

Teaching Programming with Python and PyGame

Dr. Vern Ceder (vceder@canterburyschool.org)

Mr. Nathan Yergler (nyergler@canterburyschool.org)

Canterbury School

Fort Wayne, IN

Canterbury School

Canterbury School, in Fort Wayne, IN, is a coed independent college preparatory school with a high school (9-12) enrollment of around 260 students. Academic expectations are high, with every student going on to college, with many attending very exclusive schools.

Since the founding of the high school in the mid 80's Canterbury's curriculum has had a strong computer science component. Classes in Pascal and C/C++ programming have been regularly offered and well subscribed, and our students have scored well the national average on Advanced Placement exams and have won regional programming contests. Our students who go on to pursue computer science in college have found themselves well prepared. Currently we teach Python at the introductory level and as an intermediate/advanced elective in addition to a year long Java track leading to the Advanced Placement exam.

In this paper we offer three conclusions based on our experience in teaching programming at Canterbury. First, Python is an excellent language for introducing complete neophytes to programming. Second, writing games can be a very effective way to teach more advanced programming, and finally Python combined with PyGame is a very effective combination for using games to teach programming.

Python in our Introduction to Computer class

In addition to traditional programming courses, there has always been a requirement that all students take an Intro to Computer course. Currently this course lasts for one quarter (approximately 8 weeks) and is taken by all 9th graders, covering word processing, spreadsheets, databases, powerpoint and programming in Python and HTML.

History of Intro

The original requirement was a year long course which covered the history of computing, number bases and boolean logic, Pascal programming, word processing, spreadsheets, and databases. By the mid 90's there were increasing pressures to revise this course. Programming was seen as being less important than before, while expertise with more sophisticated productivity and desktop publishing applications and online research skills were gaining importance. These considerations, combined with the desire to integrate more computer use into the academic curriculum and increasing demands on student and faculty time, led us to redesign our computer requirement.

In 1995 the required course was reduced to one semester and the programming component was more than cut in half, to somewhat less than one quarter. In recognition of the impending change in the Advanced Placement exam, the programming language was changed to C++, and a short unit on HTML coding was added as well.

In the fall of 2001 the required course was reduced again, to a single quarter, with only about 4 weeks allowed for programming. This change was again driven by a

combination of increased demands from other parts of the curriculum and a perceived decrease in the need for instruction in common productivity applications. This decreased need arises from the move toward more instruction in the lower grades and increased integration of such applications into the academic curriculum.

Implications of Reduced Time for Programming

The contraction of the Intro course has required us to think much more carefully about what we want to do and how we might most efficiently accomplish it. The original relatively heavy emphasis on programming seems to have been driven largely by the inclination and availability of the instructor and a vague belief that programming was a necessary skill for the high-tech world of the future. Unfortunately, none of those reasons are particularly convincing to administrators facing a growing school with an increasingly crowded curriculum. In addition, we also came to the conclusion that weeks of dreary battle with number bases and Pascal programs may not be the best way to serve students with little enthusiasm and less aptitude where programming is concerned.

Therefore we have been compelled to refine and articulate what we want our students to gain from exposure to programming:

1. Practice in critical and logical thinking. The need to break a problem down into its components and code a solution sharpens critical thinking skills.
2. Problem solving experience. The challenges of coding and debugging even a small program give great opportunities to practice problem solving techniques.
3. Basic understanding of the nature of software. While student programs may be trivial, the process of writing them gives students a very elementary understanding of how software is created and how it works.

The shortening of the amount of time available to teach programming also made us more resentful of the features of traditional programming languages which took time, but did not contribute to achieving our goals. In spite of the marketing appeal to parents of popular languages like C++ and more recently Java, we have found over the years that Pascal, C++ and Java are not good languages for introducing non-programmers to programming.

Drawbacks of Pascal, C++, Java for Beginners

In our opinion all three of these languages suffer from flaws as far as inexperienced high school students are concerned. First, Java, C++ and Pascal all require a fair amount of “magic” for even the simplest working program. By “magic” we mean those features that are necessary for a working program but beyond the experience of a non-programmer.

