Pure Functional Programming in Python

Rajshekhar Sunderraman
Department of Computer Science
Georgia State University
Atlanta, Georgia 30067, USA
raj@cs.gsu.edu

2026 IEEE Engineering Education World Conference (EDUNINE) | 979-8-3315-5097-4/26/$31.00 ©2026 IEEE | DOI: 10.1109/EDUNINE62390.2026.11547089

***Abstract*—The functional programming paradigm has a long and storied history, with its beginnings in the Lambda Calculus. In recent decades, it has been shown that robust software can be highly effectively produced by pure functional languages such as Haskell, due to its statically typed nature and strong type inference mechanisms built into the language. Desirable functional features are included in almost all programming languages these days. In this paper, a subset of Python is introduced, using which pure functional ideas such as immutability, pure functions with no side effects, and stateless programming can be learned by students. Bugs can be reduced, performance can be improved, and code can be made more maintainable and scalable by this approach. It is strongly felt that familiarity with pure functional programming is needed by students in computing, and it is argued that this can be taught within introductory programming courses that are taught in Python.**

I. Introduction

The functional programming paradigm has its origins in Lambda Calculus [1]. It was first implemented in a modern programming language, LISP [2] in 1960 and during the next several decades, many functional languages were introduced either as academic learning environments [3] or as industrial strength implementations [4], [5], [6].

Functional programming offers numerous advantages in software development [7]. It encourages the use of pure functions (without side effects), immutability, and declarative constructs, resulting in robust, concise, and easy to understand code. At the core of functional programming is the idea of composition and decomposition. Smaller and independent functions performing a single task are constructed and composed to form the solution of a bigger problem. Re-usability of functions is a natural result of this process. Pure functions also enable testing in isolation and debugging is streamlined. With clever optimization techniques, compilers for functional programming can achieve high degree of parallelism due to the fact that there is no shared mutable state. Immutability, a core principle of functional programming, eliminates the risk of race conditions and other concurrency issues that arise from shared mutable state. This makes it easier to write thread-safe and parallel code. Due to the fact that there is no mutable state, functional programming also reduces an entire class of common programming errors, leading to more robust and reliable software systems. With the availability of higher-order functions, a strong type system, and type inference, functional programming enables more expressive and high-level code. Some functional programming languages support lazy evaluation, a feature that allows for optimization and infinite data structures.

The incorporation of many of the features of functional programming into languages in other paradigms, such as Java, Python, Javascript, and Scala, has been resulted from the advantages offered by the functional programming paradigm. Since one or more of the non-functional languages are employed in their introductory classes by most of the colleges and universities, it is considered important to teach these features as early as possible. To this end, an approach to incorporating functional programming in CS1 and CS2 courses is presented.

Rest of the paper is structured as follows: Section II introduces features of Python that lend themselves to pure functional programming. Section III lays out restrictions on what features students can use to complete certain programming assignments and a simple example of this programming style is presented. Section IV presents a more challenging programming assignment for a later part of CS1 or an early part of CS2. Section V presents the notion of programming with expressions and shows how many problems, including recursion, can be solved using just expressions. Section VI presents related work that emphasize the importance and advantages of functional programming. Section VII discusses the relevance of pure functional programming in the context of AI and Engineering. Some concluding thoughts are presented in Section VIII.

II. Pure Functional Features in Python

In this section, the features of Python that are compatible with the functional programming paradigm are introduced.

A. Conditional Expressions

A conditional expression in Python has the following syntax:

```
expr1 IF cond ELSE expr2
```

where `expr1` and `expr2` are Python expressions (which themselves could be conditional expressions!) and `cond` is a Boolean expression. The value of the conditional expression is the value of `expr1` if `cond` evaluates to `True` and the value of `expr2` otherwise. An example conditional expression is:

```
5 if x%2 == 0 else 6
```

which evaluates to `5` if `x` is an even number and `6` otherwise.

B. Comprehension Expressions

Python provides three types of comprehension expressions: *list comprehensions*, *set comprehensions*, and *dictionary comprehensions*, which allow the programmer to construct new lists, sets, or dictionaries respectively out of existing values.

