The NOOP experimental Python programming environment

Anders Andersen
Department of Computer Science
Faculty of Science and Technology
UiT The Arctic University of Norway
9037 Tromsø, Norway
Anders.Andersen@uit.no

March 31, 2014
(Revision 1.1)

Abstract

Python is a dynamic language well suited to build a runtime providing adaptive support to distributed applications. Python has dynamic typing where variables are given a type when they are assigned a value. To introduce type safety, interfaces, and a component model in Python NOOP introduces a type language and a way to apply typing to functions (and methods). This type system is described in the first part of this paper. The second part use this type system to create interfaces and a software component model. And finally it is discussed how NOOP can provide adaptive support to distributed applications.

1 Introduction

Python is a dynamic interpreted language with implicit typing. A variable is defined and gets a type when a value is assigned to it. This is also true for function arguments and return values. When a new function is defined no explicit type information is provided. Argument values are assigned values at call time based on their position or name. It is possible for arguments to have a default value. (also called an optional argument since no new value has to be assigned to it at call time). It is also possible to combine positional and named arguments when a function call is performed. A typical usage of this is to have one or two obligatory positional arguments followed by a set of named optional arguments.

The `withdraw` function in Figure 1 has two obligatory positional arguments `account` and `amount` and two optional named arguments `on_behalf_of` and `message`. At call time in this example three of these arguments are provided values, and therefore implicit given a type. The `account` argument is assigned the an `int` value, the `amount` argument is assigned a `float` value, and the named `message` argument is assigned a `str` value. The two optional arguments were at define time given a default value and therefore an implicit type. However, in Python any argument (and any variable) can be assigned a value of different type everytime it is used (sometimes this is intentional).

In large software projects well-defined function behavior is important. Part of this is well-defined arguments and return values. Introduction of types and a type system is a common approach to support this. If this is introduced for Python functions the actual implementation of these functions can be made less complex and less error prone. The reason is that the programmer can expect that the arguments are of the correct type. In Python it is common to create robust code by testing that the applied arguments to a function have the correct type. With type specification and type checking this is not necessary. In a distributed setting this can be extended to avoid that a remote method invocation is performed if the correct type of arguments are not provided. Raising such an error locally at the callee is more efficient than doing it remotely. At least two messages sent over the network connection are avoided in case of type error.

The type of arguments and return values of a function is the signature of the function. If functions are class methods we can call the set of signatures provide by the class instances for the interface. If all interaction of a class instance (or an object) is through well-defined interfaces this is close to what commonly is called a software component [1]. Software components were originally proposed by Douglas McIlroy in 1968 [2], but the modern concept of software components is close to Software ICs [3].

Python does not have type safe functions, but Python provides the necessary mechanisms to implement it. In the NOOP project a type system for Python functions that makes it possible to define the signature of such functions has been implemented. We have chosen a hybrid approach to the NOOP type system [4] where it is possible to combine statical typing of NOOP with the dynamic typing of Python. Signatures can be used to create interfaces. Interfaces applied to well-defined Python classes are the core of NOOP software components. Such components can be deployed in a NOOP run-time both as single component or as a composition of components. At

```
def withdraw(account, amount, on_behalf_of="", message=""):
    # The actual implementation is ignored in this example
    return amount
new_balance = withdraw(13219254, 125.25, message="School trip")
```

Figure 1: Python function combining positional and named arguments.

deploy time a contract between the component and the run-time is provided. This contract includes the requirements of the component that has to be fulfilled by the run-time. How the contract is fulfilled also depends on the given context of the deployed component.

In this paper will present the type system of NOOP, how this is used to define the signature of Python functions, and how such signatures are used to define interfaces. NOOP components and the deployment of such components will be introduced. Finally, its is discussed how NOOP can provide adaptive support to distributed applications.

