Core Python: Numeric Types, Dates, and Times

by Austin Bingham and Robert Smallshire

Start Course

Bookmark Add to Channel Download Course

Table of contents Description **Transcript** Exercise files Discussion Related

Course Overview

Course Overview

[Autogenerated] Hi, everyone. My name is Austin Bingham and welcome to my course core python numeric types, dates and times. I'm a founder and principal consultant at 60 North. Basic python numeric types and 10 float are familiar to most programmers and are the right tools for many numerical computations. There are times, though, when more specialized numeric types are needed and where, naively using the basic types can lead to incorrect results. In this course, we look at a number of specialized America types, the times when it's appropriate to use them and what can happen if you use inappropriate numeric types. Some of the major topics will cover include the decimal type for performing precise face. 10 calculations. The fraction type for representing rational numbers using integral numerator is and denominators, the complex type for working with numbers with an imaginary component built in functions for working with numeric types of generically pythons. Date time package for working with dates and times. By the end of this course, you'll have a good overview of the numeric types that python provides, and you'll start to develop a sense of when and how to best deploy them before beginning this course you should be familiar with the basic numeric types and and float. You should also be broadly familiar with the basics of the Python language, including functions and working with modules. I hope you'll join me on this journey to learn more about numeric types and python with E Corp python numeric type states and Times course at plural site.

Review of int and float

Review of int and float

[Autogenerated] python comes equipped with a number of numeric types, and you should already be familiar with the two most basic and and float in. This module will briefly review Intend Float to set the field for discussing some of the other numeric types in the core language and the standard library in this module of core python numeric types, Dates and times will review the essential characteristics of into and float in particular will lick it qualities of these types that make them unsuitable for certain applications. We'll provide further references for those interested in learning more about the fascinating field of numerical computation in earlier core python courses like core Python Getting started, we extensively used to built in numeric types into and float. We've seen that the pipe and three in objects can represent integers That is, whole numbers of arbitrary magnitude, limited only by practical constraints of available memory and the time required to manipulate large numbers. This sets python apart from many other programming languages, where the standard integer types have fixed size, storing only 16 32 or 64 bits of precision python handles large integers with consummate ease. Pythons float type, an abbreviation of floating point number is specifically a 64 bit floating point number using a binary internal representation officially known as binary 64 in the IEEE 754 Standard. For those of you with background and see derived languages, this is commonly known as a double, although that terminology is inappropriate in the context of python, and we do not use it here. Of the 64 bits within a python float, one is allocated to representing the sign of the number, 11 are used to represent the ex opponent, the value to which the fraction is raised. The remaining 52 are dedicated to representing the fraction, also known as the Mantis A or significant, although owing to the way the encoding works in conjunction with the sign, we get effectively 53 bits of precision. This means that thinking in terms of decimal equivalents, python floats have at least 15 digits of decimal precision and no more than 17 digits of decimal precision. In other words, you can convert any decimals with 15 significant figures into python floats and back again without loss of information. Python floats support a very large range of values larger than would be required in most applications. To determine the limits of the float type, we can query the cyst out float info object from the built in sis module. You can see that the largest float is around 1.8 times 10 to the 308 and the smallest float greater than zero is around 2.2 times 10 to the negative. 308. If we want the most negative float or the greatest floats smaller than zero, we can negate the two values, respectively, so floats can represent a huge range of numbers, much larger than required in most applications, although you should be aware of their limitations. First of all, you shouldn't assume, in general that any python in't could be converted without loss of information to a python float. The conversion is obviously possible for small magnitude integers, but because the Mantis A has only 53 bits of binary precision, we can't represent every integer above two to the 53rd. Let's demonstrate that with a simple experiment at the rebel here, receive, for example, that the float value for the indigent to to the 53rd it's the same as the float value for 2 to 53rd plus one. Furthermore, because the float type has finite precision, some fractional values can't be represented accurately in much the same way that 1/3 can be represented as a finite precision decimal. For example, neither 0.8 nor 0.7 can be represented in binary floating point, so computations involving them return incorrect answers rounded to a nearby value, which can be represented if you're not familiar with floating point mathematics. This can seem shocking at first, but it's really no less reasonable than the fraction 2/3 not displaying an infinitely recurring series of sixes. A full treatment of careful use of floating point arithmetic is well beyond the scope of this course. But we do want to alert you to some of the issues to motivate the introduction of alternative number type supported pie python, which avoid some of these problems by making different trade offs. If you do want to learn more about floating point, we recommend David Goldberg's classic what every computer scientists should know about floating point arithmetic. Let's review what we've covered in this module. Pythons and type represents whole numbers of arbitrary magnitude. Pythons float type represents floating point numbers. Float uses 64 bits to sorts. Value one bit for the sign. 11 bits for the exponents and 52 bits for the Mantis. A pythons float is the same as double in the sea family of languages and, like double, it can't exactly represent every floating point value in its range. In the next module of core python numeric types, dates and times will look a decimal pythons numeric type representing decimal values. Exactly. Thanks for watching, and we'll see you in the next module.

