P

Programming Language Selection for University Courses

Kevin R. Parker[1], David V. Beard[1] and Bill Davey[2]
[1]Department of Informatics and Computer Science, Idaho State University, Pocatello, ID, USA
[2]RMIT University, Melbourne, VIC, Australia

Synonyms

Algorithmic language; Computer language; Computer programming; Computer-oriented language; Computing language; Software development

Definition

Programming language selection refers to the process of choosing a language for use in the initial programming courses in an academic curriculum.

Introduction

The choice of pedagogical programming languages for introductory or advanced computer science, information systems, or informatics courses is a major decision that requires considerable thought because the programming language used in a teaching environment can significantly impact not only how courses are taught but also their effectiveness.

An ideal situation would be a single language that could be used to cover all needed paradigms and topics in at least the first one or two programming courses rather than using different languages in each course. Kummerfeld and Kay (2003, p. 105) note that "students using an unfamiliar or new programming language waste considerable time correcting syntax errors." In the authors' experience, it can take 4 or more weeks of lectures and coding exercises to teach students a second programming language, course time that could be used to cover other essential topics.

This entry examines aspects of academic programming language selection with the goal of aiding academics in choosing a language that will meet the needs of their curriculum. A sound choice will both provide a solid introduction to basic programming skills and quickly elevate those skills to a level at which informatics, software engineering, and computer science majors can be more effective in subsequent computing courses.

We start by considering the language selection decision-making process that an academic program needs to conduct, detailing a number of factors and external pressures that faculty might need to consider. Finally, we touch on some

© Springer Nature Switzerland AG 2019
A. Tatnall (ed.), *Encyclopedia of Education and Information Technologies*,
https://doi.org/10.1007/978-3-319-60013-0_131-1

emerging trends that may affect choices and changes in coming years.

The History of Language Selection

In 1978 the ACM Special Interest Group on Programming Languages (SIGPLAN) sponsored a conference on the History of Programming Languages (HOPL), during which it was proposed that language importance should be assessed based on the following criteria: (1) the language has been in use for at least 10 years, (2) the language has significant influence, and (3) the language is still in use (Bergin and Gibson 1996). A study by Soloway et al. (1989) attempted to find a better match between a language and an individual's natural skills and abilities, exploring the relationship between the preferred cognitive strategies of individuals and programming language constructs.

Howatt (1995) proposed an evaluation approach for programming languages, with criteria that included the broad categories of language design and implementation, human factors, software engineering, and application domain. Howland (1997), too, presented an extensive list of criteria that the author considered to be important in choosing a language for introductory computer science instruction but concluded that the selection of a programming language should be made primarily on the basis of how well key programming concepts may be expressed in the language.

The Computing Curricula 1991 recommendations for the introductory programming course were examined by King (1992), and he reviewed the creation of several important languages as well as the emergence of various programming paradigms during the 1980s before going on to propose his own set of criteria for the selection of programming languages.

By the turn of the century, both the object-oriented (OO) paradigm and the recognition of the importance of information security were greatly impacting language choice. In a study of language selection for CS1 and CS2 classes, the Ad Hoc Advanced Placement Computer Science (AP CS) Committee (2000) noted three main requirements: emphasis on object orientation, need for safety in the language and environment, and a desire for simplicity. Wile (2002) pointed out that programming language choice is subject to both technical and social pressures, including those of the problem domains the languages are intended to address, the conceptual and computing models that underlie the designs of the languages themselves, independent of their particular problem domains, and the social and physical contexts in which the languages are used. Roberts (2004a) observed that an increasing number of universities were adopting Java as the programming language for their introductory course, not only because of the growth in the popularity of the OO paradigm but also due to the decision by the College Board to use Java in the AP CS program. Roberts (2004b) went on to reflect that two additional challenges were negatively impacting pedagogy: (1) the number of programming details that students must master had grown, and (2) the languages, libraries, and tools on which introductory courses depend were undergoing more rapid change than in the past. Finally, Gee et al. (2005) noted a divergent trend toward the use of scripting languages to teach programming concepts because they provide not only a proper programming environment but also a tool for the formation of active web pages.

Selection Approaches

Some languages were developed with the intent of solving problems, while others were designed to make teaching algorithms easier (Parker and Davey 2012). This has led to two often conflicting alternatives about which languages should be used in university courses: should academics choose a language that is commonly used in industry, or should they choose a language that best supports concept development in students? Thus, throughout the history of language education, there have been two distinct arguments for language selection: pragmatic versus pedagogical.

