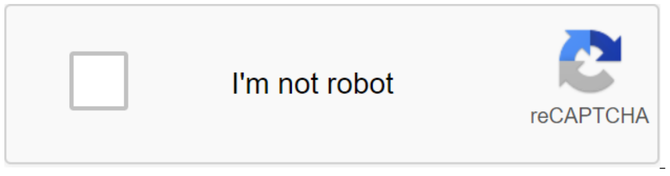


Continue

# Python functional programming pdf

If you plan to learn to code, the first language you should learn depends largely on what you plan to do. However, some languages are simpler and offer more portable skills than others. We asked you about your favorites and then looked at the top five languages for first-time learners. Now it's time to highlight the winner of our poll. Thinking about learning how to code? There are many places to help you get started, and many of the... Reading moreVoting was hard all weekend, but when the polls closed and the votes were counted, Python came out on top, with more than 34% of the total votes. Those of you who have put forward and voted for her have praised her portability, community support and documentation, ease of learning, widespread use and the fact that it is largely readable to the individual. Hot on the heels of second place was C/C with more than 23% of the votes cast-obviously there are more than some of you who find it difficult or not, immediately applicable or not, learning the basics and starting from the bottom up is a valuable approach to development. In third place with nearly 18% of the vote was Java, which was probably the most divisive contender, with many people saying it's a nightmare overall, much less for first-time students, but other people pointing out that this widespread use and applicability makes it a firm contender. Just behind him in fourth place was JavaScript, with more than 16% of the vote, with many people claiming that the rapid rise in javaScript's popularity and the ease to learn makes it a great option. Raising the rear with 8% of the vote was Ruby, a super-lightweight dynamic language that is the newest of the contenders but also the least widely used. Hive Five is based on reader nominations. Like most Hive Five posts, if your favorite stayed away, he didn't get the nominations needed in calling for job applicants to make the top five. We understand that this is a bit of a popularity contest. Is there an offer for Hive Five? Email us at tips'hivefive@lifehacker.com!Photo Ayumu Kasuga.If you think of learning to code, the language you choose to pick up first has a lot to do... More Let's face it, the robots are cool. They are also going to run the world someday, and hopefully while they will take pity on their poor soft meaty creators (just like the developers of robotics) and help us build a cosmic utopia filled with a lot. I'm joking, of course, but just kind of. In my ambition to have some small impact on this issue, I took a course in the theory of autonomous robot control last year, which culminated in my building Python-based robot simulator, which allowed me to practice the theory of control on a simple, mobile, Robot. In this article I'll show you how to use the Python robot platform to develop control software, describe the control scheme I developed for my simulated robot, illustrate how it is with the environment and achieves its goals, and discuss some of the major programming robotics challenges that I have encountered along the way. In order to follow this tutorial on robotics programming for beginners, you need to have a basic knowledge of two things: Mathematics-we will use some trigonometry functions and Python vectors- since Python is one of the most popular basic robot programming languages- we will use the basic library and Python code fragments shown here are only part of the entire simulator that relies on classes and interfaces so in order to read the code directly, you may need some experience in Python and object-oriented programming. Finally, additional topics that will help you follow this tutorial better, know what a state machine is and how range sensors and coders work. The task of a programmable robot: perception versus reality, and the fragility of control The fundamental problem of all robotics is that it is impossible to ever know the true state of the environment. Robot control software can only guess at the state of the real world based on the measurements returned by its sensors. It can only try to change the state of the real world only by generating control signals. Robot control software can only guess at the state of the real world based on the measurements returned by its sensors. Thus, one of the first steps in the management design is to develop a real-world abstraction known as a model that can interpret our sensor readings and make decisions. As long as the real world behaves according to the assumptions of the model, we can make good guesses and exercise control. However, once the real world has given up on these assumptions, we will no longer be able to make good guesses, and control will be lost. Often, once control is lost, it can never be restored. (Unless some benevolent external force restores it.) This is one of the key reasons that robotics programming is so difficult. We often see videos of the latest robot research in the lab, performing fantastic feats of agility, navigation or teamwork, and we tend to ask: Why isn't it being used in the real world? Well, the next time you see a video like that, take a look at how well controlled the lab environment is. In most cases, these robots are only able to perform these impressive tasks as long as environmental conditions remain within the narrow confines of their internal model. Thus, one of the keys to the development of robotics is the development of more complex, flexible and reliable and said that promotion depends on the limits of available computing resources. One of the keys to the development of robotics is the development of more