In the link above, you find a more comprehensive GF⇔Python tutorial, with links to Jupyter notebooks and GF source code.
record ≈ class

table ≈ dictionary

param ≈ enum

static types

Python
from enum import Enum

# class in Python
class Record:
    # Named fields
    one = None
    two = None
    three = None
    four = None

    # Constructor that fills the fields
def __init__(self, a, b, c, d):
        self.one = a
        self.two = b
        self.three = c
        self.four = d
from enum import Enum

# class in Python

class Record:
    # Named fields
    one = None
    two = None
    three = None
    four = None

    # Constructor that fills the fields
    def __init__(self, a, b, c, d):
        self.one = a
        self.two = b
        self.three = c
        self.four = d
from enum import Enum

# class in Python

class Record:
    # Named fields
    one = None
    two = None
    three = None
    four = None

    # Constructor that fills the fields
def __init__(self, a, b, c, d):
        self.one = a
        self.two = b
        self.three = c
        self.four = d
```python
from enum import Enum

class Record:
    # Named fields
    one  = None
    two  = None
    three = None
    four = None

    # Constructor that fills the fields
    def __init__(self,a,b,c,d):
        self.one  = a
        self.two  = b
        self.three = c
        self.four = d

# Define a variable of type Record
myrecord = Record(1,2,3,4)
```
# Enumeration class

class ParamInt(Enum):
    One    = 1
    Two    = 2
    Three  = 3
    Four   = 4

# Dictionary that uses ParamInt as key
mytable = {
    ParamInt.One    : 1,
    ParamInt.Two    : 2,
    ParamInt.Three  : 3,
    ParamInt.Four   : 4
}
# Define a variable of type Record
myrecord = Record(1,2,3,4)

# Dictionary that uses ParamInt as key
mytable = {ParamInt.One : 1,
            ParamInt.Two  : 2,
            ParamInt.Three: 3,
            ParamInt.Four : 4}

gokubi:tmp inari$ python3
>>> from Comparison import *
>>> myrecord.one
1
>>> mytable[ParamInt.One]
1
Types

```python
>>> type(myrecord)
<class 'Comparison.Record'>
>>> type(myrecord.one)
<class 'int'>
>>> type(Record.one)
<class 'NoneType'>
```
Types

```python
>>> type(myrecord)
<class 'Comparison.Record'>
>>> type(myrecord.one)
<class 'int'>
>>> type(Record.one)
<class 'NoneType'>
```
resource Comparison = open Prelude in {
  oper
  
  -- first we declare types
  myrecord : { one : Predef.Int ;
              two : Predef.Int ;
              three : Predef.Int ;
              four : Predef.Int } ;

  mytable : ParamInt => Predef.Int ;

  param
  
  -- param for the left-hand side of table
  ParamInt = One | Two | Three | Four ;
resource Comparison = open Prelude in 

oper

    -- first we declare types
    myrecord : { one : Predef.Int ;
                 two : Predef.Int ;
                 three : Predef.Int ;
                 four : Predef.Int } ;

    mytable : ParamInt => Predef.Int ;

param

    -- param for the left-hand side of table
    ParamInt = One | Two | Three | Four ;
-- then we define values
myrecord = { one = 1;
two = 2;
three = 3;
four = 4; }

mytable = table { One => 1;
Two => 2;
Three => 3;
Four => 4 };
This is GF version 3.10.
Built on darwin/x86_64 with ghc-8.2, flags: interrupt server
License: see help -license.
mytable = table { One => 1 ; Two => 2 ; Three => 3 ; Four => 4 };
myrecord = { one = 1 ; two = 2 ; three = 3 ; four = 4 };

Languages:
> i -retain Comparison.gf
3 msec
> cc myrecord.one
1
0 msec
> cc mytable ! One
1
0 msec
>
Common pitfalls
Compile-time tokens vs. runtime strings

```kotlin
abstract UnsupportedTokenGluing = {
    flags startcat = S ;
cat
    S ; A ;
fun
toS : A → S ;
a : A ;
}
```

https://gist.github.com/inariksit/edde72f43d439571c79f8ef758443c11
Compile-time tokens vs. runtime strings

```plaintext
concrete UnsupportedTokenGluingCnc of UnsupportedTokenGluing = {
    lin
    S, A = Str ;
    lin
    toS = addA ;  -- Unsupported token gluing:
    a = "" ;
    oper
    addA : Str -> Str = \s -> s + "a" ;
}
```
Compile-time tokens vs. runtime strings

```haskell
concrete UnsupportedTokenGluingCnc of UnsupportedTokenGluing = {

lincat
    S, A = Str ;
lin
    toS x = x ;
    a = addA "" ; -- No error

oper
    addA : Str -> Str = \s -> s + "a" ;
```
Now for the dreaded compile-time string token rule: **GF requires that every token -- every separate word -- be known at compile-time.** Rearranging known tokens in new ways, no problem: GF can generate an infinite variety of different combinations of words.

But they have to be words known to GF at compile-time. GF is not improv: as Shakespeare might have said, if anybody's going to make up new words around here, it'll be the playwright, not the actor. **You can + tokens together but only at compile-time. If you try to do it at run-time, you will get weird errors, like unsupported token gluing or, worse, Internal error in GeneratePMCFG.**

This is very different to what Python does: Python quite happily manipulates strings at any time, because to Python, strings are just arrays of characters. Space is just another character. **But to GF, words carry meaning; and run-time is too late to make up new words and new meanings.**

https://daherb.github.io/GF-for-Python-programmers/Tutorial.html
Using GF grammars from Python
Live demo using grammaticalframework.org/doc/runtime-api.html#python