The Decimal Module

Construction

[Autogenerated] Pythons float is a great tool for many computations, but there are situations. For example, in many financial calculations were we require exact decimal representations that float can't provide in this module of core python numeric types, dates and times. Well, look at the decimal type and the standard libraries a decimal module, and we'll see how it allows us to represent decimal values exactly without a loss of precision. We'll learn about the relationship between decimal and other numeric types and the pitfalls you can encounter when you mix them. We'll see how decimal preserves precision across calculations. We'll use the decimal modules context to control the precision of our calculations. Results. We'll investigate certain special values for decimal and will explore the surprising behavior of certain operators when used with decimal. As we've seen, the python float type can result in problems with even the simplest of decimal values, which would be unacceptable in any application where exact arithmetic is needed, such as in a financial setting. The decimal type, spelled with an uppercase D in the decimal module, spelled with a lower case D, is a fast, correctly rounded number type for performing arithmetic in base 10 Crucially, the decimal type is still a floating point type, albeit with a base of 10 rather than two, and has finite precision. Although this precision is user configurable rather than fixed, using decimal in place afloat for, say, an accounting application could lead to significantly fewer hard to debug edge cases. Let's take a look. We'll start by calling decimal dot get context to retrieve information about how the decimal system has been configured. The most important figure here is preK, which tells us that by default, the decimal system is configured with 28 points of decimal precision. Some of the other values in here control rounding and error signaling modes, which could be important in certain applications. We create decimal instances by calling the decimal constructor. This is obvious enough when creating a decimal from an integer, although it's a little awkward to use the module name every time. So let's pull the decimal type into the current scope. Notice, however, that when the rebel echoes the representation of the decimal object back to us, it places quotes around the seven, indicating that the constructor also accepts strings. Let's exercise that and replicate the computation that gave an inexact answer with float in the previous module. Using decimal for this computation gives us an exact answer

Fractional Values, Precision, and Special Values

[Autogenerated] for fractional values. Passing the literal value to the constructor as a string can be very important. Consider this example without the quotes, we're back to the same problem we had with floats. To understand why. Let's deconstruct what's going on here. We've typed two numbers 0.80 point seven in base, 10 into the rebel. Each of these numbers represents a python float, so python converts are literal based 10 representation into internal based to representations within the float objects. Neither of the values we have chosen can be represented exactly in base. To sow some rounding occurs. These rounded float values are then passed to the decibel constructor and used to construct the internal based 10 representations, which will be used for the computation. So although the decimal constructor supports conversion from float, you should always specify fractional decimal liberals as strings tow. Avoid the creation of an inexact intermediate base to float object. Finally, the subtraction is performed on the decimal objects. To avoid inadvertently constructing decimal objects from floats, we could modify the signal handling in the decimal module. This also has the desirable effect of making comparisons between decimal and float objects. Raise an exception to here. We carefully constructed decimal from a string on the left hand side of the expression, but use afloat on the right hand side decimal. Unlike float preserves the precision of a kn argument that includes trailing zeros. This store precision, is propagated through computations. The precision of constructed values has preserved whatever the precision setting in the module context, but only comes into play when you perform computations. First, we'll reduce the position down to just six significant figures and then create a value which exceeds that precision. Now, when we perform a computation, we see the limited context precision kick in. That is, even though one of our terms has seven digits of precision. The result of our addition is limited to six digits precision. We should also point out that, like the float type, decimal supports the special values for infinity and not a number thes propagate, as you would expect through operations