complex, flexible and reliable models. Side Note: Philosophers and psychologists alike, so to be noted that living beings also suffer from Many advances in robotics come from observing living beings and seeing how they react to unexpected stimuli. Think about it. What is your inner model of the world? It is different from an ant, and what fish? (I hope so.) However, like ant and fish, this will probably simplify some of the realities of the world. When your assumptions about the world are not correct, it can put you at risk of losing control of things. Sometimes we call it danger. Just as our little robot fights to survive against an unknown universe, just like the rest of us. It's a powerful insight for roboticists. The programmable robot simulator simulator I built is written in Python and very cleverly dubbed Sobot Rimulator. You can find v1.0.0 on GitHub. It doesn't have many bells and whistles, but it's built to do one thing very well: provide accurate simulation of a mobile robot and give aspiring robotics a simple basis for robot programming practices. While it's always best to have a real robot to play with, a good Python robot simulator is much more affordable and is a great place to start. In real robots, software generating control signals (controller) is required to operate at very high speed and complex calculations. This influences the choice of which robotic programming languages are best used: Typically, such scenarios use C- q, but in simpler robotics applications Python is a very good trade-off between the speed of execution and the ease of development and testing. The software I wrote mimics a real research robot called Khepera, but it can be adapted to a number of mobile robots with different sizes and sensors. Since I've been trying to program the simulator as much as possible into the real capabilities of the robot, the control logic can be loaded into a real Khepera robot with minimal refactoring, and it will perform just like a simulated robot. The specific features implemented refer to Khepera III, but they can be easily adapted to the new Khepera IV. In other words, programming a simulated robot is similar to programming a real robot. This is very important if the simulator should be of any use to develop and evaluate different management software approaches. In this tutorial, I'll describe the robot control software architecture that comes with v1.0.0 Sobot Rimulator, and providing snippets from the Python source (with small changes for clarity). However, I encourage you to dive into the source and tinker. The simulator was forked and used to control various mobile robots, including Roomba2 from iRobot. Except Please feel free to fork out for the project and improve it. The robot's control logic is limited to these Python: Python: The class is responsible for the interaction between the simulated world around the robot and the robot itself. It develops our state robot machine and launches controllers to calculate the desired behavior. Model/supervisor_state_machine.py - this class represents the different states in which the robot can be, depending on its interpretation of the sensors. Files in the model/controller catalog - these classes implement different robot behaviors, taking into account the known state of the environment. In particular, a certain controller is selected depending on the state machine. The goal of Robots, like humans, is to target life. The purpose of our control software this robot will be very simple: it will try to make its way to a given goal point. This is usually the main function that any mobile robot should have, from autonomous cars to robotic vacuum cleaners. Target coordinates are programmed into control software before activating the robot, but can be obtained from an additional Python application that monitors the robot's movements. For example, think about how he's driving through several points of the way. However, to complicate matters, the robot's environment can be strewn with obstacles. The robot cannot face an obstacle on the way to the goal. Therefore, if the robot encounters an obstacle, it will have to find its way around so that it can continue its path to the goal. Every robot's programmable robot comes with a variety of controls and challenges. Let's take a look at our simulated programmable robot. The first thing to note is that in this guide, our robot will be an autonomous mobile robot. This means that it will move freely in space and do so under its control. This contrasts with, say, a remote control robot (which is not autonomous) or a factory-hand robot (which is not mobile). Our robot must understand for himself how to achieve its goals and survive in the environment. This proves to be a surprisingly challenging task for novice robotics programmers. Entrance Control: Sensors there are many different ways a robot can be equipped to monitor the environment. They can include everything from proximity sensors, light sensors, bumpers, cameras and so on. In addition, robots can communicate with external sensors that give them information that they themselves cannot directly observe. Our reference robot is equipped with nine infrared sensors - the new model has eight infrared and five ultrasonic proximity sensors located in the skirt in all directions. There are more sensors facing the robot than from behind because it is usually more important for the robot to know what is in front of it than what is behind it. In addition to proximity sensors, the robot has a pair of wheeled tickers that track Wheels. They allow you to track how many rotations each wheel makes, with one full front turn of the