a wheel')
>>> mo.group()
'Batmobile'
>>> mo.group(1)
'mobile'
```

- **Optional Matching with the Question Mark “ ? ”**

Sometimes there is a pattern that you want to match only optionally. That is, the regex should find a match whether or not that bit of text is there. The ? character flags the group that precedes it as an optional part of the pattern.

```
>>> batRegex = re.compile(r'Bat(wo)?man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'
>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
```

The (wo)? part of the regular expression means that the pattern wo is an optional group. The regex will match text that has zero instances or one instance of wo in it. This is why the regex matches both 'Batwoman' and 'Batman'.

If you need to match an actual question mark character, escape it with \?.

- **Matching Zero or More with the Star “ * ”**

The * (called the star or asterisk) means “match zero or more” — the group that precedes the star can occur any number of times in the text. It can be completely absent or repeated over and over again.

```
>>> batRegex = re.compile(r'Bat(wo)*man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'
>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
>>> mo3 = batRegex.search('The Adventures of
Batwowowowoman')
>>> mo3.group()
'Batwowowowoman'
```

If you need to match an actual star character, prefix the star in the regular expression with a backslash, *.

- **Matching One or More with the Plus “ + ”**

While * means “match zero or more,” the + (or plus) means “match one or more.” Unlike the star, which does not require its group to appear in the matched string, the group preceding a plus must appear at least once. It is not optional.

```
>>> batRegex = re.compile(r'Bat(wo)+man')
>>> mo1 = batRegex.search('The Adventures of Batwoman')
>>> mo1.group()
'Batwoman'
>>> mo2 = batRegex.search('The Adventures of
Batwowowowoman')
>>> mo2.group()
'Batwowowowoman'
>>> mo3 = batRegex.search('The Adventures of Batman')
>>> mo3 == None
True
```

If you need to match an actual plus sign character, prefix the plus sign with a backslash to escape it: \+.

- **Matching Specific Repetitions with Curly Brackets “ { } ”**

If you have a group that you want to repeat a specific number of times, follow the group in your regex with a number in curly brackets.

```
>>> haRegex = re.compile(r'(Ha){3}')
>>> mo1 = haRegex.search('HaHaHa')
>>> mo1.group()
'HaHaHa'
>>> mo2 = haRegex.search('Ha')
>>> mo2 == None
True
```

- **Greedy and Nongreedy Matching**

Python’s regular expressions are greedy by default, which means that in ambiguous situations they will match the longest string possible. The non-greedy version of the curly brackets, which matches the shortest string possible, has the closing curly bracket followed by a question mark.

```
>>> greedyHaRegex = re.compile(r'(Ha){3,5}')
>>> mo1 = greedyHaRegex.search('HaHaHaHaHa')
>>> mo1.group()
'HaHaHaHaHa'
>>> nongreedyHaRegex = re.compile(r'(Ha){3,5}?')
>>> mo2 = nongreedyHaRegex.search('HaHaHaHaHa')
>>> mo2.group()
'HaHaHa'
```

- **Appendix : Character Classes**

Shorthand character class	Represents
`\d`	Any numeric digit from 0 to 9.
`\D`	Any character that is *not* a numeric digit from 0 to 9.
`\w`	Any letter, numeric digit, or the underscore character. (Think of this as matching “word” characters.)
`\W`	Any character that is *not* a letter, numeric digit, or the underscore character.
`\s`	Any space, tab, or newline character. (Think of this as matching “space” characters.)
`\S`	Any character that is *not* a space, tab, or newline.