Combining with Other Types

[Autogenerated] as we've seen, decimals can be combined safely with python integers, but the same cannot be said of floats or, more generally, other numeric types. Arithmetic operations with floats will raise a type error. This is, by and large a good thing, since it prevents inadvertent precision and representation problems from creeping into programs. Note, however, that unless you disable it in the context, you can compare decimals with floats. Decimal objects play very well with the rest of python and usually wants any input. Data has been converted to decimal objects. Program code could be very straightforward and proceed as it would with floats. And it's that said, there are a few differences to be aware of. One difference is that when using the module, ISS or remainder operator, the sign of the result is taken from the first operandi. The dividend, rather than from the second operandi, the divisor. So the calculation negative seven module is three. Within its produces the value, too. This means that negative seven is to greater than negative nine, the largest multiple of three, which is less than negative seven. For python integers, the result of the module is operation always has the same sign as the divisor here. Three. And to have the same sign, however, with the decimal type module issues, is a different convention of returning result, which has the same sign as the dividend. If we evaluate decibel negative, seven module is decimal. Three. We get decimal negative one. This means that negative seven is one less than negative six, the next multiple of three toward zero from negative seven. It may seem capricious that Python has chosen different modules conventions for different number types, and indeed it's somewhat arbitrary which convention different programming languages use. But it works this way so that float retains compatibility with Legacy Python versions, whereas Decimal is designed to implement the IEEE 854 decimal floating point standard. One result of this is that widespread implementations of common functions may not work is expected with different number of types. Consider a function to test whether a number of odd, which is typically written like this. This works well for ends, and it works for floats. But when used with decimal, it fails for negative odd numbers. This is because the negative one is not equal to positive one to fix. This weekend right is odd as a not even test, which also works for negative decimals to maintain consistency and preserve the important identity. X equals X floor divided by why Times y plus X module is why the interview division operator also behaves differently. So for integers, negative seven floor divided by three gives us negative three. This means that three divides into the largest multiple of three less than negative seven, which is negative. Nine. Negative three times. However, with decimals, the result is decimal negative, too. This means that three divides into the next multiple of three towards zero from negative seven, which is negative. Six negative two times. It's confusing that the double forward slash operator is known in Python S, the floor division operator, but has not been implemented this way. In the case of decimal, rather four decimal it trunk eights toward zero. So it's perhaps better to think of Double Ford slash as simply the interview division operator who semantics are type dependent. The functions of the math module cannot be used with the decimal type, although some alternatives are provided as methods on the decimal class. For example, to compute square roots used the sq Artie Method. A list of other methods supported by decimal can be found in the python documentation at python dot org's before moving on. Let's recap which kinds of values that America types we have in our toolkit so far can represent. Although Float cannot exactly represent 0.7, this number can be explicitly represented by decimal. Nevertheless, many numbers, such as 2/3 cannot be represented exactly in either binary or decimal float point representations. To plug some of the gaps in the real number line, we must turn to 1/4 number type for representing rational fractions.

Summary

[Autogenerated] Let's review what we covered in this module. The decimal type uses a base 10 floating point number representation and thus can exactly represent decimal values. Decimals can be safely constructed with strings and integers, but constructing them with floats can lead to a loss of data. To help avoid this problem, the decimal context could be configured to disallow construction from and other operations with floats. Decimals preserve and propagate their precision across calculations, but the decimal context can be used to limit the precision of computations. Decimal supports the special infinity, and not a number values certain. Operators like module is behave differently with decimal than with other numeric types. Functions in the math module do not work with decimal objects. In the next module of core python numeric types, dates and times, we'll look at fraction pythons numeric type for representing rational numbers. Thanks for watching, and we'll see you in the next module

The Fractions Module

Construction

[Autogenerated] both float and decimal or floating point representations with different characteristics, and both are limited by the fact that they can't exactly represent even seemingly simple, rational numbers. Like 2/3 in this module of core python numeric types, dates and times we look at the fraction type from the standard library fractions module. We'll see how it lets us represent rational numbers exactly. By storing integral numerator is and denominators. We'll cover constructing fractions from other types like floats and strings, while demonstrating how this can produce unexpected results. When done naively well, look at arithmetic with fractions and see how they could be used with standard library math module. The Fractions module contains Thief Fraction type for representing so called rational numbers. Rational numbers consist of the quotient of two integers, such as the number 2/3 with the numerator of two and the denominator of three, or in the number 4/5 with the numerator of four and a denominator of five. An important constraint on rational numbers is that the denominator must not be zero. Let's see how to construct fractions, the first form of the constructor rule. Look at accepts two integers for the numerator and denominator respectively. We represent 2/3 with fraction to three. Similarly, we can represent 4/5 with Fraction 45 This is also the form in which the fraction instances echoed back to us by the rebel. Attempting to construct with zero denominator raises a zero division error. Of course, given that the denominator can be won any and however large can be represented as a fraction by passing it as the only argument diffraction.