List comprehensions: Python allows you to create new lists out of existing values using the following syntax:

```
[ expr for x in xs if cond ]
```

This returns a list of values for `expr`, which will usually involve the variable `x`, where `x` is a value from a collection `xs` with `cond` evaluating to `True`. Here are a few examples of list comprehension expressions:

```
>>> [x**2 for x in [1,2,3,4,5]]
[1, 4, 9, 16, 25]
>>> [x+y for x in [1,2,3]
         for y in range(x) if x+y>3]
[4, 5]
```

Set comprehensions: Python allows you to create new sets out of existing values using the following syntax:

```
{ expr for x in xs if cond }
```

This returns a set of values for `expr`, which will usually involve the variable `x`, where `x` is a value from a collection `xs` with `cond` evaluating to `True`. Here are a few examples of list comprehension expressions:

```
>>> {x+y for x in [1,2,3] for y in [1,2]}
{2, 3, 3, 4, 4, 5}
>>> {x+y for (x,y) in zip([1,2],[5,4])}
{6}
```

Dictionary comprehensions: Python allows you to create new dictionaries out of existing values using the following syntax:

```
{ keyexpr:valexpr for x in xs if cond }
```

This returns a dictionary of key-value pairs made up of values for `keyexpr` and `valexpr`, which will usually involve the variable `x`, where `x` is a value from a collection `xs` with `cond` evaluating to `True`. Here are a few examples of dictionary comprehension expressions:

```
>>> {x:"a"*(x**2) for x in [1,2,3]}
{1: 'a', 2: 'aaaa', 3: 'aaaaaaaaa'}
>>> {n:p for (n,p) in zip(['Jon','Ali'],
                          ['111','222'])}
{'Jon': '111', 'Ali': '222'}
```

C. Pure Functions and Lambda Expressions

A **pure function** is one that that transforms its inputs to an output in exactly the same way every time it is called with a particular input without reading from or writing to an external value. These functions do not cause side effects and their sole purpose is to transform inputs to an output. An example of an impure function is shown below.

```
z = 12
def f(x):
  y = x + z
  z = z + 1
  return y
```

The function `f` not only reads value of `z` but also writes to it before returning its output. We shall see examples of pure functions in later sections.

A **lambda expression** is a function without a name and has the following syntax:

```
lambda plist: expr
```

Here `plist` is a comma separated list of input parameter variables and `expr` is the return value of the anonymous function being defined. An example lambda expression is shown below:

```
>>> f = lambda x,y: math.sqrt(x**2+y**2)
>>> f(3,4)
5.0
```

The function takes as input two parameters, `x` and `y` and returns the square root of the sum of the squares of the two inputs. The lambda expression has been assigned to the variable `f`. A call to the function `f` with 3 and 4 as arguments is shown in the above Python interaction.

D. Map, Filter, Reduce

Python provides two higher-order built-in functions `map` and `filter` and an external library, `functools`, provides a higher-order function `reduce`, all of which are useful in the functional programming paradigm. A higher-order function is one which either takes as one of its inputs a function object or that returns a function object. The input or output function object may be named or anonymous.

map: The `map` function takes two arguments, a single input function and a list of items whose item type matches the input type of the single input function and returns a list of items produced by applying the single input function to each item of the input list and constructing a list of the output value of the single input function. An example is shown below:

```
>>> list(map(lambda x: 2*x, [1,2,3,4,5]))
[2, 4, 6, 8, 10]
```

Note that the output of the `map` function needs to be converted to a list to see the value as the output of `map` is a `map`-object.

filter: The `filter` function takes two arguments, a single input Boolean function and a list of items whose item type matches the input type of the single input function and returns a list of items from the input list that when provided as input to the single input function returns `True`. An example is shown below:

```
>>> list(filter(lambda x: x%2==0,
                [1,2,3,4,5,6,7))
[2, 4, 6]
```

Note that the output of the `filter` function needs to be converted to a list to see the value since the output of `filter` is a `filter`-object.

reduce: the `reduce` function in Python is available in the `functools` library captures the "accumulator" pattern of code in which a collection is processed by accumulating the results of a computation on each element of the computation. The `reduce` function takes as input a two input function (an accumulator variable and an element variable from the collection) which performs the computation on the item and accumulates the result in the accumulator, a collection, and an initial value for the accumulator. The output of the `reduce` function is the final accumulator value. An example of summing the elements of a numeric list is shown below:

```
>>> from functools import reduce
>>> reduce(lambda acc,x: acc+x,[1,2,3],0)
6
```

E. Other Immutable Objects and Built-In Functions

Immutable data is an important aspect of pure functional programming. Here, some of Python's immutable data structures and methods useful to incorporate in the functional programming paradigm are presented.