Pragmatic Approach

The pragmatic approach favors choosing a language that will assist students in getting a job upon graduation. This approach views the language selection approach's most important consideration to be a language's industry acceptance as well as the marketability of individuals proficient in its use.

Industry Acceptance

Industry acceptance refers to a language's market penetration in industry (Riehle 2003), i.e., the extent to which a language is used in business and industry. Sometimes referred to as industrial relevance, this can be determined based on current and projected usage, as well as the number of current and projected positions. Stephenson (2000) rates this factor as the greatest influence in language selection, as indicated by 23.5% of schools participating in his study. Lee and Stroud (1996) note that real-world acceptability once had little weight, as indicated by the earlier use of ALGOL and Pascal, but that attitude seems to be changing. Their opinion is that students having an industrially accepted language on their resume is a significant consideration. A 2001 survey of all Australian universities indicated that perceived industry demand was the major factor in their selection of an introductory language (de Raadt et al. 2003). King (1992) concurs that many language decisions are made on the basis of current or projected future popularity, noting that there are a number of practical benefits to choosing a popular language including greater student motivation to study a language that they have heard is in demand, as well as a good selection of books and language implementations available for more popular languages.

Marketability

Marketability refers to high employability of graduates. This may include regional, national, or international marketability, depending on the placement of a program's graduates. When language selection is driven by demand in the workplace, i.e., what employers want, not only can that factor into improving the likelihood of quality future employability, but it can also increase student enthusiasm when studying a language if they feel it may improve their employability (de Raadt et al. 2003).

Language marketability is considered in several studies. In the census of introductory programming courses conducted by de Raadt et al. (2003), the most commonly listed factor in language selection (by 56% of the participants) was the desire to teach a language that equips graduates with marketable skills. Watt (2000) notes the need for transferable skills that will be useful in the students' future careers, while Emigh (2001) contends that the primary concern in language evaluation must be the demand in the workplace and employers' expectations of graduates. In fact, graduates' marketability can be further enhanced by exposing them to multiple languages (de Raadt et al. 2003). For example, a progression from C to C++ to Java will provide graduates with the qualifications for more advertised positions than exposure to any single language in isolation. Extrinsically motivated students aspiring to a lucrative career may demand to be taught those tools that are currently in vogue in the industry, and in such cases, universities may have to accept that pedagogical issues in the choice of platform and language must be secondary to marketability concerns (Jenkins 2001).

Pedagogical Selection

Many academics question whether industry trends should drive changes in curriculum and programming courses as they often seem to (Smolarski 2003; Mclver and Conway 1996; Howland 1997). Their stance is that decisions regarding the language used in an introductory course should be based on how well it underscores fundamental skills that help to make any student-developed software well-written and error-free as well as to prepare students for ensuing courses, rather than on what language is currently favored by employers (Smolarski 2003).

Avoiding the Complexities of Industrial Environments

These arguments also bring to the forefront the possibility that attempting to teach problem

solving while simultaneously introducing a professional grade language into the first course conflict because students are distracted by difficulties associated with that language and its environment (Johnson 1995; Jenkins 2002; Gee et al. 2005; Allison et al. 2002; Kelleher and Pausch 2005). A language that requires significant overhead to address even trivial problems forces the language, rather than the techniques of problem-solving, to become the object of study (Zelle 1999).

Clear Problem-Solving Principles

A teaching language should be designed in such a way that it enhances teaching the fundamentals of basic programming tasks. This is the argument espoused by Wirth (1993), Kölling et al. (1995) and many other developers of teaching languages and is commonly invoked by proponents of the various "pure" teaching languages. The argument generally concludes that it is better to adopt a language not commonly used in industry. Those sharing this viewpoint favor adopting a teaching language that promotes conceptual cleanness and efficiency as well as other instructional goals, rather than using a real-world production language whose rich feature set can introduce unnecessary complexity (Kölling et al. 1995).

Language Selection Process

Common problems that may be faced when designing an introductory course range from interdepartmental political squabbles if the course is a service course to logistical challenges if the course must accommodate a substantial number of students (Solntseff 1978). Clearly, strife surrounding the introductory programming language course and the language appropriate for that course has been ongoing for decades (Smolarski 2003). With no generally accepted approach for performing the evaluation, the selection of a programming language for instructional purposes is often a contentious task. The process often takes the form of an informal faculty discussion, with various faculty members championing their preferred language. The process continues until an eventual consensus is reached or the dominant faction wears down the opposition. As the number of faculty, students, and language options increases, this process becomes increasingly daunting. As it stands, the language selection process lacks structure and is seldom replicable (Parker et al. 2006a).