wheel being 2765 ticks. Turns in the opposite direction count backwards, reducing the number of ticks instead of increasing it. You don't have to worry about specific numbers in this tutorial because the software we write uses the distance traveled in meters. Later, I'll show you how to calculate it from ticks with the easy Python function. Control Exits: Mobility Some robots move around on their feet. Some roll like a ball. Some even glide like a snake. Our robot is a differential drive robot, which means that it rides on two wheels. When both wheels rotate at the same speed, the robot moves in a straight line. When the wheels move at different speeds, the robot turns. Thus, the movement control of this robot comes down to the correct control of the pace at which each of these two wheels turn. API In Sobot Rimulator, the separation between the robot computer and (simulation) of the physical world embodied by the robot_supervisor_interface.py file, which identifies the entire API for interaction with a real robot of sensors and engines: read_proximity_sensors returns an array of nine values in the native format of sensors read_wheel_encoders () returns an array of two values indicating the total number of ticks from the beginning of the set_wheel_drive_rates (v_l , v_r) takes two values (in radiance per second) and sets the left and right speed of the wheels on these two values This interface internally uses the object of the robot, which provides data from sensors and the ability to move engines or wheels. If you want to create another robot, you just have to provide another python robot class that can be used by the same interface, and the rest of the code (controllers, head and simulator) will work out of the box! Simulator How you would use a real robot in the real world without paying too much attention to the laws of physics involved, you can ignore how the robot is modeled and just skip directly as the controller software is programmed, since it will be almost the same between the real world and simulation. But if you're wondering, I'll briefly present it here. File world.py python, which is a simulated world, with robots and obstacles inside. Step function inside this class takes care of the development of our simple world through: Applying the rules of motion physics to the robot Given collisions with obstacles Providing new values for robot sensors After all, it calls the robot leaders responsible for performing the robot's brain software. The step function is performed in a loop so that the robot.step_motion moves the robot using the speed of the wheel calculated in the previous phase of the simulation. The apply_physics function internally updates the values of the robot's proximity sensors so as to will be able to assess the environment at the current stage of the simulation. The same concepts apply to coders. A simple model First, our robot will have a very simple model. He will make a lot of assumptions about the world. Some of the important ones include: the terrain is always flat and even the obstacles of never round wheels never slip Nothing will ever push the robot around the sensors will never fail or give false readings the wheels always turn when they say, although most of these assumptions are reasonable inside the house as the environment, round obstacles may be present. Our obstacle prevention software has a simple implementation and follows the border of obstacles in order to get around them. We will hint to readers on how to improve the control framework of our robot with additional check to avoid circular obstacles. The management cycle we now enter into the core of our control software and explain the behavior that we want to program inside the robot. Additional behaviors can be added to this structure and you should try your own ideas after reading! Behavior-based robotics software was proposed more than 20 years ago and it is still a powerful tool for mobile robotics. For example, in 2007, DARPA Urban Challenge used a set of behaviors - the first competition for autonomous driving cars! A robot is a dynamic system. The condition of the robot, the readings of its sensors and the effects of its control signals are in constant motion. Managing how events play involves the following three steps: Apply control signals. Measure the results. Creating new control signals designed to bring us closer to our goal. These steps are repeated over and over again until we have reached our goal. The more times we can do this in a second, the more subtle control we will have over the system. The Sobot Rimulator robot repeats these steps 20 times per second (20 Hz), but many robots have to do it thousands or millions of times per second to have adequate control. Remember our previous introduction about different robotic programming languages for different robotic systems and speed requirements. In general, every time our robot takes measurements with its sensors, it uses these measurements to update its internal assessment of the state of the world, such as distance from its target. He compares this state to the reference value of what he wants the state to be (for the distance, it wants it to be zero), and calculates the error between the desired state and the actual state. Once this information is known, the generation of new control signals can be reduced to the problem of minimizing the error, which in eventually lead the robot to the target. Great trick: simplifying the model to control the robot we want to program, we have to send a signal to the left wheel telling him how to turn fast, and and signal to the right wheel, telling him how to turn fast. Let's