Arithmetic and Operations

[Autogenerated] fractions can also be constructed directly from float objects. Be aware, however, that if the value you expect can't be exactly represented by the binary float such a 0.1, you may not get the results you bargained for. Fractions support interoperability with decimal, though, so if you can represent the value as a decimal, you get an exact result. Finally, as with decimals, fractions could be constructed from a string. Arithmetic with fractions is without surprises. You can add them, subtract them, multiply them, divide them, performed floor division and ma jealous. Notice how decimal calculates the least common denominator when the operate denominators are different. Unlike decimal, Fraction type does not support methods for square roots and similar operations. This is for the simple reason that the square root of a rational number, such as to maybe an irrational number and not represent a bill as a fraction object. However, fraction objects can be used with the math dot seal and math dot floor functions, which return integers between them python. It's floats, decimals and fractions allow us to represent a wide variety of numbers on the real number line, with various trade offs and precision exactness convenience and performance later in this course module will provide a compelling demonstration of the power of rational numbers for robust computation. Let's review what we've covered in this module. The fraction type allows us to represent rational numbers by storing integral numerator is and denominators. Fractions could be constructed from one or two integers, but will raise a zero division error. If you pass zero as the denominator, you can construct fractions from floats with. This may produce unexpected results. If you use floats within exact representations, you can construct fractions from decimals and strings with unsurprising results. Fractions support standard arithmetic operators. Fractions don't support operations like square root that can result in irrational values. However, you can pass them to the floor and ceiling functions of the math module in the next module of core python numeric types, dates and times. Will it get complex pythons numeric type for representing numbers with imaginary components. Thanks for watching and we'll see you in the next module

Complex Numbers

Construction

[Autogenerated] many areas of science and engineering rely heavily on the notion of numbers with imaginary components, and Python provides built in support for this kind of mathematics. In this module of core python numeric types, dates and times, we'll look at pythons built in type for working with numbers with imaginary components complex. We'll see how to construct complexes from objects of other types. Well, look at accessing the components of complex numbers. We'll explore the sea math module for performing standard math operations with complex is, and we'll look at an example of how to use complex in a realistic application, Python supports one more numeric type that for complex numbers, this course isn't the place to explain complex numbers in depth. If you need to use complex numbers, you probably already have a good understanding of them so well quickly cover the syntactic specifics. For python. Complex numbers are built into the python language and don't need to be imported from a module. Each complex number has a real part and an imaginary part, and Python provides a special, literal syntax to produce the imaginary part by placing A J suffix onto a number where J represents the imaginary square root of minus one. Here we specified the number, which is twice the square root of minus one, depending on which background you have. You may have been expecting an eye here rather than a J. Python uses the convention adopted by the electrical engineering community for denoting complex numbers, where complex numbers have important uses as well See shortly rather than the convention used by the mathematics community, an imaginary number can be combined with the regular float, representing a real number using the regular arithmetic. Operators notice that this operation results in a complex number with non zero real and imaginary parts. So python displays both components of the number in parenthesis to indicate that this is a single object. Such values have a type of complex. The complex constructor can also be used to produce complex number objects. It could be passed one or two numeric values representing the rial and optional imaginary components of the number. Or it could be passed a single string argument containing a string delimited by optional parentheses, but which must not contain white space. Notice that the complex constructor will accept any numeric type so it could be used for conversion from other numeric types. This is much the same as the end. In float constructors, however, the real and imaginary components of complex are represented internally as floats with all the same advantages and limitations to extract the real and imaginary components as floats used the rial and I mag attributes. Complex numbers also support a method to produce the complex conjugate.

Operations

[Autogenerated] the functions of the math module cannot be used with complex numbers. So a module see math is provided containing versions of the functions, which both accept and return complex numbers. For example, the regular math on sq artie function cannot be used to compute the square roots of negative numbers. The same operation works fine, however, with C math dot sq Artie returning an imaginary result. In addition to complex equivalents of all the standard math functions, see math contains functions for converting between the standard Cartesian form and pull their coordinates to obtain the phase of a complex number. Also known in mathematical circles as its argument use. See math dot phase. But to get its module, ISS or magnitude, use the built in ABS function. We'll return to the ABS function shortly. In another context, thes two values could be returned as a to prepare using the C math dot polar function, and this, of course, can be used in conjunction with to pull unpacking. The operation can be reversed using the C math dot wrecked function. Note, though, that repeated conversions may be subject to floating point rounding error. As we've experienced here,