Many of the factors influencing the selection of a programming language for an introductory course at one US university is ably discussed in Smith and Rickman (1976), a study praised for its "comparable thoroughness" (Solntseff 1978). Parker et al. (2006a, b) examined over 60 papers relevant to language selection in academia and presented their criteria to be used when selecting a language for programming courses. Their study resulted in a selection approach that involves weighting each of those selection criteria based on its relative importance in each unique selection process.

Selection Criteria

Several factors should be considered when selecting a programming language. While different curricula place greater emphasis on different factors, generally all must be considered.

Parker et al. (2006a) compiled an extensive set of selection criteria and proposed a process for the application of those criteria to evaluate various languages to be used in programming classes. The selection criteria take into account the programming features of each language under consideration, the appropriateness of each of these features for teaching beginning (and perhaps advanced) programming concepts, the current and future industry acceptance of each language, the availability of quality textbooks, the costs associated with adopting each language, the infrastructure and support implications of each language, and the impact of the decision on the tactical and strategic direction of the department and curriculum. Parker et al. (2006b) carefully analyzed the preceding selection criteria. Those with commonalities were grouped together to produce the programming

Programming Language Selection for University Courses, Table 1 Higher-order selection criteria

Software cost
Reasonable financial cost
Availability of student/academic version
Programming language acceptance in academia
Academic acceptance
Availability of quality textbooks
Programming language industry penetration
Stage in life cycle
Industry acceptance
Marketability
Software characteristics
System requirements
Operating system platform dependence
Source code availability
Student-friendly features
Development environment
Debugging facilities
Language pedagogical features
Ease-of-learning fundamental concepts
Coding safety and support for secure code
Advanced features for subsequent courses
Language intent
Scripting or full-featured language
Web development support
Mobile app development support
Language design
Target application domain support
Language paradigm
Methodology or paradigm support
Teaching approach support
Language support and required training
Availability of support
Instructor and staff training
Student experience
Anticipated experience level for incoming students

language selection criteria shown in Table 1, with a few enhancements added.

Reasonable Financial Cost

Reasonable financial cost refers to the price to acquire the programming language software and/or the development environment. While this may involve individual packages or a site license for a network version, educators should inquire about academic discounts for educational institutions, an alliance in which the university or department can enroll, or even a free, downloadable version.

Availability of Student/Academic Version

If a student version or academic version is available, students can install the development environment on their personal machine, making it convenient for them to work on their assignments even if the computer lab is not accessible. However, if the department has no option other than to use a network-based version because a student version is unavailable, students may be forced to work on their assignments in campus labs, restricted by hours of operation, availability of transportation, etc. If an academic version has a limited feature set, then the benefit to the

students may be considerably reduced, but a student version should at least be considered.

Academic Acceptance

Academic acceptance refers to the popularity of a language at other academic institutions. This can be gauged by assessing current or projected use at other institutions. For example, as OO programming increased in popularity and the College Board decided to emphasize Java in the AP CS program, a number of universities, colleges, and secondary schools adopted Java as the programming language for their introductory programming courses (Roberts 2004a).

Availability of Quality Textbooks

The availability of quality textbooks is impacted by many factors. The life cycle stage of the language affects the availability of textbooks, because while it is often difficult to find a quality textbook for a newly released language, more textbooks become available over the time it takes for a language to mature. The academic acceptance of a language also plays a significant role in the availability of textbooks because the more widespread the use of a language in academia, the greater the textbook market for related textbooks. Publishers service that market by offering a larger selection of textbooks. Finally, textbook availability may also be affected by the teaching approach used. For example, functions-first, objects-first, or objects-early are all approaches used to teach OO languages, but fewer recent texts present the material from a functions-first perspective. Availability of reference books should also be taken into account (Lee and Stroud 1996).

Stage in Life Cycle

A language's stage in the programming language life cycle affects not only textbook availability, as discussed earlier, but it may also impact the widespread use of a language in both industry and academia. Universities generally prefer a language that is still in its earlier stages, rather than one like FORTRAN that is in its declining years. The programming language life cycle as described by Sharp (2002) is based on the natural principles of growth, maturation, and decay. The processes of natural advantage and evolution operate in the world of programming languages just as they operate in the biological domain, but in the case of languages, the main forces are efficiency of expression versus profitable adoption.

Industry Acceptance

As discussed earlier, industry acceptance is the market penetration of a particular language in industry (Riehle 2003). While many academic programs place great emphasis on this factor, Howland (1997) objects that too many languages are chosen simply because of their current popularity instead of on a sound pedagogical basis.