For example, consider explaining the following C++ version of the traditional “hello world” program to an audience of complete programming novices:

```
#include <iostream.h>
int main()
{
    cout << "hello world";
    return(0);
}
```

A complete explanation requires a discussion of include files (and the

preprocessor), libraries and the need to explicitly add input/output functions to a program (and you can see them wonder “why would anyone want to write a program WITHOUT input and output?”), and the notion of a program returning an integer value to the operating system (which will be usually ignored on their Windows systems). The slightly preferable alternative to such a mind-numbing lecture is to gloss over such features with the assurance that “you don’ t need to understand this right now, you just need to include these elements exactly this way for your program to work”. In other words, magic.

Magic like this takes time to explain and makes the subject seem less accessible. “If I can’ t even understand all the stuff in the simplest tiny program, how can I hope to handle the really hard stuff?” The magic also adds the frustration of bugs caused by typo’ s in code learned (sort of) and entered by rote. Therefore we were interested in moving away from C++ in our Intro course and staying away from Java, since both require a relatively high amount of magic for even simple programs.

The second flaw in languages like C++ and Java is that their syntax is generally confusing and often non-obvious to beginners. Beginners find such issues as the placement of semicolons somewhat baffling – “So you don’ t place a semicolon after an #include line (whatever that is) but you put one after a variable declaration, but you don’ t after an if statement, but in general you need one after every statement, which is usually but not necessarily the same as a line...” “Huh?” Similarly the distinction between equality and assignment operators is counterintuitive to most students, and a never ending source of bugs.

Finally, the need to explicitly mark code blocks with curly braces or begin-end pairs can make life difficult for neophytes as they struggle with the structure of their programs, while the lack of any layout requirements enables students to arrange their code in such a way as to actually obscure its structure. In this respect high school students are like more experienced programmers – teaching and enforcing a voluntary coding style is much more difficult than teaching the language itself.

The Decision to Try Python

This dissatisfaction with traditional languages made us receptive to alternative languages. We attended Guido Van Rossum’ s tutorials on Python at LinuxWorld 2001 and were so impressed that we made the decision to try Python in our Intro classes on the plane flying home. Since the term had already begun, we decided to let one of our two sections stay with C++ while we tested Python in the other.

The results of that test run were very positive. Students were able to produce working programs more quickly than with C++ and simple programs actually *were* simple. In addition, the ability to try their first snippets of code interactively was also helpful, and the structure enforced by Python’ s indentation made the code’ s structure more apparent. Our test run convinced us that Python was far better for an Intro course than Pascal, C++ or Java. Further, when students whose Intro experience was with Python went on to advanced courses in C++ and Java, they performed no differently than students with comparable experience in C++.

Downside of Python for Beginners

While we are enormously pleased with Python as a programming language for introductory classes, we did note a few issues which were awkward or confusing to Intro students. We want to make it clear in discussing these issues that we are only speaking

from the point of view of novice programmers. Experienced programmers might well have very different views.

First, console input presents a problem: `input()` is not appropriate for strings and `raw_input()` either requires dealing with types and typecasting or puts us back in the realm of magic. In our experience getting input is one of the largest sources of errors for Python beginners. This has led to our writing a simple input function which returns an integer, float or string, depending on the input string. (See example on <http://tech.canterburyschool.org/pycon/>) This function handles our students' needs but does have the disadvantage of not being part of the language "out of the box".

A second and related issue for novices is the way Python handles types. In contrast to the rigid insistence upon type compatibility found in C++ and Java, Python seems less predictable to the beginner. Type compatibility is not needed for assignment, is needed logically, but not syntactically, for the comparison and equality operators, and is required for the `+` operator. Confusion about this behavior is probably the second most common source of errors for inexperienced programmers.

Finally, in common with C++ and Java, Python has different operators for assignment and equality, even though its syntax enforces correct usage. Again like C++ and Java Python uses the same operator for integer or floating point division depending on the types of the arguments. This is a source of confusion to newbies since it is not obvious to them that `3/5` should evaluate differently than `3/5.0`, let alone `a/b` vs. `a/float(b)`.

The above issues illustrate why programming is viewed as nearly impossible by the uninitiated. Not only is the logical process of writing a program difficult, one also must deal with picky and apparently arbitrary rules and traps. Thankfully in Python such problems are far fewer than those in C++ or Java.