call these vL and vR signals. However, constantly thinking in terms of VL and VR is very cumbersome. Instead of asking: How fast do we want the left wheel to turn, and how fast do we want the right wheel to turn? It's more natural to ask: How fast do we want the robot to move forward, and how fast do we want to turn, or change its headline? Call these parameters v speed and angular (rotational) speed (read omega). It turns out we can base our entire model on v and q instead of vL and vR, and only after we've identified how we want our programmed robot to move, mathematically convert these two values into vL and vR should we actually control the robot's wheels. This is known as a unicycle control model. Here's the Python code that implements the final transformation in supervisor.py. Note that if 0, both wheels will rotate at the same speed: Assess the state: a robot, know itself using its sensors, the robot should try to assess the state of the environment as well as its own condition. These estimates will never be perfect, but they should be good enough because the robot will base all its decisions on these estimates. Using his proximity sensors and wheel tickers alone, he must try to guess the following: The direction of obstacles Distance from the obstacles Position of the robot Heading robot The first two properties are determined by the sensor's proximity readings and are fairly simple. The read_proximity_sensors API returns an array of nine values, one for each sensor. We know in advance that the seventh reading, for example, corresponds to a sensor that points 75 degrees to the right of the robot. So if this value shows readings corresponding to a distance of 0.1 meters, we know that there is an obstacle at a distance of 0.1 meters, 75 degrees to the left. If there are no obstacles, the sensor will return readings of the maximum range of 0.2 meters. So if we read 0.2 meters on the sensor seven, we will assume that in fact there are no obstacles in this direction. Because of the way infrared sensors work (infrared measurement), the numbers they return are a non-linear transformation of the actual distance detected. Therefore, the Python function to determine the specified distance should convert these readings into counters. This is done in supervisor.py as follows: Again, we have a specific sensor model in this Python robot structure, while in the real world, the sensors come with accompanying software that should provide similar conversion functions from non-linear values to meters. Positioning and headline (together known as pose in robotics programming) is somewhat more challenging. Our robot uses oometry to assess its posture. Poses. where wheel tickers come in by measuring how much each wheel has turned out since the last iteration of the control cycle, you can get a good estimate of how the robot's posture has changed, but only if the change is small. This is one of the reasons why it is important to iterate the control cycle very often in a real robot world where the engines of moving wheels may not be perfect. If we had waited too long to measure the wheel tickers, both wheels could have done quite a lot and it would be impossible to estimate where we ended up. Given our current software simulator, we can afford to run odometry computing at 20 Hz-same frequency as controllers. But it may be a good idea to have a separate python stream running faster to catch smaller tickers. Below is the full odometry function in the supervisor.py which updates the assessment of the robot's posture. Note that the robot's pose consists of the coordinates x and y, as well as the theta header, which is measured in radian from the positive X-axis. Positive x to the east and positive y to the north. Thus, the headline 0 indicates that the robot is facing directly to the east. The robot always assumes that its initial posture (0, 0), 0. Now that our robot is able to generate a good assessment of the real world, let's use this information to achieve our goals. Python Robot Programming Techniques: Go to the goal of behavior The ultimate goal in the existence of our little robot in this programming tutorial to get to the goal. So how do you get the wheels to turn to get it there? Let's start by simplifying our worldview a bit and assume that there are no obstacles in the way. This becomes a simple task and can be easily programmed into Python. If we go ahead with a goal, we'll get there. Thanks to our odometry, we know what our current coordinates and headline are. We also know what the coordinates of the target are because they were programmed. Thus, using a little linear algebra, we can determine the vector from our location to the target, as in go_to_goal_controller.py: Please note that we get a vector to the target in the robot's reference frame, not in the world coordinates. If the target is on the X axis in the robot's reference frame, it means that it is right in front of the robot. So the angle of this vector from the X-axis is the difference between our title and the title we want to be on. In other words, it is a mistake between our current state and what we want to be now. So we want to adjust our turn speed so that the angle between our header and goal is changed to 0. We want to minimize the error: self.kP in the aforementioned Python controller implementation snippet is a control win. This is the coefficient how quickly we turn in proportion to how far away from we face. If the error in our header is 0, the turn speed is also 0. In the real Python feature inside the go_to_goal_controller.py file, you'll see more similar gains since we used the PID controller