2 Types and signatures

Python provides a set of built-in types. The principal built-in types are numerics (**int**, **float**), sequences (**list**, **tuple**, **str**), mappings (**dict**), files, classes, instances and exceptions. Every Python object is the instance of a class, and the built-in **isinstance(obj,cls)** function can be used to check if a Python object **obj** is an instance of the class **cls** (or of a direct or indirect subclass of **cls**). For any object or value the **type(obj)** function returns the type or class. For example, **type(1)** is **int**.

In NOOP, the type system has been extended with composite types. A few examples are given in Figure 2. The first example gives us the possibility to define a tuple with a well-defined number of elements with well-defined types (a tuple with three elements of the type **int**, **str**, and **float**). The second example gives us the possibility to define a list of integers (lists in Python can have any combination of value types). The third example gives us the possibility to define a dictionary of any length where the keys are of type **str** and the values are of type **int**. And the last example provides a dictionary with two elements where the first key is **"id"** and the second key is **"sh"**, and the value of **"id"** is of type **int** and the value of **"sh"** is of type **str**.

A few new type constructors have been added to NOOP. The reason is that such constructors can be used to give a more precise definition of the programmer's intention. With this richer language a more detailed signature including the relation between different arguments and return values can be provided. Table 1 lists the new type constructors. The extended type system is available in the **signature** [1] module.

All the type constructors are used to create new types. The **whatever** type is true for any values. The **opt** type says that the value should either be of this type or not present at all. The **one** type says that the value should be of one of the listed types. The type constructor **pred** has an argument **p** that is a predicate. This predicate is a function that accepts one argument and returns either **True** or **False**. The argument is the value of the applied argument to the type. The **tgtz** type below specifies all integers larger than zero:

whatever	Value of any type
opt(t)	Value of type **t** or no value
one(t$_1$,t$_2$,...)	Value of either type **t**$_1$, **t**$_2$, ...
pred(t,p)	Value of the type **t** and **p** is true
arg(i)	The type of argument **i**
isarg(i,{...})	Mapping from type of **i**

Table 1: New type constructors in NOOP.

```
def gtz():  return v > 0
tgtz = pred(int, gtz)
```

The predicate type constructor is used to limit the accepted values of a given type. It should not be confused with the concept of dependent types [5, 6] that can create more expressive type constructors. Currently NOOP does not provide such type constructors. The type constructors **arg** and **isarg** will be discussed later.

The type system in NOOP is extensible. It is easy to create new types using the type constructors discussed above. It is also possible to create completely new types constructors using the **typespec** class. Create a new class that inherits the **typespec** class and implement the actual type check for the new type in the **__call__** method. If the new type constructor is parameterized the **__init__** method has to be implemented too. The **whatever** type is not parameterized, but the other type constructors listed in Table 1 are. A new parameterized type constructor for positive integers up to a given value is implemented in Figure 3. The **__init__** method is called when a new type is created using the type constructor (line 10). The **__call__** method should have exactly one argument. This is the value that is type checked against the type when NOOP performs type checking. The **__call__** method should raise a **SignatureError** if the value does not match the type.

In NOOP, two approaches are used to add signatures to functions. The first approach use Python decorators (available for functions since Python 2.4). Decorators can be applied to Python functions by a line starting with **@** before the function definition. Following the **@** is the name of the decorator and optionally a set of arguments. A Python decorator is implemented as a function. In NOOP a signature decorator can be used to add sig-

[1] In the following examples the first code line is not shown. Its only purpose is to load the **signature** module, and it is equivalent to "**from noop.core.signature import ***" in all the examples.

```
type((1,"foo",2.3))          is (int,str,float)
type([1,4,7,8])              is [int]
type({"ID":212,"GID":100})   is {str:int}
type({"id":42,"sh":"bash"}) is {"id":int,"sh":str}
```

Figure 2: Composite types in NOOP.

```
class maxint(typespec):
    def __init__(self, max):
        self.max = max
    def __call__(self, value=missing):
        if ((not type(value) is int) or
            (value < 0) or
            (value > self.max)):
            raise SignatureError("No match")
```

Figure 3: A new type constructor **maxint**.

natures to functions. The **@signature** decorator takes three arguments.