Strings: Python string data is immutable. One cannot modify a string object in Python but can extract various derived strings from other string values using various operations such as the slice operator or the concatenation operator.

Tuples: Python tuple data is also immutable and once a tuple value is created it cannot be modified. Values can be extracted from the tuple to form other data values.

Collection methods: There are numerous methods on collection objects such as lists, dictionaries, and sets that return new values without modifying the collection. These methods can be used to preserve purity of functional programming in Python. A few of these methods are shown below:

```
>>> xs = [1,5,3,2]
>>> ys = sorted(xs)
>>> print(xs,ys)
[1, 5, 3, 2] [1, 2, 3, 5]
>>> d = {"a":20, "b":12, "c": 98}
>>> list(d.keys())
['a', 'b', 'c']
>>> list(d.items())
[('a', 20), ('b', 12), ('c', 98)]
>>> list(zip([1,2,3],['a','b','c']))
[(1, 'a'), (2, 'b'), (3, 'c')]
>>> ",".join(["a","b"])
'a,b'
```

III. Restrictive Programming in Python

To encourage students in introductory programming classes using Python to learn principles of the functional programming paradigm, certain restrictions on Python features to use to complete a programming assignment are enforced. The students may use conditional expressions, list/set/dictionary comprehensions, lambdas, map, filter, reduce, and other useful functions/methods that do not mutate their inputs such as `zip()`, `list()`, `sorted()`, and `join()`. The student may not use any kind of loop-, if-, if-else-, if-elif-else-statements. Typically, the solution should contain sequence of assignment statements, where the variables on the left-hand side of the assignment is used only to "name" an expression. Decomposing the solution into functions is fine as long as the code within the function also follows the restrictions. Many of the local looping code can be accomplished using the collection comprehensions. Others can be accomplished by recursive functions.

Example: Consider the problem of finding "twin" primes less than a given number, where a twin prime is a pair of primes that differ by 2. Below, a Python program that follows the restrictions of pure functional programming is presented. This solution uses the classic "sieve" algorithm to generate all primes and then extracts consecutive primes that are twin primes.

```
def removeMultiples(x,xs):
  return [] if xs == [] else
    removeMultiples(x,xs[1:])
    if xs[0]%x == 0 else
    [xs[0]] + removeMultiples(x,xs[1:])

def sieve(xs):
    return [] if xs == [] else
        [xs[0]] +
         sieve(
           removeMultiples(xs[0],xs[1:]))

def filterTwins(pairs):
    return [pair for pair in pairs
                 if pair[0]+2 == pair[1]]

def twinPrimes(n):
    ps = sieve([x for x in range(2,n+1)])
    return
      filterTwins(list(zip(ps,ps[1:])))
```

IV. A Challenging Programming Assignment

In this section, a challenging programming assignment given in a CS2 class is presented. Students were taught the functional programming constructs discussed in this paper and were asked to produce a solution in that style for the following programming assignment.

Programming Assignment: Write a Python program to compute a Relational Algebra expression corresponding to an atomic formula in Datalog. Here are two sample atomic formulas from Datalog:

```
q(x,x,x,x,x)
p(x,20,x,y,"john",x,y)
```

As can be seen, an atomic formula in Datalog begins with a NAME token (denoting the relation name) followed by a left parenthesis (LPAREN), followed by a list of arguments

(we will have at least one argument) and ends with a right parenthesis (RPAREN). The list of arguments is COMMA separated and each of argument can be either a NUMBER constant (assume positive integers) or a STRING constant, or a NAME corresponding to a variable.

The conversion from Datalog atom to Relational Algebra string works as follows (described in an inside-out manner, with `p(x,20,x,y,"john",x,y)` as an example):

Step 1: Produce a list of variables, `x0, ..., xn-1`, where n is the number of arguments in the Datalog atom and generate the following string:

```
rename[x0,x1,x2,x3,x4,x5,x6](p)
```

Step 2: Generate the select conditions; There are two types of select conditions: one that correspond to the numeric or string constants in the Datalog atom and the other that correspond to repeated variable arguments in the Datalog atom. Combine all of the conditions with `and` and generate the following string:

```
select[x1=20 and x4="john" and
       x0=x2 and x2=x5 and x3=x6]
  (rename[x0,x1,x2,x3,x4,x5,x6](p))
```

As can be seen, there are two conditions that correspond to the constant arguments of the atom, two conditions to ensure that the 3 instances of variable `x` are equated and one condition to ensure that the 2 instances of the variable `y` are equated. Notice that this string builds on the string generated in the previous step.