Marketability

Marketability refers to the employability of graduates and was discussed earlier when pragmatic concerns were considered. Language selection is often driven by workplace demand because marketable skills influence future employability. Emigh (2001) cautions, however, that 2 or 3 years may lapse between when a student takes an introductory programming course and when he or she enters the workforce. Even curricula that adopt newer programming languages can make no guarantee that employers will still be using those languages when the students are hired.

System Requirements

The system requirements associated with a programming language or its development environment should be considered in the selection process. The amount of hard disk space needed for software installation, operating system compatibility, and the memory requirements for optimal software performance all factor into the decision. Both student and lab computers must meet the minimum requirements of the selected language.

Operating System Platform Dependence

Some languages are platform dependent, meaning that they will only run on systems using certain operating systems. For example, the Swift programming language can only be used on macOS,

iOS, watchOS, tvOS, OS X, and Linux platforms. Other languages, such as Java, are platform independent, meaning that Java development environments are available for a variety of operating systems. Platform dependence may be a deal breaker for faculty members who are fanatic about a particular operating system.

Source Code Availability

Source code availability refers to whether a language and its associated development environment are open source or proprietary. Open-source software allows users to modify, change, and share the source code. For example, PHP is an open-source language and can be easily modified by any member of the open-source community. On the opposite end of the spectrum, Microsoft is responsible for additions, deletions, or modifications in any of the languages supported by the. Net framework, while Sun is responsible for the ongoing evolution of the Java language.

Development Environment

Development environments range from simple text editors and command-line compilers to fully interactive and integrated development environments (IDE) (McIver 2002). The development environment can improve or impair productivity (Jensen 2004). Unless an IDE is easy to use, students may be distracted by the environment rather than concentrating on learning programming concepts (Kölling et al. 1995). However, well-designed programming environments assist students in learning to program (Eisenstadt and Lewis 1992).

Debugging Facilities

Although debugging facilities may be considered part of the IDE, when assessing a programming language, one should specifically evaluate the diagnostic aids associated with the language (Tharp 1982). Programming environments must contain extensive tools for tracing and debugging (Ad Hoc AP CS Committee 2000). Error diagnostics should be clear and meaningful (Mclver and Conway 1996), and the language should not only be robust but must also be graceful in failure (Conway 1993).

Ease of Learning Fundamental Concepts

The learning curve differs greatly between languages or IDEs. For example, Python is widely considered to be simpler to learn than a .Net language like C#. Basic concepts include sequence, selection, and iteration, as well as arrays, procedures, basic input/output, and file manipulation. In addition to ease of learning, the language must have a concise syntax and straightforward semantics (Conway 1993).

Coding Safety and Support for Secure Code

Coding safety is assessed by whether the language offers features like strong type checking and array bounds checking while avoiding features like variants and pointers in unsafe mode. Kölling et al. (1995) note that a language should have a safe, statically checked type system, with no undetectable uninitialized variables and no explicit pointers. Support for secure code involves the inclusion of security-related features like Java's sandbox, which limits the memory addresses that a Java program can access.

Advanced Features for Subsequent Courses

Some academic programs use a single language to introduce basic programming concepts in an introductory course and then to progress to advanced concepts, like multithreading, in a subsequent course. In such scenarios, it is critical that the programming language includes sufficient advanced features to support an advanced programming course.

Scripting or Full-Featured Language

Educators may also want to consider scripting languages in addition to full-featured languages. Scripting languages like Python offer sufficient richness to cover most of the requirements of an introductory course while avoiding many implementation issues and reducing the complexity of the development environment. Full-featured languages, on the other hand, may offer a more

complete set of language features that an instructor may want to address.

Web Development Support

Given the prevalence of web-based systems, it is essential that today's students have the skills to develop web-based applications. Programming languages offer varying levels of web development support. Web development capabilities are not limited to scripting languages, discussed above, but are offered by full-featured languages like ASP.Net that provide a high level of support for web development.

Mobile App Development Support

Today's students make extensive use of cell phone apps. The choice to consider mobile application development limits the selection of languages that can be considered due to platform considerations. App development languages for Android are not the same as those used for Apple iOS app development. For example, two languages that have recently emerged are Kotlin and Swift. Kotlin is one of the official languages for Android development, while Swift was created by Apple to work with primarily iOS and a few additional platforms. More mainstream languages like Java, C#, and Objective-C can also be used for app development, but the field of language options is still narrowed considerably. Emulators and simulators for mobile design and development may help alleviate the platform issue.