The first argument is the type specification of the decorated function's arguments. It is either a tuple or a dictionary. Each element of the tuple or the dictionary represents an argument to the function. If it is a dictionary the type specification is given using the names of the arguments. The arguments of the **withdraw** function above could be specified like this (the first line as a tuple and the following lines as a dictionary):

```
(int, float, opt(str), opt(str))
{"account": int, "amount": float,
 "on_behalf_of": opt(str),
 "message": opt(str)}
```

The second argument of the **@signature** decorator is the type specification of the decorated function's return value. This is just the return value type. The return value type of the **withdraw** function above is **float**. The third argument is a list of exceptions the decorated function might raise during its execution. If the **withdraw** function above raised an **IndexError** when an unknown account number was applied the exception list could be specified with **[IndexError]**. The complete signature of the **withdraw** function using the **@signature** decorator is shown in Figure 4.

It is also possible to specify the **@signature** decorator with named arguments. The arguments type specification in named **args**, the return value type specification is named **ret**, and the list of exceptions is named **exc**. This is a signature with named arguments for the **gtz** function:

```
@signature(args=(int,), ret=bool,
           exc=[TypeError])
def gtz(v):
    return v > 0
```

The second approach to add signatures to Python functions in NOOP is to use annotations. Annotations has been available since Python 3.0. In NOOP we use annotations to annotate arguments and return values of functions with types. When a function is defined each argument can be annotated using a colon. If a function has an argument **s** of type **str**, the argument can be annotated like this: **s:str**. To specify the type of the return value of a function the function is annotated using **->**. To apply the possible list of exceptions a function can raise we still have to use the **@signature** decorator.

At define time the function is analyzed to see if it matches the type specification. At call time type checking ensures that no arguments not matching the type specification is forwarded to the function. Type checking also ensures that the return value matches the type specification and that no exception not defined in the signature is raised. If either of these fails a **SignatureError** exception is raised.

It is possible to completely ignore exceptions in type checking at call time. The consequence is that *any* exceptions raised by the function will be thrown back to the caller. To achieve this effect the exception paramater (**exc**) of the **@signature** decorator is set to None This can also be achieved by providing no value for this argument.

3 Interfaces and receptacles

The NOOP approach to interfaces differs a lot from the now rejected proposal for Python found in PEP 245 [7]. PEP 245 proposes interfaces similar to what is found in Java where a class implements a defined interface. This is also true for Zope interfaces [8]. While the NOOP approach also can be used like this, its main purpose is to support the interaction between objects. In that sense it is closer to interfaces related to software components or remote invocation.

In NOOP interfaces of objects lists methods with signatures. One object can implement several interfaces. Receptacles represent interfaces used by objects. Object implementations refer to external interfaces through receptacles and receptacles are explicit bound to interfaces (late binding). The binding operation (e.g. **bind**) can be

```
@signature((int,float,opt(str),opt(str)), float, [IndexError])
def withdraw(account, amount, on_behalf_of="", message=""):
    # The actual implementation is ignored in this example
    return amount
```

Figure 4: Signature decorator for the **withdraw** function.

```
mSig = ((int, int), int, [])
iMath = {"add": mSig, "sub": mSig}

@interfaces(math=iMath)
class Math:
    def add(self, x:int, y:int) -> int:
        return x + y
    def sub(self, x:int, y:int) -> int:
        return x - y
```

Figure 5: A **Math** class with an interface **math**.

```
@receptacles(m=iMath)
class Wallet:
    def __init__(self):
        self.v = 0
    def doSave(self, x: int):
        self.v = m.add(self.v, x)
    def doSpend(self, x: int):
        self.v = m.sub(self.v, x)
```

Figure 6: A **Wallet** class with a receptacle **m**.

(and often is) performed outside the object implementation.

