Multiparadigm Programming with Python Chapter 5

H. Conrad Cunningham

05 April 2022

Contents

Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of April 2022 is a recent version of Firefox from Mozilla.

5 Python Types

5.1 Chapter Introduction

The goals of this chapter (5) are to:

- examine the general concepts of type systems
- explore Python's type system and built-in types

Note: In this book, we use the term Python to mean Python 3. The various examples use Python 3.7 or later.

5.2 Type System Concepts

The term *type* tends to be used in many different ways in programming languages. What is a type?

Chapter 7) on object-based paradigms discusses the concept of type in the context of object-oriented languages. This chapter first examines the concept more generally and then examines Python's builtin types.

5.2.1 Types and subtypes

Conceptually, a *type* is a set of values (i.e., possible states or objects) and a set of operations defined on the values in that set.

Similarly, a type `S` is (a behavioral) *subtype* of type `T` if the set of values of type `S` is a "subset" of the values in set `T` an set of operations of type `S` is a "superset" of the operations of type `T`. That is, we can safely *substitute* elements of subtype `S` for elements of type `T` because `S`'s operations behave the "same" as `T`'s operations.

This is known as the *Liskov Substitution Principle* [12,20].

Consider a type representing all furniture and a type representing all chairs. In general, we consider the set of chairs to be a subset of the set of furniture. A chair should have all the general characteristics of furniture, but it may have additional characteristics specific to chairs.

If we can perform an operation on furniture in general, we should be able to perform the same operation on a chair under the same circumstances and get the same result. Of course, there may be additional operations we can perform on chairs that are not applicable to furniture in general.

Thus the type of all chairs is a subtype of the type of all furniture according to the Liskov Substitution Principle.

5.2.2 Constants, variables, and expressions

Now consider the types of the basic program elements.

A *constant* has whatever types it is defined to have in the context in which it is used. For example, the constant symbol `1` might represent an integer, a real number, a complex number, a single bit, etc., depending upon the context.

A *variable* has whatever types its value has in a particular context and at a particular time during execution. The type may be constrained by a declaration of the variable.

An *expression* has whatever types its evaluation yields based on the types of the variables, constants, and operations from which it is constructed.

5.2.3 Static and dynamic

In a *statically typed language*, the types of a variable or expression can be determined from the program source code and checked at "compile time" (i.e., during the syntactic and semantic processing in the front-end of a language processor). Such languages may require at least some of the types of variables or expressions to be *declared* explicitly, while others may be *inferred* implicitly from the context.

Java, Scala, and Haskell are examples of statically typed languages.

In a *dynamically typed language*, the specific types of a variable or expression cannot be determined at "compile time" but can be checked at runtime.

Lisp, Python, JavaScript, and Lua are examples of dynamically typed languages.

Of course, most languages use a mixture of static and dynamic typing. For example, Java objects defined within an inheritance hierarchy must be bound dynamically to the appropriate operations at runtime. Also Java objects declared of type `Object` (the root class of all user-defined classes) often require explicit runtime checks or coercions.

5.2.4 Nominal and structural

In a language with *nominal typing*, the type of value is based on the type *name* assigned when the value is created. Two values have the same type if they have the same type name. A type `S` is a subtype of type `T` only if `S` is explicitly declared to be a subtype of `T`.

For example, Java is primarily a nominally typed language. It assigns types to an object based on the name of the class from which the object is instantiated and the superclasses extended and interfaces implemented by that class.

However, Java does not guarantee that subtypes satisfy the Liskov Substitution Principle. For example, a subclass might not implement an operation in a manner that is compatible with the superclass. (The behavior of subclass objects are this different from the behavior of superclass objects.) Ensuring that Java subclasses preserve the Substitution Principle is considered good programming practice in most circumstances.

In a language with *structural typing*, the type of a value is based on the *structure* of the value. Two values have the same type if they have the "same" structure; that is, they have the same *public* data attributes and operations and these are themselves of compatible types.

In structurally typed languages, a type `S` is a subtype of type `T` only if `S` has all the public data values and operations of type `T` and the data values and operations are themselves of compatible types. Subtype `S` may have additional data values and operations not in `T`.

Haskell is an example of a primarily structurally typed language.

5.2.5 Polymorphic operations

Polymorphism refers to the property of having "many shapes". In programming languages, we are primarily interested in how *polymorphic* function names (or operator symbols) are associated with implementations of the functions (or operations).