Target Application Domain Support

Target application domain support could also be referred to as "problem domain." If the programming course is serving a particular problem domain, the extent to which a language supports programming for specific applications must be considered (Howatt 1995). Examples of application domains include FORTRAN's support for scientific programming, COBOL's support for business data processing, and RPG's support for report generation. If a programming course is being developed to serve Electrical Engineering, C or C++ may be the preferred language. If a programming course is designed to serve a data science program, languages known to support data analysis, like Python, should be considered.

Methodology or Paradigm Support

Methodology or paradigm support assesses how well a programming language supports the paradigm under consideration. For example, if the OO paradigm is being taught, then the language must support integral OO concepts like abstraction, polymorphism, inheritance, and encapsulation. Similarly, if the aim is to teach event-driven programming, then IDE support for a graphical user interface (GUI) and ease of implementing graphical components is essential.

Teaching Approach Support

Teaching approach support refers to how well a language supports the teaching approach preferred by the faculty. For example, the intent of the course may be to teach programming concepts with the language simply serving as a vehicle through which those concepts are reinforced, or it may be to teach the features of a particular language, such as using C# to develop an advanced user interface.

Availability of Support

This refers to the availability of support staff, including tutors, computer lab staff, and network administrators, to support the teaching and administration of a language. Language evaluators must consider the likelihood that their language questions, as well as those of the students, will be answered (Cunningham 2004). This should also take into account the availability of support through forums or listservs on the Internet, as well as vendor support (Tharp 1982). The availability of other resources like teachers' guides, example programs, student workbooks, and programming assignments should also be evaluated. Support may also involve something as basic as someone who can install a language and its IDE on a server, so it is accessible to all students. It may also include the availability of student tutors who have previous experience with the language.

Instructor and Staff Training

The training required for instructors and support staff, the time needed to learn a language or its IDE, and the availability of qualified instructors to teach a particular language must also be taken into consideration. Adopting a new language requires a willingness on the part of the department to invest in training its instructors because they "must continuously enrich their qualifications and implement new training methods and techniques supplemented with practical methods and techniques supplemented with practical experience while teaching a new language that is as new to them as it is to their class" (Emigh 2001, p. 2).

Anticipated Experience Level for Incoming Students

The anticipated programming experience level for incoming students is important because students' previous experience and training can positively or negatively impact their grasp of new programming paradigms and languages (Traxler 1994). If students coming into a program have been previously exposed to a particular language, then it may influence language selection. If a program consistently sees incoming students with little or no programming experience, it may be necessary to adjust its requirements and its programming language selection accordingly.

Developing Trends

Those performing programming language selection must be aware of trends that may require changes in course content. Educators must, however, take care not to be overly influenced by trends, because they are just that. Trends change from year to year, and educators must provide students with a firm grasp of the fundamentals of programming even at the risk of ignoring trends. That said, increased reliance on functional programming languages, the importance of data science, and mobile app development all have a potential impact on what is covered in courses.

Functional Programming

Functional languages use side effect free functions as a fundamental building block in the language, rather than objects or procedures.

Functional programming focuses on the application of functions to arguments. In fact, the main program is itself a function that is defined in terms of other functions. Functional programming has no implicit state and places its emphasis entirely on expressions or terms (Hu et al. 2015).

Functional programming was introduced in the 1950s and is experiencing a resurgence in recent years with the popularity of languages like Haskell, Clojure, and Scala driven by explosive growth in the use of smartphones and connected devices.

As computers, tablets, smartphones, and IoT gadgets become ubiquitous, servers can act as bottlenecks to performance. The functional programming model allows sections of software to more easily and efficiently run in parallel across different CPU cores or machines, without requiring complex synchronization. This gives the functional paradigm an edge over the OO approach when doing concurrent processing such as web requests (Puryear 2016).

Functional programming is generally more difficult for students to comprehend than other paradigms. It is more intuitive for them to view the world as a group of interacting objects rather than attempting to model everything as functions. As such, the resurgence of functional programming may seem to be of little concern when selecting a language for introductory programming courses. However, it is relevant to this discussion because "there is growing interest among some academics to introduce functional programming and functional thinking as early as possible within the computer science curriculum" (Winter 2014, p. 33).

Data Science

Whether you call it big data, data science, data analytics, or something else entirely, it may seem incongruous to include it in a discussion of programming languages. However, as the importance of data science continues to increase, the programming languages used to support data

science may begin to influence the selection of a language for programming courses. R and Python are two of the more popular open-source programming languages for data analysis, while Scala and Julia may also be encountered in this domain.