Assesment of Python in Intro

Overall, Python is the best language we have found for introducing programming to absolute beginners. After our initial trial on one section, we immediately switched the remainder of our Intro sections to Python with equally positive results. In addition, we were impressed enough with Python as a teaching language that we immediately started considering other places in the curriculum where Python would be useful.

Games, Teaching and Python

Since the mid 90' s we have used games as an important part of the programming curriculum at Canterbury. Games are used where appropriate as example programs and assignments and are allowed and even encouraged for student generated projects.

This approach is somewhat contrary to the traditional fare in school programming texts, which tends to prefer problems and examples based on more "adult" domains like business, math, and science, eg, figuring discounts, calculating volumes and finding prime numbers. As useful as such exercises might be, our preference for games is based on the following assumptions:

1. There is more engagement and therefore more learning, when the student feels that he or she is working on a "real" program, as opposed to a canned exercise.
2. In the experience of most teenagers computer games are arguably one of the most "real" types of programs.
3. When devising projects game ideas usually come easier to students than other scenarios.

4. Even simple games tend to be more complex than other types of exercises, requiring fairly complex input/output, interface design and program logic.
5. Games attract other students as testers and this interaction leads to greater motivation for debugging and improvement of the code.

Over the years Canterbury students have produced a wide variety of games of almost every type with an impressive level of complexity and polish. Usually at least once a school year a student will write a game good enough that school-wide “craze” erupts, with as much as half the student body playing the game. Such popularity and recognition gives student programmers extremely powerful positive feedback, as well as motivation for further coding and improvements. It may be only briefly, but geeks CAN be cool.

Caveats in Using Games

On the other hand, there are several issues that must be addressed in using games in a school environment. Practically speaking, there are some important programming concepts that don't obviously lend themselves to games – searching and sorting come to mind. In addition, to succeed in other programming situations, like the Advanced placement exams, college courses, even programming contests, students must be able to deal with traditional problems and case studies.

There are also considerations which are more closely related to school climate. First of all, games will understandably be viewed as “non-academic” by administrators, faculty, and parents. Without the support of the school administration, extensive and long term use of games in programming classes is not practical. Therefore it is wise to make an effort to spread the word about the value of game projects in as many ways and to as many constituencies as possible.

Secondly, it is undeniable that the time spent coding or playing games is time not spent on other subjects. We have found that student written games can experience great surges of popularity, which can be a problem in an intense academic curriculum and with students struggling to maintain their grades. Students need some faculty guidance and control to encourage them to exercise self-restraint. With most students the threat of losing the option of writing and playing their own games is enough to get their compliance.

Similarly students need to be made aware that the administration is not likely to tolerate large noisy groups of game-players disrupting the library or computer lab where academic work is taking place.

At Canterbury we have dealt with these issues by:

1. making sure that the administration and faculty understand the amount of effort and the academic value in game projects
2. making it clear that only student written games are sanctioned by the school
3. having a clear policy that research and paper writing take precedence over “testing” games for access to computers
4. making sure that students understand the need for self-control during game “crazes”

Conclusion

Overall, we have felt that using games as part of our programming courses has been very beneficial. When games are used, particularly as projects, students take on harder challenges and put in more work. In addition the possibility of having their programs used by their peers encourages more thoughtful coding and more thorough debugging.

Finally, the status gained by writing a successful game is a great motivational boost for student programmers.

Python in Electives

In addition to our introductory computer course, Python has been used in two different electives at Canterbury. The first, *Programming Games with Python*, was offered in the Spring of 2002 as a May Term elective.

Programming Games with Python

May Term is an idea borrowed from many college curricula, with dual goals of breaking the monotony of May and exposing students to a wider range of topics than the regular curriculum allows. Since the conception of May Term in the early 1990' s, Spring semester final exams have been scheduled roughly three weeks before the end of school, in mid-May. After a week of exams, most courses end and students are allowed to select focused "mini-courses," lasting the remainder of the semester. May Term courses are typically offered in areas that are of personal interest to faculty members, which leads to a wide variety of topics and courses. Past offerings have included Introductory Hebrew, Chinese Cooking (a perennial favorite), and, of course, Programming Games with Python (PGwP). We decided to offer PGwP based on our burgeoning interest in Python, Zope and PyGame. Additionally, there was a contingent of students interested in programming games; offering a Python course provided a way to meet their needs, but still require learning to take place.