Step 3: Generate the project list; this list consists of the unique variables in the Datalog atom (in this case the unique variables are x and y); Generate the following string, again building on the previous string:

```
project[x0,x3](
  select[x1=20 and x4="john" and
         x0=x2 and x2=x5 and x3=x6]
    (rename[x0,x1,x2,x3,x4,x5,x6](p)))
```

Step 4: Produce the rename list of unique variables from the original names of variables and generate the final string:

```
rename[x,y](project[x0,x3](
  select[x1=20 and x4="john" and
         x0=x2 and x2=x5 and x3=x6]
    (rename[x0,x1,x2,x3,x4,x5,x6](p))))
```

Here are two functions to solve the problem:

```
# Input is the Datalog atom.
#   data = p(x,20,x,y,"john",x,y)
# Returns pair (predicate, arguments),
# predicate = "p" and
# arguments = [('var','x'),('num',20),
#  ('var','x'),('var','y'),('str','John'),
#  ('var','x'),('var','y')]
def parseF(data):
    pass

# Input is same as output of parseF
# Returns Relational Algebra expression
# for the input
def convert2ra(rname_args):
    pass
```

V. Python Programming using (only) Expressions

In this section, it is shown that recursion can be achieved using the Y-Combinator [8] by using ideas from Lambda Calculus. The Y-Combinator is a very clever lambda expression that provides the repeated application of a function so that recursive calls, as they are understood in programming, are simulated. The `removeMultiples` function from Section III will be used to illustrate the idea. The following is the lambda-expression for the Y-Combinator in Python:

```
Y = (lambda f:
       (lambda g: f(lambda n: g(g)(n)))
       (lambda g: f(lambda n: g(g)(n)))
     )
```

The Y-Combinator takes as input a function `f`, a lambda expression, and repeatedly applies `f` to its inputs. The lambda expression for the `removeMultiples` function is shown below:

```
rmMul =
  lambda f: lambda x: lambda xs:
    [] if xs == [] else
    f(x)(xs[1:]) if xs[0]%x == 0 else
    [xs[0]] + f(x)(xs[1:])
```

The above expression is basically the recursive function from Section III in which the recursive call is replaced by a parameter to a lambda expression. So, when one provides `rmMul` as input to `Y`, i.e., `Y(rmMul)`, the implementation of `removeMultiples` recursive function without using recursion is obtained, and is shown below:

```
removeMultiples =
  ((lambda f:
     (lambda g: f(lambda n: g(g)(n)))
     (lambda g: f(lambda n: g(g)(n)))
  ))(lambda f: lambda x: lambda xs:
       [] if xs == [] else
       f(x)(xs[1:]) if xs[0]%x == 0 else
       [xs[0]] + f(x)(xs[1:]))
```

Here is a Python session that shows a sample run of `removeMultiples`:

```
>>> removeMultiples(2)([2,3,4,5,6,7,8])
[3, 5, 7]
```

As can be seen from this example, one can write entire programs in Python in the form of an expression. Of course, it is not practical to code this way in real languages, but the example illustrates the mathematical basis of functional programming languages.

VI. Related Work

The choice of a first programming language in computer science education is a widely discussed topic. Several references advocate for starting with a Functional Programming (FP) style, arguing that its core principles simplify learning fundamental concepts and lead to better programming habits.

The classic textbook titled "Structure and Interpretation of Computer Programs (SICP)" [9], which for years formed the basis of MIT's introductory CS curriculum, uses the Scheme language (a dialect of Lisp, which is functional). It advocates that by focusing on abstraction, composition of functions, and recursion early on, students gain a deeper, language-agnostic understanding of computation. It avoids early exposure to mutable state to isolate the key computational ideas.

The text "How to Design Programs" [10] is an evolution of the SICP approach, that provides a detailed, structured, and beginner-friendly design process for programming using the Racket language (a modern descendant of Scheme). It emphasizes data-driven design and writing programs via a systematic sequence of steps, which is much easier to teach and enforce with pure functional languages.