Python is of particular interest in this discussion because it is widely touted as being easy to learn in comparison with most other programming languages. Python is both easy to learn and simple to use (Newman 2012). Many features of Python facilitate both teaching and learning, such as a simple and flexible syntax, immediate feedback, easy-to-use modules, and strict requirements on proper indentation (Grandell et al. 2006). According to an analysis published on the ACM (Association for Computing Machinery) website, Python has become the most popular language for teaching introductory computer science in the United States (Guo 2014).

When taken together, the increasing importance of data analytics, the widespread usage of Python in data analytics, and Python's shallow learning curve have combined to result in many universities adopting Python as their introductory programming language.

Mobile App Development

The proliferation of smart phone usage and their ubiquitous phone apps have made mobile app development one of the hottest trends in software development. The thorniest issue, as addressed earlier, is the platform, since different programming languages are used to develop Android apps and Apple iOS apps. Android is available for dozens of smartphone models, meanwhile iOS only for a handful of iPhone and iPad models developed by Apple (Dogtiev 2017). At the time this was written, Android had 66.74% of the market, while iOS had only 31.46% (NetMarketShare 2017). Regardless of the platform chosen, projections show that by 2021, the total app downloads will jump to a stunning 352 billion (Dogtiev 2017). Mobile app development is a critical trend that may shape programming language selection for years to come.

Observations

The reality is that the choice of a programming language for introductory courses often involves compromise. While there are economic, political, and pedagogical factors that must be considered in the decision-making process, the importance of each of these factors may depend on the specific aims and priorities of the institution, educator, or course. Educators must be certain that none of the factors in the above criteria are neglected or sacrificed to more highly visible concerns (Mclver and Conway 1996).

Language selection has long been a difficult and unstructured task. Few issues in the computing education world are as strategically important or as contentious as the choice of programming language (Jensen 2004). However, the programming language used in the introductory programming course can have a significant impact on how the course is taught as well as its effectiveness (Parker et al. 2006a).

Cross-References

- ▶ Programming and Coding in Secondary Schools
- ▶ Programming Languages for Secondary Schools
- ▶ Programming Languages for Secondary Schools, Java
- ▶ Programming Languages for Secondary Schools, Pascal
- ▶ Programming Languages for Secondary Schools, Python
- ▶ Programming Languages for University Courses
- ▶ Teaching Computer Languages in Universities

References

Ad Hoc AP CS Committee (2000) Round 2: potential principles governing language selection for CS1-CS2. Retrieved 11 Nov 2017 from http://www.cs.grinnell.edu/~walker/sigcse-ap/99-00-principles.html

Allison I, Ortin P, Powell H (2002) A virtual learning environment for introductory programming. In: Proceedings of the 3rd annual conference of the learning and teaching support network centre for information and computer sciences, Loughborough, pp 48–52. Retrieved 11 Nov 2017 from https://openair.rgu.ac.uk/bitstream/handle/10059/326/Allison%20LTSN-ICS%20paper.pdf?sequence=1&isAllowed=y

Bergin TJ, Gibson RG (1996) History of programming languages-II. ACM Press, New York

Conway D (1993) Criteria and considerations in the selection of a first programming language. Technical report 93/192, Department of Computer Science, Monash University

Cunningham W (2004) Language comparison framework. Portland pattern repository. Retrieved 11 Nov 2017 from http://wiki.c2.com/?LanguageComparisonFramework

de Raadt M, Watson R, Toleman M (2003) Introductory programming languages at Australian universities at the beginning of the twenty first century. J Res Pract Inf Technol 35(3):163–167. Retrieved 11 Nov 2017 from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.84.7197&rep=rep1&type=pdf

Dogtiev A (2017) App download and usage statistics 2017. Business of apps. Retrieved 11 Nov 2017 from http://www.businessofapps.com/data/app-statistics/

Eisenstadt M, Lewis MW (1992) Errors in an interactive programming environment: causes and cures. In: Eisenstadt M, Keane MT, Rajan T (eds) Novice programming environments: explorations in human-computer interaction and artificial intelligence. Lawrence Erlbaum Associates, Hillsdale, pp 111–130

Emigh KL (2001) The impact of new programming languages on university curriculum. In: Proceedings of ISECON 2001, Cincinnati, 18, pp 1146–1151. Retrieved 11 Nov 2017 from http://proc.edsig.org/2001/16c/ISECON.2001.Emigh.pdf