A Practical Example

[Autogenerated] to keep this firmly grounded in reality. Here's an example of the practical application of complex numbers to electrical engineering. Specifically will analyze the phase relationship of voltage and current in a sea circuits. First, we create three functions to create the complex values for the impedance of inductive, capacitive and resistive electrical components. The impedance of a circuit is the sum of the quantities for each component. We can now model a simple series circuit within induct ER of 10 homes. Reactant ce, a resistor of two gnomes, resistance and a capacitor with five homes. Reactant ce we now use See math dot phase to extract the phase angle from the previous result and convert this from radiance two degrees using a handy function in the math module. This means that the voltage cycle legs the current cycle by a little over 26 degrees in this circuit. Let's review what we've covered in this module. Pythons complex numeric type models numbers with imaginary components. You can construct complex from other numeric types or from strings complex stores. Its component values as floats. Functions from the standard library math module do not work on complex values, but the sea math module provides versions of the math functions for complex in the next module of core python Doom, Eric Types, dates and times we'll look at a few of the built in functions that Python provides for working with numeric types. Thanks for watching, and we'll see you in the next module.

Built-in Functions Relating to Numbers

Built-in Functions for Numbers

[Autogenerated] along with the methods to find on the numeric types themselves and the functions to find in standard library modules, Python provides a number of built in functions for working with numeric types. In this module of core python America type states and times we look at built in functions for rounding numbers and calculating absolute values well. Briefly review literal forms for creating integers using different bases, and we'll see how to create string versions of integral values in different bases. As we've seen, Python includes a large number of built in functions, and we'd like you to have seen them all, excluding a few we think you should avoid by the end of the core python Siris. Several of the built in functions are operations on numeric types, so it's appropriate to cover them here. We already briefed the encounter the ABS function when looking at complex numbers where it returned to the magnitude of the number, which is always positive. When used with integers, floats, decimals or fractions, it simply returns the absolute value of the number, which is the non negative magnitude without regards to its sign in effect for all number types, including complex ABS returns the distance from zero another. Built in is the round function, which rounds to a given number of decimal digits. So, for example, we can round 0.281223 decimal digits and get 0.281 We'll recon ground 0.6 to 5 to one decimal digits and get to your 10.6 to avoid bias when they're two equally close alternatives rounding us towards even numbers. So round 1.5 rounds up and round 2.5 rounds down. As for ABS, the round function is implemented for INT, where it has no effect float, which we've already seen decimal and even for fraction round is not supported. However, for complex be aware that when used with Float, which uses a binary representation round which is fundamentally a decimal operation could give surprising results. For example, rounding 2.675 to 2 places should yield 2.68 since 2.675 is midway between 2.67 and 2.68 and the algorithm rounds towards the even digit. However, in practice we get an unexpectedly rounded down results as we've seen before. This is caused by the fact that are literal float represented in base 10 can't be exactly represented in base too. So what is getting grounded is the binary value, which is close to, but not quite the value we specified. If avoiding these quirks is important to your application, you know what to do. Use the decimal type. All this talk of number basis brings us to another set of built in functions based conversions.

Base Conversions

[Autogenerated] back in the corps Python getting started. Course we saw that python supports entered your literal zin base to or binary using the zero B prefix base eight or Octel using the zero prefix and based 16 or Hexi decimal using the zero X prefix. Using Lee been Oct and hex functions. We can convert in the other direction with each function returning a string containing a valid python expression. If you don't want the prefix, you can strip it off using string slicing. Thean constructor and conversion function also accepts an optional base argument. Here. We use it to parse a strain containing a Hexi decimal number without the prefix into a knitted your object. The valid values of the base argument are zero and then 2 to 36 inclusive for numbers and basis to to 36 as many digits as required from the sequence of 0 to 9, followed by a dizzy are used. All the letters, maybe in lower case or upper case when specifying binary octel or Hexi decimal strings. The standard python prefix may be included finally based zero tails pipin to interpret the string according to whatever the prefixes or if no prefixes present. Assume it is decimal. Note that base one or you Neri Systems of counting are not supported. Let's review what we've covered in this module. The built in ABS function calculates the magnitude of a number. The built in round function rounds to a given number of decimal digits. Round doesn't work for complex numbers and can get surprising results with float for the unwary. Python has integer literal forms for binary Octel and Hexi decimal representations. The been function produces a binary representation of an integer. Oct produces the octo representation. Evidente, Jer and Hex produces the hex, a decimal representation. Thean Constructor accepts an optional base argument. It specifies the base to use when interpreting the string argument, and it takes any value from 23 36 based can also be zero, in which case Python uses the strings prefix or decimal, in the next module of core python numeric types, dates and times. We'll look at pythons, sophisticated support for working with dates and times. Thanks for watching, and we'll see you in the next module