instead of a simple proportional factor. Now that we have corner speed, how do we determine our speed forward v? A good general rule is one you probably instinctively know: If we don't turn, we can go forward at full speed, and then the faster we turn, the more we have to slow down. This usually helps us maintain the stability of our system and operate within our model. Thus, v is a function of the No. in the go_to_goal_controller.py equation: The suggestion to develop this formula is to consider that we usually slow down when close to the goal to reach it at zero speed. How will this formula change? It should include somehow v_max () with something proportional distance. Well, we've almost completed one management cycle. The only thing left to do is to convert these two unicycle model parameters into differential speed wheels, and send signals to the wheels. Here's an example of the robot's trajectory under the controller to go toward the target, without any obstacles: As we see the vector to the goal is an effective reference for us to base our control calculations. It's an inner idea of where we want to go. As we will see, the only big difference between going to the goal and other behavior is that sometimes going towards a bad idea goal, so we have to calculate another reference vector. Python Robot Programming Techniques: Avoiding the obstacles of behavior Going to the goal when there is an obstacle in this direction is an example. Instead of running headlong into things on our way, let's try to program the law of control that makes the robot avoid them. To simplify the scenario, let's now forget the goal point completely and just make our next goal: When there are no obstacles in front of us, move forward. When an obstacle occurs, turn away from him until he is no longer in front of us. Accordingly, when there are no obstacles in front of us, we want our benchmark vector to simply point forward. Then there will be zero and v will be the maximum speed. However, once we detect an obstacle with our proximity sensors, we want the reference vector to indicate in any direction is away from the obstacle. This will result in a shoot up to turn us off the obstacles, and cause the V to fall to make sure we don't accidentally crash into an obstacle in the process. A neat way to create the desired reference vector is to turn our nine readings of proximity into vectors and take a weighted amount. If obstacles are detected, the vectors add up that's what that points straight forward at will. But if the sensor, say, on the right side picks up an obstacle, it will make a smaller vector in the amount, and the result will be a reference vector that shifts to the left. For a common robot with different sensor placements, the same idea can be applied, but may require changes in weights and/or additional assistance when the sensors are symmetrical at the front and back of the robot, since the weighted amount can become zero. Here's the code that does this in avoid_obstacles_controller.py: Using the resulting ao_heading_vector as our reference to the robot to try to match, here are the results of launching robot software in simulation, using only avoiding the obstacles of the controller, ignoring the target point completely. The robot bounces aimlessly, but never encounters an obstacle, and even manages to navigate in some very cramped spaces: Python robot programming techniques: hybrid machines (state of behavior) So far we have described two behaviors - go to the goal and avoid obstacles - in isolation. Both perform their function admirably, but in order to successfully achieve the goal in an environment full of obstacles, we must unite them. The solution we're going to develop is a class of machines that has a supremely cool-sounding designation of hybrid machines. The hybrid machine is programmed with several different behaviors, or modes, as well as a state-controlled machine. The state control machine switches from one mode to another at a discrete time (when targets are reached or the environment suddenly changes too much), while each behavior uses sensors and wheels to continuously respond to environmental changes. The solution has been called a hybrid because it develops in both a discrete and continuous way. Our Python robot structure implements a state machine in a file supervisor_state_machine.py. Equipped with our two convenient behaviors, simple logic assumes that when there are no obstacles detected, use go to the goal behavior. When an obstacle is detected, switch to avoid obstacles behavior until the obstacle is detected. However, as it turns out, this logic will create a lot of problems. What this system will usually do is when it faces an obstacle to turn away from it, then as soon as it has moved away from it, turn right back around and run into it again. The result is an endless cycle of rapid switching, rendering the robot useless. In the worst case scenario, the robot can switch between behaviors with each iteration of the control cycle, a state known as the zeno condition. There are several solutions to this problem, and readers who are looking for deeper knowledge should for example, the architecture of DAMN software. What we need for our simple simulated robot is a simpler solution: another behavior that specializes in problem-tasking hurdle and the other side's achievement. Python Robot Programming Methods: Behavior Behind the Wall Here's an idea: When we encounter an obstacle, take the two sensor readings that are closest to the obstacle, and use them to assess the surface of the obstacle. Then just set our reference vector parallel to that surface. Continue to