In general, two primary kinds of polymorphic operations exist in programming languages:

1. *Ad hoc polymorphism*, in which the same function name (or operator symbol) can denote different implementations depending upon how it is used in an expression. That is, the implementation invoked depends upon the types of function's arguments and return value.

   There are two subkinds of ad hoc polymorphism.

   a. *Overloading* refers to ad hoc polymorphism in which the language's compiler or interpreter determines the appropriate implementation to invoke using information from the context. In statically typed languages, overloaded names and symbols can usually be bound to the intended implementation at *compile time* based on the declared types of the entities. They exhibit *early binding*.

      Consider the language Java. It overloads a few operator symbols, such as using the `+` symbol for both addition of numbers and concatenation of strings. Java also overloads calls of functions defined with the same name but different signatures (patterns of parameter types and return value). Java does not support user-defined operator overloading; C++ does.

      Haskell's *type class* mechanism implements overloading polymorphism in Haskell. There are similar mechanisms in other languages such as Scala and Rust.

   b. *Subtyping* (also known as *subtype polymorphism* or *inclusion polymorphism*) refers to ad hoc polymorphism in which the appropriate implementation is determined by searching a hierarchy of types. The

function may be defined in a supertype and redefined (overridden) in subtypes. Beginning with the actual types of the data involved, the program searches up the type hierarchy to find the appropriate implementation to invoke. This usually occurs at runtime, so this exhibits *late binding*.

The object-oriented programming community often refers to inheritance-based subtype polymorphism as simply *polymorphism*. This is the polymorphism associated with the class structure in Java.

Haskell does not support subtyping. Its type classes do support *class extension*, which enables one class to inherit the properties of another. However, Haskell's classes are not types.

2. *Parametric polymorphism*, in which the same implementation can be used for many different types. In most cases, the function (or class) implementation is stated in terms of one or more type parameters. In statically typed languages, this binding can usually be done at compile time (i.e., exhibiting early binding).

   The object-oriented programming (e.g., Java) community often calls this type of polymorphism *generics* or *generic programming*.

   The functional programming (e.g., Haskell) community often calls this simply *polymorphism*.

5.2.6 Polymorphic variables

A *polymorphic variable* is a variable that can "hold" values of different types during program execution.

For example, a variable in a dynamically typed language (e.g., Python) is polymorphic. It can potentially "hold" any value. The variable takes on the type of whatever value it "holds" at a particular point during execution.

Also, a variable in a nominally and statically typed, object-oriented language (e.g., Java) is polymorphic. It can "hold" a value its declared type or of any of the subtypes of that type. The variable is declared with a static type; its value has a dynamic type.

A variable that is a parameter of a (parametrically) polymorphic function is polymorphic. It may be bound to different types on different calls of the function.

5.3 Python Type System

What about Python's type system?

5.3.1 Objects

Python is *object-based* [7, Ch. 3]; it treats all data as *objects*.

A Python object has the following *essential* characteristics of objects [7, Ch. 3]:

a. a *state* (value) drawn from a set of possible values

   The state may consist of several distinct data attributes. In this case, the set of possible values is the Cartesian product of the sets of possible values of each attribute.

b. a set of *operations* that access and/or mutate the state

c. a unique *identity* (e.g., address in memory)

A Python pbject has one of the two *important but nonessential* characteristics of objects [7, Ch. 3]. Python does:

d. *not* enforce *encapsulation* of the state within the object, instead relying upon programming conventions and name *obfuscation* to hide private information

e. exhibit an *independent lifecycle* (i.e., has a different lifetime than the code that created it)

As we see in Chapter 7, each object has a distinct dictionary, the directory, that maps the local names to the data attributes and operations.

Python typically uses dot notation to access an object's data attributes and operations:

- `obj.data` accesses the attribute `data` of `obj`
- `obj.op` accesses operation `op` of `obj`
- `obj.op()` invokes operation `op` of `obj`, passing any arguments in a comma-separated list between the parentheses

Some objects are immutable and others are mutable. The states (i.e., values) of:

- *immutable* objects (e.g., numbers, booleans, strings, and tuples) cannot be changed after creation
- *mutable* objects (e.g., lists, dictionaries, and sets) can be changed in place after creation

Caveat: We cannot modify a Python tuple's structure (i.e., length) after its creation. However, if the components of a tuple are themselves mutable objects, they can be changed in-place.