Felliesen et al. [11] advocate for the functional programming paradigm as a foundational tool in introductory computer science. Unlike industrial languages that often prioritize syntax and machine-level details, the fun"design recipe" that helps students decompose complex problem statements into concise, testable goals. The book demonstrates how a functional paradigm allows novices to focus on transferable skills—such as data representation and algebraic reasoning—rather than the "off-the-shelf" complexities of imperative languages.

The seminal paper, "Why Functional Programming Matters" [7], while not aimed exclusively at beginners, argues that the two most powerful tools for building well-structured software are higher-order functions and lazy evaluation—both hallmarks of functional programming. The author argues that these features significantly increase modularity and re-usability, which are crucial for students to master from the start.

In an important and recent paper titled "Why Functional Programming Matters in CS Education" [12] the author specifically argues for using a functional language in the first Computer Science course. He asserts that a functional language makes it easier and more efficient to teach essential topics like abstraction, modularity, recursion, and algorithmic efficiency without the complicating factor of managing state (mutation). It provides a cleaner lens through which to view fundamental computing concepts.

VII. Python Functional Programming and AI in Engineering

The engineering discipline is an inherently "functional" discipline wherein the design and implementation of systems rely heavily on building functional components, testing/proving their correctness, and composing these components in interesting ways to build larger systems. Engineers naturally think in a functional manner.

Now, in the era of Artificial Intelligence that pervades our society, engineering practitioners are creating the next generation of intelligent systems and applications that rely heavily on the hardware-software interface and Python is emerging as a key programming language. It is argued that the best practice introduced in this paper is foundational for AI. The Python Functional Programming principles taught—especially immutability and referential transparency—are considered essential foundational skills for modern AI and ML engineering. These principles are vital for building robust, debuggable, and scalable data pipelines and are directly transferable to the data-centric, stateless nature of frameworks like PyTorch and TensorFlow.

Moreover, to meet the professional education challenge of preparing professionals for an AI-driven world requires instilling principles that ensure data integrity and predictable system behavior. The proposed methodology directly addresses this by making students practice programming that intrinsically protects data from unintended modification, thereby improving code quality and reducing errors in complex AI systems.

VIII. Conclusion

In the paper, an approach to introduce functional programming in Python in introductory programming courses has been presented. The advantages of the functional programming paradigm and the ability to use features such as immutable data, pure functions, higher-order functions, and functional expressions in writing code are considered an essential skill for every Computer Scientist. Robust solutions to problems in their subsequent courses as well as in their careers can be produced by students by learning to code in a purely functional style in Python.

References

[1] A. Church, "A set of postulates for the foundation of logic," *Annals of Mathematics*, vol. 33, no. 2, pp. 346–366, 1932.

[2] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine," *CACM*, vol. 3, no. 4, p. 184–195, Apr. 1960.

[3] Scheme, "Scheme Programming Language," https://www.scheme.org/, 2025, accessed: 2025-07-25.

[4] M. R. Hansen and H. Rischel, *Functional programming in F#*. Cambridge University Press, 2013.

[5] G. Hutton, *Programming in Haskell*. Cambridge University Press, 2014.

[6] J. Whitington, *OCaml from the very beginning*. Coherent Press, 2017.

[7] J. Hughes, *Why functional programming matters*. USA: Addison-Wesley Longman, 1990, p. 17–42.

[8] "Y-Combinator: Fixed-point combinator," https://en.wikipedia.org/wiki/Fixed-point_combinator, accessed: 2025-07-25.

[9] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*. Cambridge: MIT Press/McGraw-Hill, 1996.

[10] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, *How to Design Programs: An Introduction to Computing and Programming*. MIT Press, 2001.

[11] M. Felleisen, R. Findler, M. Flatt, and S. Krishnamurthi, *How to Design Programs, second edition: An Introduction to Programming and Computing*. MIT Press, 2018. [Online]. Available: https://books.google.com/books?id=P6hcDwAAQBAJ

[12] M. T. Morazán, "Why Functional Programming Matters in CS Education," in *Proceedings of the 10th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design*, ser. FARM '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 29–33. [Online]. Available: https://doi.org/10.1145/3546193.3547051