Gee QH, Wills G, Cooke E (2005) A first programming language for IT students. In: Proceedings of the 6th annual conference of the learning and teaching support network centre for information and computer sciences, York. Retrieved 11 Nov 2017 from https://eprints.soton.ac.uk/261172/1/LTSN6-ProgrammingforIT.doc

Grandell L, Peltomäki M, Back RJ, Salakoski T (2006) Why complicate things? Introducing programming in high school using Python. In: Proceedings of the eighth Australasian computing education conference (ACE2006), Hobart. Retrieved 11 Nov 2017 from http://tucs.fi/publications/attachment.php?fname=inpGrPeBaSa06a.pdf

Guo P (2014) Python is now the most popular introductory teaching language at top US. Universities. Communications of the ACM Blog. Retrieved 11 Nov 2017 from https://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities/fulltext#comments

Howatt JW (1995) A project-based approach to programming language evaluation. ACM SIGPLAN Not 30 (7):37–40. Retrieved 11 Nov 2017 from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.22.2727&rep=rep1&type=pdf

Howland JE (1997) It's all in the language: yet another look at the choice of programming language for teaching computer science. J Comput Small Colleges 12(4):58–74. Retrieved 11 Nov 2017 from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.34.3832&rep=rep1&type=pdf

Hu Z, Hughes J, Wang M (2015) How functional programming mattered. Natl Sci Rev 2(3):349–370. Retrieved 11 Nov 2017 from https://doi.org/10.1093/nsr/nwv042

Jenkins T (2001) The motivation of students of programming. In: ACM SIGCSE Bulletin, Proceedings of the 6th annual conference on Innovation and technology in computer science education ITiCSE'01, 33 (3). Retrieved 11 Nov 2017 from https://www.cs.kent.ac.uk/pubs/2001/1401/content.pdf

Jenkins T (2002) On the difficulty of learning to program. In: Proceedings of the 3rd annual conference of the learning and teaching support network centre for information and computing science, Loughborough. Retrieved 11 Nov 2017 from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.596.9994&rep=rep1&type=pdf

Jensen C (2004) Choosing a language for .NET development. Borland developer network. Retrieved 11 Nov 2017 from https://edn.embarcadero.com/article/31849

Johnson LF (1995) C in the first course considered harmful. Commun ACM 38(5):99–101

Kelleher C, Pausch R (2005) Lowering the barriers to programming: a taxonomy of programming environments and languages for novice programmers. ACM Comput Surv 37(2):83–137. Retrieved 11 Nov 2017 from https://www.cs.cmu.edu/~caitlin/papers/NoviceProgSurvey.pdf

King KN (1992) The evolution of the programming languages course. In: SIGCSE'92 Proceedings of the spi1;twenty-third SIGCSE technical symposium on computer science education, pp 213–219

Kölling M, Koch B, Rosenberg J (1995) Requirements for a first year object oriented teaching language. In: Proceedings of the twenty-sixth SIGCSE technical symposium on computer science education, Nashville, pp 173–177. Retrieved 11 Nov 2017 from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.29.9469&rep=rep1&type=pdf

Kummerfeld SK, Kay J (2003) The neglected battle fields of syntax errors. In: ACE'03 Proceedings of the fifth Australasian conference on computing education 20, pp 105–111. Retrieved 11 Nov 2017 from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.11.1991&rep=rep1&type=pdf

Lee PA, Stroud RJ (1996) C++ as an introductory programming language. In: Woodman M (ed) Programming language choice: practice and experience. International Thomson Computer Press, London, pp 63–82. Retrieved 11 Nov 2017 from https://assets.cs.ncl.ac.uk/TRs/496.pdf

McIver L (2002) Evaluating languages and environments for novice programmers. In: Proceedings of the fourteenth annual meeting of the psychology of programming interest group, London, pp 100–110. Retrieved 11 Nov 2017 from http://www.ppig.org/papers/14th-mciver.pdf

McIver L, Conway DM (1996) Seven deadly sins of introductory programming language design. In: Proceedings of software engineering: education and practice (SE:E&P'96), Dunedin, pp 309–316. Retrieved 11 Nov 2017 from http://users.monash.edu/~damian/papers/PDF/SevenDeadlySins.pdf

NetMarketShare (2017) Mobile/Tablet operating system market share. Retrieved 11 Nov 2017 from https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1

Newman M (2012) Python programming for physicists. Computational physics with Python. CreateSpace Publishing, Charleston, pp 9–87. Retrieved 11 Nov 2017 from http://www-personal.umich.edu/~mejn/computational-physics/programming.pdf