Dates and Times with the Datetime Module

The Datetime Module

[Autogenerated] while they may not come immediately to mind with thinking about numeric types, dates and times are important to miracle concepts, which Python provides excellent support in this module of core python numeric types, dates and times, we'll explore the standard library date time module. We'll see howto work with calendar dates using the date time dot date type. Well, look at how to work with times of day using daytime dot time, and we'll use daytime dot daytime toe work with times on a specific date. We'll use date time dot time Delta to represent differences between times and see how we can use it to do some arithmetic with dates and times will cover pythons support for working with time zones. The last important scaler types we consider in this course come from the date time module. The types in this module should be your first resort when you need to represent time related quantities, the types are date a Gregorian calendar date. Note that this type assumes a pro LEP tick Gregorian calendar that extends backwards for all eternity and into the infinite future. For historical dates, this must be used with some care. The last country to adopt the Gregorian calendar with Saudi Arabia in 2016. Time the time within an ideal day which ignores leap seconds, date time, a composite of date and time. Each of these two value types of the time component can be used in so called naive or aware modes. In naive mode, the value lacks time zone and daylight savings. Time information and their meaning with respect to other time values is purely by convention within a particular program. In other words, part of the meaning of the time is implicit. On the other hand, in Aware mode, these objects have knowledge of both time zone and daylight savings time, and so can be located with respect to other time objects. The abstract TZ info and concrete time zone classes are used for representing the time zone information required for aware time objects. Finally, we have Time Delta a duration expressing the difference between to date or daytime instances. As with the other number and scaler types, we've looked at all objects of these types are immutable. Once created, their values cannot be modified

Dates

[Autogenerated] let's start by importing the daytime module and representing some calendar dates. The year, month and day are specified in order of descending size of unit duration, although if you can't remember the order, you can always be more explicit with keyword arguments. Each value is an integer, and the month and day values are one based. So is in the example here. January is month one, and the sixth day of January is day six. Just like regular dates for convenience, the date class provides a number of named constructors or factory methods implemented as class methods. The first of these is today, which returns the current date. There's also a constructor which can create a date from a posits time stamp, which is the number of seconds since the first of January 1970. For example, the billions second fell on the ninth of September 2001. The third, named Constructor, is from Ordinary, which accept an interview number of days, starting with one on the first of January in year one. Assuming the

Gregorian calendar extends back that far, the year, month and day values could be extracted with the attributes of the same name. There are many useful instance methods. On date, we cover some of the more frequently used ones here to determine the weekday use either the weekday or ice a weekday methods. The former returns a zero based day number with the range of 0 to 6. Inclusive Where. Monday zero and Sunday at six. Theis, a weekday method uses a one based system where Monday is one Sunday at seven to return a string in ISO 86 So one format, by far the most sensible way to represent dates as text used the isil format method for more control over date formatting a strings. You can use thes stir F time method right as string format time, using a wide variety of placeholders as specified in the python documentation or the format method of the string type with a suitable for Matt Placeholder format string. Unfortunately, both of these techniques delegate to the underlying platform dependent libraries underpinning your python interpreter so the format strings could be fragile with respect to portable code. Furthermore, many platforms did not provide tools to modify the result in subtle ways, such as omitting the leading zero on month days, less than 10 on this computer, we can insert a hyphen to suppress leading zeros, but this is not portable, even between different versions of the same operating system. A better and altogether more Python IQ solution is to extract the date components individually and pick and choose between date specific form. Writing operators and date attribute access for each component, which is more powerful and portable. Finally, the limits of date instances could be determined with the men and Max class attributes and the interval between successive dates retreat from the resolution. Class attributes the result from which is, in terms of the time Delta type, which will return to shortly.

Times

[Autogenerated] the time class is used to represent the time within an unspecified day with optional time zone information. Each time value is specified in terms of four attributes for hours, minutes, seconds and microseconds. Each of these is optional, although of course the proceeding values must be provided if positional arguments are used. As is so often the case, keyword arguments can lend a great deal of clarity to the code. All values are zero based. Integers recall that for date they were one based, and the value we have just created represents the last representive all instant of any day. Curiously, there are no named constructors for time objects. The components of time could be retrieved through the expected attributes. As for dates and I. So 8601 string representation can be attained with E isso format method, with more sophisticated formatting available through the stir F time method and the regular straw dot format method. Although the same caveats about delegating to the underlying see library apply with the portability traps for the unwary, we prefer the more Python IQ version, the minimum and maximum times, and the resolution could be obtained using the same class attributes as for dates