Methodology

PGwP was offered to students with any previous programming experience. Enrollment was limited to 12 students, with the course filling to capacity. The non-specific prerequisite resulted in a group of students with a wide range of experience. Some had just completed the four weeks of programming in Intro, and had only seen basic Python or C++ programming (this was the end of our transition year); others had completed a year of C++ culminating in the AP Exam. The goal and sole evaluation metric for the course was the production of a simple game utilizing PyGame.

With this disparity of experience, a traditional, instructor-led introduction to Python was impractical. It would invariably be too slow or too fast for the majority of students. We instead developed a Python/PyGame tutorial. The idea behind the tutorial was to lead students through the development of a simple game from start to finish. Tic-Tac-Toe was chosen as the tutorial game for three primary reasons. First, it required a certain amount of program logic, which allowed for the demonstration of non-PyGame Python constructs. Second, it was simple enough that it could be easily deconstructed into logical, easily digested steps. Finally, Tic-Tac-Toe allows students to get something working as early as possible. More complex programs require considerably more "ground work" before any visible progress is made.

The Python/PyGame tutorial assumed no prior Python exposure, although it did rely upon some previous programming experience. No attempt was made to explain variables, functions, and other basic programming constructs, except for Python-specific details. After two days, experienced students had completed the tutorial and had begun work on their games. Novice programmers completed the tutorial in about a week.

Some students with only basic C++ exposure required additional direction. However, in the majority of cases, the tutorial was able to convey the principles of both Python and PyGame in a minimal amount of time.

Student Products and Responses

The games produced by students during the course were as varied as the students' previous experience. Some students produced simple board games, including Connect Four and Reversi. Another student attempted a "hunting" game. Overall, the games degree of complexity was directly proportional to the student's previous level of experience.

In addition to creating more complex programs, experienced programmers also had the most observations about Python and PyGame. Students with a C/C++ background came into the course with an anti-interpreted language bias. The common view was that an interpreted language could not be viable for games, due to a perceived lack of speed, low-level access, etc. PyGame answered most of those concerns. Students with previous DirectX experience appreciated the simplicity of operations that *should* be simple (blitting, etc), and found to their surprise that the speed of Python games exceeded their expectations.

These same students also discovered some of Python's potential draw-backs. The most common complaint was the lack of a Boolean type, a problem which has now been addressed (PEP 0285). Boolean types were most often missed when students used "flag" variables, and then used statements such as, "if x == True:". In this case students were pleased to find that Python allowed statements such as "if x:". Other "features" which were missing included a ternary operator (PEP 0308) and switch statement (PEP 0275). We believe the ternary operator is pure evil (especially in the hands of high school students) and so felt justified saying that it's not missing, it's just right. The students looking to use these features were typically experienced programmers, often looking for an "edge" over their novice counterparts. Instead of the ternary operator or switch statements, students were convinced to (reluctantly) use if..elif..else blocks.

Among novice programmers, PGwP was generally well received. Students who were exposed to Python in the Introduction to Computers course were generally more comfortable with the Python/PyGame combination than those who had taken C in Intro. However, even among those with previous Python exposure, the event-driven model of programming was a difficult concept after experience with only functional, linear programming models.

Programming Python: A Semester Elective

After the success of our May Term course, the decision was made to offer a full semester Python elective. Programming Python was first offered in the Fall of 2002. Like it's May Term predecessor, Programming Python had a simple prerequisite of any programming experience. The initial goal of the course was to offer a parallel to the traditional C++, Java and Pascal electives, in Python. However, within four weeks, it became clear that the class would cover much more ground than previously expected. The course was re-tooled to focus on more advanced topics than previously covered in our high school programming electives. Network/Internet programming, GUI interfaces, and game programming became integral parts of the new curriculum.