All Python objects have a type.

5.3.2 Types

In terms of the discussion in Section {#sec:type-system-concepts}, all Python objects can be considered as having one or more conceptual types at a particular

point in time. The types may change over time because the program can change the possible set of data attributes and operations associated with the object.

A Python variable is bound to an object by an assignment statement or its equivalent. Python variables are thus *dynamically typed*, as are Python expressions.

Although a Python program usually constructs an object within a particular *nominal type* hierarchy (e.g., as an instance of a class), this may not fully describe the type of the object, even initially. And the ability to dynamically add, remove, and modify attributes (both data fields and operations) means the type can change as the program executes.

The type of a Python object is determined by *what it can do*—what data it can hold and what operations it can perform on that data—rather than *how it was created*. We sometimes call this *dynamic, structural typing* approach *duck typing*. (If it walks like a duck and quacks like a duck, then it is a duck, even if is declared as a snake.)

For example, we can say that any object is of an *iterable* type if it implements an `__iter__` operation that returns a valid iterator object. An iterator object must implement a `__next__` operation that retrieves the next element of the "collection" and must raise a `StopIteration` exception if no more elements are available.

In Python, we sometimes refer to a type like iterable as a *protocol*. That is, it is an, perhaps informal, *interface* that objects are expected to satisfy in certain circumstances.

5.4 Built-in Types

Python provides several built-in types and subtypes, which are named and implemented in the core language. When displayed, these types are shown as follows:

```
<class 'int'>
```

That is, the value is an instance of a *class* named `int`. Python uses the term *class* to describe its nominal types.

We can query the nominal type of an object `obj` with the function call `type(obj)`. In the following discussion, we show the results from calling this function interactively in Python REPL (Read-Evaluate-Print Loop) sessions.

For the purpose of our discussion, the primary built-in types include:

- Singleton types
- Number types
- Sequence types
- Mapping types
- Set types
- Other types (e.g., callable, class, module, and user-defined object types)

TODO: Probably should elaborate the "other types" more than currently.

5.4.1 Singleton types

Python has single-element types used for special purposes. These include `None` and `NotImplemented`.

5.4.1.1 `None`

The name `None` denotes a value of a singleton type. That is, the type has one element written as `None`.

Python programs normally use `None` to mean there is no meaningful value of another type. For example, this is the value returned by Python procedures.

5.4.1.2 `NotImplemented`

The name `NotImplemented` also denotes a value of a singleton type. Python programs normally use this value to mean that an arithmetic or comparison operation is not implemented on the operand objects.

5.4.2 Number types

Core Python supports four types of numbers:

- integers
- real numbers
- complex numbers
- Booleans

5.4.2.1 Integers (`int`)

Type `int` denotes the set of integers. They are encoded in a variant of two's complement binary numbers in the underlyng hardware. They are of *unbounded precision*, but they are, of course, limited in size by the available virtual memory.

```
>>> type(1)
<class 'int'>
>>> type(-14)
<class 'int'>
>>> x = 2
>>> type(x)
<class 'int'>
>>> type(99999999999999999999999999999999)
<class 'int'>
```

5.4.2.2 Real numbers (`float`)

Type `float` denotes the subset of the real numbers that can be encoded as double precision floating point numbers in the underlying hardware.

```
>>> type(1.01)
<class 'float'>
>>> type(-14.3)
```

```
<class 'float'>
>>> x = 2
>>> type(x)
<class 'int'>
>>> y = 2.0
>>> type(y)
<class 'float'>
>>> x == y  # Note result of equality comparison
True
```

5.4.2.3 Complex numbers (`complex`) Type `complex` denotes a subset of the complex numbers encoded as a pair of floats, one for the real part and one for the imaginary part.

```
>>> type(complex('1+2j')) # real part 1, imaginary part 2
<class 'complex'>
>>> complex('1') == 1.0   # Note result of comparison
True
>>> complex('1') == 1     # Note result of comparison
True
```

5.4.2.4 Booleans (`bool`) Type `bool` denotes the set of Boolean values `False` and `True`. In Python, this is a subtype of `int` with `False` and `True` having the values 0 and 1, respectively.

```
>>> type(False)
<class 'bool'>
>>> type(True)
<class 'bool'>
>>> True == 1
True
```

Making `bool` a subtype of `int` is an unfortunate legacy design choice from the early days of Python. It is better not to rely on this feature in modern Python programs.