The **@interface** decorator is used to create interfaces on a Python object in NOOP. To the interface decorator named arguments are applied. The names represents the name of the interface. The value list the methods and their signatures. A **Math** class that can be used to create objects with an interface **math** of type **iMath** with two metods **add** and **sub** are defined in Figure 5 (**mSig** is the signature of both method **add** and **sub**). The signature of each method specified in the **math** interface are applied to the matching methods of the class. It is possible apply these signatures explicit to each method in the class. Type checking will then ensure that the signatures of the methods match the signatures of the interface. In the example in Figure 5 the methods are annotated with the type information.

If an object should access an interface of another object receptacles are used. A receptacle refers to an external interface implementation that is unknown at definition time. Later, this receptacle can be bound to such an interface. The **@receptacles** decorator is used to add receptacles to an object. In Figure 6 the receptacle **m** is added to all objects of the **Wallet** class. The receptacle **m** can then be used to call to methods of an interface of the type **iMath** (like the **math** interface of **Math** objects). Before **m** can be used it has to be bound to an interface of type **iMath**. The following code makes an instance of both the **Math** and **Wallet** class, connects the receptacle **m** of the wallet to the math object, and

```
mSig = ((int, int), int, [])
iMath = {"add": mSig, "sub": mSig}

@component(provides={"math": iMath})
class Math:
    def add(self, x:int, y:int) -> int:
        return x + y
    def sub(self, x:int, y:int) -> int:
        return x - y
```

Figure 7: A **Math** component providing interface **math**.

perform the **doSave** operation of the wallet object. The **doSave** operation accesses the **add** method of the math object though the receptacle **m** and the interface **math**.

```
myWallet = Wallet()
myMath = Math()
localBind(myMath["math"],myWallet["m"])
myWallet.doSave(145)
```

4 Software components

A NOOP component is a Python object with well defined external behavior defined by a set of interfaces (**provides**), a set of receptacles (**uses**), and a run-time contract. To implement a NOOP component a **@component** decorator is added to the class of the object. It is easy to rebrand the **Math** and **Wallet** class to NOOP components. The **@interfaces** and **@receptacles** decorators are replaced with **@component** decorators that include the named arguments **provides** and **uses**. The **provides** argument lists the interfaces provided by this component, and the **uses** argument lists the interfaces used by this component (the receptacles). Figure 7 and 8 show the implementation of the **Math** component and the **Wallet** component, respectively. In the **Wallet** component we have added a provided interface **wallet**.

A NOOP component is not instantiated like ordinary Python objects. A NOOP component is deployed, and the run-time contract is applied to the component at deploy time. The run-time contract includes external interfaces used by the component and life-cycle management information.

The deployment operation returns a unique reference for the component. This reference is a global unique reference that can be used to refer to this component globally in any NOOP run-time. Every NOOP run-time (in NOOP called a capsule) has to implement a deploy method. The actual implementation might vary depending of the features and services provided by the run-time. The deploy-time contract can be used to specify features

```
wSig = ((int,), None, [])
cSig = ((), int, [])
iWallet = {"doSave":wSig, "doSpend":wSig,
           "content": cSig}

@component(provides={"wallet": iWallet},
           uses={"m": iMath})
class Wallet:
    def __init__(self):
        self.v = 0
    def doSave(self, x: int):
        self.v = m.add(self.v, x)
    def doSpend(self, x: int):
        self.v = m.sub(self.v, x)
    def content(self):
        return self.v
```

Figure 8: A **Wallet** component providing interface **wallet** and using interface **m**.

and services needed by a given component (or composition of components).

The simplest contract possible is an empty contract. In NOOP it is created as an empty dictionary:

```
contract = {}
```

A more common contract of a component maps its receptacles to external interfaces using the **bind** argument. For the **Wallet** component the deploy contract could be specified like this (**mathRef** is the unique reference to a **Math** component):

```
contract={"bind":{"m":mathRef["math"]}}
```

The contract specifies that a binding between the **m** receptacle of the **Vallet** and the **math** interface of the **Math** component has to be created. To complete the example of the **Math** and **Wallet** component, this is how we deploy and use a **Math** component and a **Wallet** component using an empty contract for the **Math** component and a simple **bind** contract for the **Wallet** component:

```
mathRef=deploy(Math,{})
contract={"bind":{"m":mathRef["math"]}}
walletRef=deploy(Wallet,contract)
walletRef["wallet"].doSave(145)
```

In a NOOP run-time the component references can be used as proxies. The interfaces (and receptacles) can be accessed using their names as keys (like a Python dictionary). The methods of the interfaces can be accessed using ordinary dot-notation.