Combined Dates and Times

[Autogenerated] you may have noticed that throughout this section we fully qualified the types in the daytime module with the module name, and the reason why will now become apparent. The composite type, which combines date and time into a single object, is also called a time with a lower case D. For this reason, you should avoid using from daytime import daytime because from that point on the date time name and now refers to the class rather than the enclosing module as such, trying to get hold of the time type, then result in retrieval of the time method of the daytime class tow. Avoid this nonsense. You could import the date time class and bind it to an alternative name. Or use a short module name by using import to date time as D T. We'll continue what we've been doing and fully qualify the name. As you might expect, the compound daytime constructor accepts year, month, day, hour, minute, second and microsecond values of which at least year, month and Dave must be supplied. The argument ranges are the same as for the separate date and time constructors. In addition, the daytime class sports a rich selection of named constructors implemented as class methods. The today and now methods are almost synonymous, although now maybe more precise on some systems. In addition, the now method allows specifications of a time zone, but we'll return to that topic later. Remember that these functions and all other constructors Racine so far returned the local time according to your machine, without any record of where that might be. You can get a standardised time using the UTC now function, which returns the current coordinated universal time UTC taking into account the time zone of your current locale. As with the date, class daytime supports the from orginal and from timestamp methods simplemente by a U T. C from Time Stamp, which also returns a naive daytime object. If you wish to combine separate date and time objects into a single date time instance, you can use the combined classmethod, for example, to represent 8 15 this morning. You can do this. The final named constructor stir P time read as string parts time can be used to parse a date in string format according to a supplied format string, using the same syntax as used for rendering dates and times two strings in the other direction with stur f. Time to obtain separate date and time objects from a daytime object used the date and time methods. Beyond that, the date time type essentially supports the combination of the attributes and methods supported by date and time individually, such as the Day a tribute and I so format for ice 086 so one day times.

Durations

[Autogenerated] durations are modeled in python by the time Delta type, which is the difference between two dates or date times the time. Delta constructor is superficially similar to the constructor for other types, but it's, um, important differences. The constructor accepts a combination of days, seconds, microseconds, milliseconds, minutes, hours and weeks. Although position arguments could be used, we strongly urge you to use keyword arguments for the sake of anybody reading your code in the future, including yourself. The constructor normalizes and sums the arguments specifying one millisecond in 1000 microseconds. Results in a total of 2000 microseconds noticed that only three numbers are stored internally, which represent today's seconds and microseconds. No special string formatting operations are provided for time Delta's, although you can use thes straw function to get a friendly representation compared to the Ripper we've already seen. Timed out. The objects arise when performing arithmetic on daytime or date objects. For example, subtracting to date times yields a time delta or to find the date in three weeks time, we can add a time delta to a date time. Be aware that arithmetic on simple time objects is not supported

Timezones

[Autogenerated] So far, all of the time related objects we have created have been so called naive times, which represent times in local time. To create time zone aware objects, we must attach instances of a T Z info object to our time values, time zones and daylight saving time are very complex domain mired in international politics and which could change at any time. As such, the Python standard library does not include exhaustive time zone data. If you need up to date time zone data, you'll need to use the third party pie T Z or date you till modules that said Python three. Although not python, to contains rudimentary support for times own specifications. The T Z info abstraction, on which more complete time zone support could be added, is supported in both python, too, and python three. The T Z info classes abstract and so cannot be in stanciute it directly. Fortunately, Python three includes a simple time zone concrete class which can be used to represent time zones which are fixed offset from you TC, for example, here in Norway, we are currently in the Central European Time, or C e ___ time zone, which is UTC plus one. Let's construct a time zone object to represent this. I can now specify this TZ info instance. When constructing a time or daytime object, here's the departure time of my flight to London tomorrow. The Time Zone class has an attribute called UTC, which is an instance of Time zone configured with zero offset from UTC, useful for representing UTC times in the wintertime. London is on U T. C. So I'll specify my arrival in UTC. The flight duration is 9300 seconds more usable e formatted as two hours 35 minutes Form or complete time zone support, including correct handling of daylight savings time, which is not handled by the basic Time zone class. You'll need toe either subclass that easy info based class yourself for which instructions are provided in the python documentation or employee. One of the third party packages, such as pie TZ

Summary

[Autogenerated] Let's review what we've covered in this module. Pythons Date Time module provides support for working with dates and times the date time dot date type represents specific. A calendar dates. The date time dot Time type represents times of day without being associated with a specific day. The date time dot dates Time type represents a combination of both a specific day in a specific time. On that day, daytime dot time Delta represents durations or differences between two points in time time. Delta allows us to perform certain kinds of arithmetic with dates and times. The T Z Info class provides an abstract interface for working with time zone information. The time zone type is a simple implementation of the TZ info interface. But because global time zone details are subject to change at any moment, the Python standard library steers clear of providing exhaustive time zone support in the next module of core python numeric types, dates and times. We'll look at an extended example from the field of computational geometry that leverages what we've learned and illustrates why different kinds of never types could be critical in doing numerical computations. Thanks for watching, and we'll see you in the next module