5.4.2.5 Truthy and falsy values Python programs can test any object as if it was a Boolean (e.g., within the condition of an `if` or `while` statement or as an operand of a Boolean operation).

An object is *falsy* (i.e., considered as `False`) if its class defines

- a special method `__bool__()` that, when called on the object, returns `False`
- a special method `__len__()` that returns `0`

Note: We discuss special methods Chapter 7.

Otherwise, the object is *truthy* (i.e., considered as `True`).

The singleton value `NotImplemented` is explicitly defined as truthy.

Falsy built-in values include:

- constants `False` and `None`
- numeric values of zero such as `0`, `0.0`, and `0j`
- empty sequences and collections such as `''`, `()`, `[]`, and `{}` (defined below)

Unless otherwise documented, any function expected to return a Boolean result should return `False` or `0` for false and `True` or `1` for true. However, the Boolean operations `or` and `and` should always return one of their operands.

5.4.3 Sequence types

A *sequence* denotes a serially ordered collection of zero or more objects. An object may occur more than once in a sequence.

Python supports a number of core sequence types. Some sequences have immutable structures and some have mutable.

5.4.3.1 Immutable sequences An immutable sequence is a sequence in which the structure cannot be changed after initialization.

5.4.3.1.1 `str` Type `str` (string) denotes sequences of text characters—that is, of *Unicode code points* in Python. We can express strings syntactically by putting the characters between single, double, or triple quotes. The latter supports multi-line strings.

Python does not have a separate character type; a characer is a single-element `str`.

```
>>> type('Hello world')
<class 'str'>
>>> type("Hi Earth")
<class 'str'>
>>> type('''
... Can have embedded newlines
... ''')
<class 'str'>
```

5.4.3.1.2 `tuple` Type `tuple` type denotes fixed length, heterogeneous sequences of objects. We can express tuples syntactically as sequences of comma-separated expressions in parentheses.

The tuple itself is immutable, but the objects in the sequence might themselves be mutable.

```
>>> type(())    # empty tuple
<class 'tuple'>
>>> type((1,))  # one-element tuple, note comma
<class 'tuple'>
>>> x = (1,'Ole Miss')  # mixed element types
>>> type(x)
<class 'tuple'>
>>> x[0]        # access element with index 0
1
>>> x[1]        # access element with index 1
'Ole Miss'
```

5.4.3.1.3 `range` The `range` type denotes an immutable sequence of numbers. It is commonly used to specify loop controls.

- `range(stop)` denotes the sequence of integers that starts from 0, increases by steps of 1, and stops at `stop-1`; if `stop <= 0`, the range is empty.
- `range(start, stop)` denotes the sequence of integers that starts from `start`, increases by steps of 1, and stops at `stop-1`; if `stop <= start`, the range is empty.
- `range(start, stop, step)` denotes the sequence of integers that starts from `start` with a *nonzero* stepsize of `step`.

  If `step` is positive, the sequence increases toward `stop-1`; if `stop <= start`, the range is empty.

  if negative, the sequence decreases toward `stop+1`.

  A range is a lazy data structure. It only yields a value if the output is needed.

```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(1, 5))
[1, 2, 3, 4]
>>> list(range(0, 9, 3))
[0, 3, 6]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(5, 0, -1))
[5, 4, 3, 2, 1]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
>>> list(range(0, 0))
[]
```

5.4.3.1.4 `bytes` Type `bytes` type denotes sequences of 8-bit bytes. We can express these syntactically as *ASCII character strings* prefixed by a "`b`".

```
>>> type(b'Hello\n   World!')
<class 'bytes>
```

5.4.3.2 Mutable sequences A mutable sequence is a sequence in which the structure can be changed after initialization.

5.4.3.2.1 `list` Type `list` denotes variable-length, heterogeneous sequences of objects. We can express lists syntactically as comma-separated sequence of expressions between square brackets.

```
>>> type([])
<class 'list'>
>>> type([3])
<class 'list'>
>>> x = [1,2,3] + ['four','five'] # concatenation
>>> x
[1, 2, 3, 'four', 'five']
>>> type(x)
<class 'list'>
>>> y = x[1:3] # get slice of list
>>> y
[2, 3]
>>> y[0] = 3   # assign to list index 0
[3, 3]
```

5.4.3.2.2 `bytearray` Type `bytearray` denotes mutable sequences of 8-bit bytes, that is otherwise like type `bytes`. They are constructed by calling the function `bytearray()`.

```
>>> type(bytearray(b'Hello\n   World!'))
<class 'bytes>
```

5.4.4 Mapping types

Type `dict` (dictionary) denotes mutable finite sets of key-value pairs, where the *key* is an index into the set for the *value* with which it is paired.