Parker KR, Davey B (2012) The history of computer language selection. In: Tatnall A (ed) Reflections on the history of computing: preserving memories and sharing stories. Springer, Boston, pp 166–179. Retrieved 12 Nov 2017 from https://hal.inria.fr/hal-01526795/document

Parker KR, Ottaway TA, Chao JT (2006a) Criteria for the selection of a programming language for introductory courses. Int J Knowl Learn 2(1/2):119–139

Parker KR, Chao JT, Ottaway TA, Chang J (2006b) A formal language selection process for introductory programming courses. J Inf Technol Educ 5:133–151. Retrieved 11 Nov 2017 from http://www.jite.org/documents/Vol5/v5p133-151Parker140.pdf

Puryear M (2016) 2016's top programming trends. TechCrunch. Retrieved 11 Nov 2017 from https://techcrunch.com/2016/12/26/2016s-top-programming-trends/

Riehle R (2003) SEPR and programming language selection. CrossTalk J Def Softw Eng 16(2):13–17. Retrieved 11 Nov 2017 from https://pdfs.semanticscholar.org/af9f/8ed4fb5b02339da159ff868d82d2ec0661a7.pdf

Roberts E (2004a) Resources to support the use of java in introductory computer science. In: Proceedings of the 35th SIGCSE technical symposium on computer science education, pp 233–234. Retrieved 11 Nov 2017 from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.4693&rep=rep1&type=pdf

Roberts E (2004b) The dream of a common language: the search for simplicity and stability in computer science education. In: Proceedings of the 35th SIGCSE technical symposium on computer science education, pp 115–119. Retrieved 11 Nov 2017 from https://www-cs-faculty.stanford.edu/~eroberts/talks/SIGCSE-2004/DreamOfACommonLanguage.pdf

Sharp R (2002) Programming language lifecycles–where's Java at? Software reality

Smith C, Rickman J (1976) Selecting languages for pedagogical tools in the computer science curriculum. SIGCSE'76 Proc Sixth SIGCSE Tech Symp Comput Sci Educ 8(3):39–47

Smolarski DC (2003) A first course in computer science: languages and goals. Teach Math Comput Sci 1 (1):137–152. Retrieved 11 Nov 2017 from http://math.scu.edu/~dsmolars/smolar-e.pdf

Solntseff N (1978) Programming languages for introductory computing courses: a position paper. In: SIGCSE'78 papers of the SIGCSE/CSA technical symposium on computer science education, pp 119–124

Soloway E, Bonar J, Ehrlich K (1989) Cognitive strategies and looping constructs: an empirical study. In: Soloway E, Spohrer JC (eds) Studying the novice programmer. Lawrence Erlbaum Associates, Hillsdale, pp 853–860. Retrieved 11 Nov 2017 from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.11.8250&rep=rep1&type=pdf

Stephenson C (2000) A report on high school computer science education in five US states. Funded by IBM. Retrieved 11 Nov 2017 from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.195.2415&rep=rep1&type=pdf

Tharp AL (1982) Selecting the "right" programming language. In: SIGCSE'82 proceedings of the thirteenth SIGCSE technical symposium on computer science education, pp 151–155

Traxler J (1994) Teaching programming languages and paradigms. In: 2nd All-Ireland conference on the teaching of computing, Dublin

Watt DA (2000) Programming languages–trends in education. In: Proceedings of Simposio Brasileiro de Linguagens de Programacao, Recife. Retrieved 11 Nov 2017 from http://www.dcs.gla.ac.uk/~daw/publications/PLTE.ps

Wile DS (2002) Programming languages. In: Marciniak JJ (ed) Encyclopedia of software engineering, 2nd edn. Wiley, Hoboken, pp 1010–1023

Winter V (2014) Bricklayer: an authentic introduction to the functional programming language SML. In: Proceedings of the 3rd international workshop on trends in functional programming in education, Soesterberg, May 2014, pp 33–49. Retrieved 11 Nov 2017 from https://arxiv.org/pdf/1412.4881.pdf

Wirth N (1993) Recollections about the development of Pascal. In: HOPL-II The second ACM SIGPLAN conference on history of programming languages, pp 333–342. Retrieved 11 Nov 2017 from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.475.6989&rep=rep1&type=pdf

Zelle JM (1999) Python as a first language. In: Proceedings 13th annual Midwest computer conference (MCC 99). Retrieved 11 Nov 2017 from http://mcsp.wartburg.edu/zelle/python/python-first.html