The course began with covering Python and programming primitives. With an

enrollment of 8 students, 4 had previous Python experience from Intro to Computers. The primitives were obviously review for them. The other 4 students were experienced programmers. After covering primitives and basic console programming, the course quickly moved on to functions and classes. Extra attention was paid to these areas, as they were new to most students, and were critical to the rest of the course.

It is worth noting that almost no attention was paid to Python-specific class features. This includes most double-underscore “special” methods (with the obvious exception of `__init__`), mix-ins, and introspective methods. While some students would probably have been able to understand and implement these ideas, they were viewed by the instructor as inappropriate for an intermediate, survey-style course.

After basic object oriented programming had been covered, we began to seek projects that would allow the students to explore new areas while reinforcing the basic programming concepts. We started with network programming using Python's `socket` module, and moved on to GUI programming with `TkInter`. The few students with MFC or Visual Basic experience found `TkInter` to be unintuitive and clumsy. The instructor agreed. At each step we attempted to integrate previous material with the new information. For example, after discussing `TkInter` and creating a small GUI application, students worked on a networked “chat” program. Finally, with a few weeks left in the semester, we began working with `PyGame`.

Integrating PyGame

In using `PyGame` within the context of our programming course, we decided to focus on a graphically simpler program that would require students to devote a large percentage of their time to game logic. Students were started with the Tic-Tac-Toe exercise previously used during May Term. When students were comfortable with the event model, screen drawing and the class structure, we began developing Checkers as a class project. Each student was required to complete an implementation, but cooperation and idea sharing was encouraged. While most students initially scoffed at the choice of checkers as “too simple”, they soon found that the logic behind the game was deceptively simple.

After the completion of basic, two-player checkers, students were instructed to create a “network” version. In all previous network programming exercises, the protocol was predefined as part of the assignment. However, part of the assignment in this case was defining the protocol. Student choices in this area were interesting: some chose to transmit move information in a peer-to-peer like fashion, while others required the user to designate a “server” player.

Conclusion

The overall experience of offering a Python programming elective was positive. Python's simple, clean syntax allowed the course to cover much more ground than previous programming electives. Experienced programmers probably had the most complaints about Python, although most were due to previous experience with other programming languages. Distrust of indentation blocking, switch-statement envy and discomfort with dynamic typing were characteristic of students with previous C, C++ or Java experience. This distrust of indentation blocking was typically overcome after students completed a few Python programs. These same students were used to relying on compiler type checking, and so often were caught with simple mistakes. For example,

they would become used to Python' s auto-magic type conversion for output, and then attempt to compare variables of different types (i.e., "1" is not equal to 1).

Less experienced students were sometimes tempted by Python' s advanced features. One student in particular decided that instead of input checking, he would wrap large blocks of code in try..except:pass clauses. Of course, just masking an error doesn' t work well for large projects and he spent many, many hours tracking down simple bugs that were near impossible to find without the aid of tracebacks. After teaching Python for a semester, it has become apparent that knowledge of syntax does not necessarily imply knowledge of good programming practice.

With it' s small "market share" (compared to Java and C++), Python instructors have some obstacles to overcome. First and foremost is textbooks. While Python textbooks exist, we were unable to find one which we felt was appropriate. Some of the textbooks we reviewed seem to focus on text-processing and Internet capabilities. While these are important areas where Python is useful, they are not representative of the full capabilities. Many other texts suffer from poor organization, discussing functions, classes, etc before any input or output is taught. While functional programming is an excellent goal, it has been our experience that students are more interested in immediate feedback. Programs and exercises with canned, hard-coded variables are not effective teaching tools for introductory or intermediate students. Finally, there are many books available which are useful a resources for teaching Python, but which lack exercises for students.

The Future of Python at Canterbury

Nearly two years after we began using Python at Canterbury, we are presently evaluating it' s success and it' s future. The Introduction to Computer course works much better with Python, and both students and faculty appreciate its benefits in that environment. The Programming Python elective will be offered again during the 2003-2004 school year. However, Python' s traction will be limited, even at Canterbury. The AP Computer Science exam will be moving to Java from C++ for the 2003-2004 school year. As a college preparatory school, Canterbury will continue to focus on what colleges use, or are perceived to use. Growth in the college prep school sector will come from universities and colleges using Python and promoting Python amongst their students.