The key can be any *hashable* object. That is, the key can be any immutable object or an object that always gives the same hash value. However, the associated value objects may be mutable and the membership in the set may change.

We can express dictionaries syntactically in various ways such as comma-separated lists of key-value pairs with braces.

```
>>> x = { 1 : "one" }
>>> x
```

```
{1: 'one'}
>>> type(x)
<class 'dict'>
>>> x[1]
'one'
>>> x.update({ 2 : "two" }) # add to dictionary
>>> x
{1: 'one', 2: 'two'}
>>> type(x)
<class 'dict'>
>>> del x[1]  # delete element with key
>>> x
{2: 'two'}
```

5.4.5 Set Types

A *set* is an *unordered collection* of distinct *hashable* objects.

There are two built-in set types—`set` and `frozenset`.

5.4.5.1 `set`

Type `set` denotes a mutable collection.

We can create a nonempty set by putting a comma-separated list of elements between braces as well as by using the `set` constructor.

For example, sets `sx` and `sy` below have the same elements. The operation `|=` adds the elements of the right operand to the left.

```
>>> sx = { 'Dijkstra', 'Hoare', 'Knuth' }
>>> sx
{'Knuth', 'Hoare', 'Dijkstra'}
>>> sy = set(['Knuth', 'Dijkstra', 'Hoare'])
>>> sy
<  {'Knuth', 'Hoare', 'Dijkstra'}
>>> sx == sy
True
>>> sx.add('Turing')  # add element to mutable set
>>> sx
{'Turing', 'Knuth', 'Hoare', 'Dijkstra'}
```

5.4.5.2 `frozenset`

Type `frozenset` denotes an immutable collection.