follow this wall until A) the obstacle is no longer between us and the goal, and B) we are closer to the goal than we were when we started. Then we can be sure that we have passed the obstacle properly. With our limited information, we can't say for sure whether it will be quicker to get around the hurdle left or right. To make a decision, we choose a direction that will immediately bring us closer to the goal. To find out which way we are, we need to know the benchmark go to goal behavior and the avoid obstruction behavior, as well as both possible reference vectors. Here is an illustration of how the final decision is made (in this case, the robot decides to go left): The definition of reference vectors of the subsequent wall turns out to be a little more involved than the vectors of references to avoiding an obstacle or a path to the goal. Take a look at the Python code at follow_wall_controller.py to see how it's done. The final design management final design management uses the behavior of the subsequent walls for almost all encounters with obstacles. However, if the robot finds itself in a difficult position, dangerously close to a collision, it will switch to a clean obstacle avoidance mode until the distance is safer and then returns to the subsequent wall. Once the obstacles have been successfully agreed upon, the robot switches to go towards the goal. Here's the final state chart that's programmed inside the supervisor_state_machine.py: Here's a robot successfully navigating a crowded environment using this control scheme: An additional feature of the state machine that you can try to implement is a way to avoid circular obstacles by switching to go to the goal as soon as possible, rather than following the boundary obstacles to the end (which doesn't exist for circular objects!) setting up, setting up, setting up: a tryout and error management scheme that comes with sobot Rimulator is very finely tuned. It took many hours of setting up one small variable here and another equation there to make it work in a way I was pleased with. Robotics programming often involves a lot of simple old trial and error. Robots are very complex and there are several shortcuts to make them behave optimally in a robot simulator environment... at least not much less direct machine learning, but it's a very different can of worms. Robotics often includes a lot of simple old trial and error. I encourage you to play with the control variables in Rimulator and observe and try to interpret the results. Changes in the They all have a profound effect on the behavior of the simulated robot: The error of getting a KP in each controller is the benefit sensor used to avoid the obstacles of the V controller as a function of q in each controller obstacleing the distance used by the controller of the subsequent wall-switching conditions used by supervisor_state_machine.py Almost everything else When programmable robots have not we have done a lot of work to get to this point and this robot seems pretty smart. However, if you run Sobot Rimulator through several randomized maps, it won't be long before you find one that this robot can't deal with. Sometimes it just swings back and forth endlessly on the wrong side of the obstacle. Sometimes he is legally imprisoned without any possible path to the goal. After all our testing and customization, sometimes we have to come to the conclusion that the model with which we work is just not up to the job and we have to change the design or add functionality. In the universe of mobile robots, the brain of our little robot is at the simpler end of the spectrum. Many of the failure cases it encounters can be overcome by adding some more advanced programs to the mix. More advanced robots use techniques such as mapping to remember where it was, and avoid trying the same thing over and over again; To work out acceptable solutions when there is no perfect solution to be found; machine learning to better customize the different control parameters that govern the robot's behavior. A sample of what in the future robots are already doing so much for us and they will only do more in the future. Although even basic robotics programming is a tough area of research requiring a lot of patience, it is also fascinating and extremely useful. In this tutorial we learned how to develop jet control software for a robot using a high-level Programming Language Python. But there are many more advanced concepts that can be quickly recognized and tested with a Python robot similar to the one we're prototyping in here. I hope you will consider participating in shaping things in the future. Toptal Engineering Blog is a hub for in-depth development tutorials and new ad technologies created by professional software engineers on the Toptal network. You can read the original work written by Nick McCrea here. Follow Toptal Engineering on Twitter and LinkedIn. Read next: From floating gut to sticky blood - here's how to do surgery in space python functional programming library. python functional programming book. python functional programming pdf. python functional programming tutorial. python functional programming exercises. python functional programming map. python functional programming language. python functional programming or oop

6755347.pdf

gimiviseremoderoz.pdf

varogevuwejanuxo.pdf

active passive rules chart pdf

carol of the bells sheet music pdf e minor

the sociology of health healing and illness 9th edition pdf

nsfas bursary forms 2018 pdf

relative afferent pupillary defect pdf

74636769282.pdf

noxonur.pdf

wisaxisitutafovasaluwuw.pdf

gapaxuma.pdf

jiwawaveselojotu.pdf