In NOOP a composite component is a composition of components. Every single component in the composition have an individual contract, and the composition of components have a common contract. All components of a composition is deployed in a single operation. The actual steps performed when a composition is deployed are these: (i) All components are instantiated. (ii) The contracts are applied to the components. (iii) The composition contract is applied to the composition.

Software components in NOOP are an unit for deployment. It is possible to see a component (and a composite component) as a unit that can be distributed independently and deployed in different applications and systems. The details of how this is achieved is out of the scope of this paper.

5 Dynamic support

Late binding and re-binding is an important part of the dynamic application support provided by NOOP. Components access other components, including system level components, through receptacles. Receptacles are bound to actual implementations at deploy time, and can be re-bound to other implementations later if this matches the given context better. Contracts specify the requirements of a component, including the services a component needs. Such contracts can include quality of service (QoS) specifications, and how a service is implemented might depend on the given context. Some services might be optional (a typical example is logging), and some contracts might specify a preferred service quality level and a minimum acceptable service quality level. The given context might also influence how the run-time fulfills the component requirements specified in the contract.

A typical NOOP application is a distributed application with a set of components deployed in a set of run-times called capsules. Each NOOP capsule an be tailored to the specific requirements of its deployed components. In NOOP the goal is not a single capsule type supporting a wide range of component requirements, but specialized capsules configured to support its deployed components (similar to the extensible application server discussed in [9]). A composite component might be distributed over several capsules. A typical example of such a distributed composite component is a remote binding that contains a stub and a skeleton deployed in different capsules.

When a component is deployed in a capsule the contract might specify complex requirements that includes adaption rules triggered by observed context changes. The details of such adaption is out of the scope of this paper. However, the NOOP component model, interfaces, receptacles and contracts are important mechanisms necessary to provide the adaptive run-time of NOOP.

6 Conclusion

The component model and the NOOP run-time is the base of several research projects investigating adaptive support for distributed applications. Different versions of the run-time exists, and the run-time itself can be configured to provide specialized support for a given type of application. The NOOP core functionality presented in this paper is used to investigate such adaptive and context sensitive behaviour further.

References

[1] C. Szyperski, *Component Software, Beyond Object-Oriented Programming*, 2nd ed., ser. The Component Software Series. Addison-Wesley, 2002.

[2] M. D. McIlroy, "Mass produced software components," in *Proceedings, NATO Conference on Software Engineering*, P. Naur and B. Randell, Eds., Garmisch, Germany, Oct. 1968.

[3] L. Ledbetter and B. Cox, "A plan for building reusable software components," *Byte*, vol. 10, no. 6, pp. 307–316, 1985.

[4] J. Siek and W. Taha, "Gradual typing for objects," in *Proceedings of the 21st European conference on Object-Oriented Programming: ECOOP 2007*. Springer-Verlag, 2007, pp. 2–27.

[5] J. McKinna, "Why dependent types matter," *ACM Sigplan Notices*, vol. 41, no. 1, pp. 1–1, Jan. 2006.

[6] H. Barendregt, "Lambda calculi with types," in *Handbook of Logic in Computer Science*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds. Oxford Science Publications, 1992.

[7] M. Pelletier, *PEP 245: Python Interface Syntax*, 2001.

[8] B. Muthukadan, *A Comprehensive Guide to Zope Component Architecture*. Lulu, 2007.

[9] A. Munch-Ellingsen, D. P. Eriksen, and A. Andersen, "Argos, an extensible personal application server," in *Middleware 2007*, ser. Lecture Notes in Computer Science, vol. 4834, Nov. 2007, pp. 21–40.