We can extend the `set` example above as follows:

```
>>> fx = frozenset(['Dijkstra', 'Hoare', 'Knuth'])
>>> fx
frozenset({'Knuth', 'Hoare', 'Dijkstra'})
>>> fy = frozenset(sy)
>>> fy
```

```
frozenset({'Knuth', 'Hoare', 'Dijkstra'})
>>> fx.add('Turing')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

5.4.6 Other object types

We discuss callable objects (e.g., functions), class objects, module objects, and user-defined types (classes) in later chapters.

TODO: Perhaps be more specific about later chapters.

5.5 What Next?

TODO

5.6 Chapter Source Code

TODO, if needed.

5.7 Exercises

TODO

5.8 Acknowledgements

In Spring 2018, I wrote the general Type System Concepts section as a part of a chapter that discusses the type system of Python 3 [4] to support my use of Python in graduate CSci 658 (Software Language Engineering) course.

In Summer 2018, I revised the section to become Section 5.2 in Chapter 5 of the evolving textbook *Exploring Languages with Interpreters and Functional Programming* (ELIFP) [7]. I also moved the "Kinds of Polymorphism" discussion from the 2017 List Programming chapter to the new subsection "Polymorphic Operations". This textbook draft supported my Haskell-based offering of the core course CSci 450 (Organization of Programming Languages).

In Fall 2018, I copied the general concepts section from ELIFP and recombined it with the Python-specific content [4] to form this chapter to support my Python-based offering of the elective course CSci 556 (Multiparadigm Programming) and for a posible future book [5].

In Spring 2019, I extracted the general concepts discussion [5] to create a chapter [6] for use in my Scala-based offering of CSci 555 (Functional Programming).

The type concepts discussion draws ideas from various sources:

- my general study of a variety of programming, programming language, and software engineering over three decades [1–3,8–19].

- the *Wikipedia* articles on the Liskov Substitution Principle [20], Polymorphism [21], Ad Hoc Polymorphism [23], Parametric Polymorphism [24], Subtyping [25], and Function Overloading [22]

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a unified bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

5.9 Terms and Concepts

TODO: revise for the current content

Object, object characteristics (state, operations, identity, encapsulation, independent lifecycle), immutable vs. mutable, type, subtype, Liskov Substitution Principle, types of constants, variables, and expressions, static vs. dynamic types, declared and inferred types, nominal vs. structural types, polymorphic operations (ad hoc, overloading, subtyping, parametric/generic), early vs. late binding, compile time vs. runtime, polymorphic variables, duck typing, protocol, interface, REPL, singleton types (`None` and `NotImplemented`), number types (`int`, `float`, `complex`, `bool`, `False`, falsy, `True`, truthy), immutable sequence types (`str`, `tuple`, `range`, `bytes`), mutable sequence types (`list`, `bytearray`), mapping types (`dict`, key and value), set types (`set`, `frozenset`),other types.

5.10 References

[1] Richard Bird. 1998. *Introduction to functional programming using Haskell* (Second ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[2] Kathryn Heninger Britton, R. Alan Parker, and David L. Parnas. 1981. A procedure for designing abstract interfaces for device interface modules. In *Proceedings of the 5th international conference on software engineering*, IEEE, San Diego, California, USA, 195–204.

[3] Timothy Budd. 2000. *Understanding object-oriented programming with Java* (Updated ed.). Addison-Wesley, Boston, Massachusetts, USA.

[4] H. Conrad Cunningham. 2018. Python 3 reflexive metaprogramming. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/csci658/notes/PythonMetaprogramming/Py3RefMeta.html

[5] H. Conrad Cunningham. 2018. Multiparadigm programming with Python 3. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/csci556/Py3MPP/Ch05/05_Python_Types.html

[6] H. Conrad Cunningham. 2019. *Type system concepts*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/csci555/notes/TypeConcepts/TypeSystemConcepts.html

[7] H. Conrad Cunningham. 2022. *Exploring programming languages with interpreters and functional programming (ELIFP)*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ELIFP/ELIFP.pdf

[8] Cay S. Horstmann. 1995. *Mastering object-oriented design in C++*. Wiley, Indianapolis, Indiana, USA.

[9] Cay S. Horstmann and Gary Cornell. 1999. *Core Java 1.2: Volume I—Fundamentals*. Prentice Hall, Englewood Cliffs, New Jersey, USA.

[10] Paul Hudak. 1989. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys* 21, 3 (1989), 359–411.

[11] Roberto Ierusalimschy. 2013. *Programming in Lua* (Third ed.). Lua.org, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil.

[12] Barbara Liskov. 1987. Keynote address—Data abstraction and hierarchy. In *Proceedings on object-oriented programming systems, languages, and applications (OOPSLA '87): addendum*, ACM, Orlando, Florida, USA, 17–34.

[13] Bertrand Meyer. 1997. *Object-oriented program construction* (Second ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[14] Martin Odersky, Lex Spoon, and Bill Venners. 2008. *Programming in Scala* (First ed.). Artima, Inc., Walnut Creek, California, USA.

[15] David L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15, 12 (December 1972), 1053–1058.

[16] David L. Parnas. 1976. On the design and development of program families. *IEEE Transactions on Software Engineering* SE-2, 1 (1976), 1–9.

[17] Michael L. Scott. 2015. *Programming language pragmatics* (Third ed.). Morgan Kaufmann, Waltham, Massachusetts, USA.

[18] Robert W. Sebesta. 1993. *Concepts of programming languages* (Second ed.). Benjamin/Cummings, Boston, Massachusetts, USA.

[19] Simon Thompson. 1996. *Haskell: The craft of programming* (First ed.). Addison-Wesley, Boston, Massachusetts, USA.

[20] Wikpedia: The Free Encyclopedia. 2022. Liskov substitution principle. Retrieved from https://en.wikipedia.org/wiki/Liskov_substitution_principle

[21] Wikpedia: The Free Encyclopedia. 2022. Polymorphism (computer science). Retrieved from https://en.wikipedia.org/wiki/Polymorphism_(computer_science)

[22] Wikpedia: The Free Encyclopedia. 2022. Function overloading. Retrieved from https://en.wikipedia.org/wiki/Function_overloading

[23] Wikpedia: The Free Encyclopedia. 2022. Ad hoc polymorphism. Retrieved from https://en.wikipedia.org/wiki/Ad_hoc_polymorphism

[24] Wikpedia: The Free Encyclopedia. 2022. Parametric polymophism. Retrieved from https://en.wikipedia.org/wiki/Parametric_polymorphism

[25] Wikpedia: The Free Encyclopedia. 2022. Subtyping. Retrieved from https://en.wikipedia.org/wiki/Subtyping