Computational Geometry

Example Problem: Collinearity

[Autogenerated] we've covered our number of numerical types that Python provides. So now we'll look at an extended example that demonstrates the importance of knowing how to use these different types to solve real problems in numerical computation. In this module of core python numeric types, dates and times will introduce a problem from the domain of computational geometry will implement the computation using floats, and we'll see some unexpected and undesirable effects that arise from the nature of floating point arithmetic. Well, then, implement the computation using a different new miracle type that doesn't have the same shortcomings. This float, but which has some of its own rational numbers, have an interesting role to play in the field of computational geometry, a world where lines have zero thickness, circles are perfectly round, and points are dimension lis, creating robust geometric algorithms using finite precision number types such as pythons Float is fiendishly difficult because it's not possible to exactly represent numbers such as 1/3 which rather gets in the way of performing simple operations like dividing online into exactly three equal segments. As such, rational numbers model by Pythons fraction type can be useful for implementing robust geometric algorithms These algorithms are often deeply elegant and surprising because they must avoid any detour into the room of irrational numbers, which cannot be represented in finding precision. Which means that using seemingly innocuous operations like square root, for example, to determine the length of a lying using path, a gris is not permitted. One example algorithm, which benefits from rational numbers, is a simple Colin E already test. That is a test that determines whether three points lie on the same line. This could be further refined to consider whether a query point P is above exactly on or below the line. A robust technique for implementing Colin E already is to use a so called orientation test, which determines whether three points were arranged counterclockwise and a straight lines there neither counterclockwise nor clockwise war clockwise. You don't need to understand the mathematics of the orientation test to appreciate the point that we're about to demonstrate. Suffice to say that the orientation of 32 dimensional points could be concisely computed by computing the sign of the determinant of a three by three matrix containing the X and Y coordinates of the points and question where the determinant happens to be the signed area of the triangle formed by the three points. This function returns plus one if the Poly Line P Q. R executes a left turn and the Lupus counterclockwise or zero if the Poly Linus straight or negative one. If the Poly line executes a right turn and the Lupus clockwise, thes values can in turn be interpreted in terms of whether the query point P is above, on or below the line through Q and R. To cast this formula and python, we need a sine function and a means of computing the determinant. Both of these air straightforward, although perhaps not obvious, and give us the opportunity to learn some new python. First, the sine function. You may be surprised to learn, and you wouldn't be alone, that there's no built in or library function and python, which returns the sign of a number as negative 10 or plus one. As such, we need to roll around the simple. The solution is probably something like this. With this sign of five is one sign of negative. Five is negative one, and the sign of zero is zero. This works well enough, a more elegant solution would be to exploit an interesting behaviour of the bull type specifically how it behaves under subtraction. Let's do a few experiments. Intriguingly, subtraction of bull objects has an interview result. In fact, when used in arithmetic operations this way, true is equivalent to positive one, and false is equivalent to zero. We can use this behavior to implement a most elegant sine function.

Now we need to compute the determinant. In our case, this turns out to reduce down to simply this, So the definition of our orientation function using to pull coordinate Paris reach point becomes simply this. Notice the num type argument. We use this to convert the numeric type used in the P Q and R arguments into another type before calculating the determinant by default. Orientation will compute using floats, but numb type will allow us to use other types in the computation as well. Let's test this on some examples. First, we set up three points a B and C. Now we test the orientation of ABC. This represents a left turn, so the function returns positive one, On the other hand, the orientation of a CB is negative one. Let's introduce 1/4 point D, Which is Colin Ear with A and C. As expected, our orientation function returns zero for the group A. C. D.

Implementing with float and Fraction

[Autogenerated] so far, so good. Everything have done so far is using entered your numbers which have arbitrary precision and our function on Lee uses multiplication and subtraction with no division to result in float values, so all of that precision is preserved. But what happens if we use floating point values as our input data? Let's try some different values using floats. Here are three points which lie on a diagonal line. As we would expect, our orientation test determines that these points are co linear. Furthermore, moving the point e up a little by increasing its Y coordinate even by a tiny amount gives the answer we would expect. Now let's increase the y coordinate just a little more. In fact, with increase it by the smallest possible amount to the next Represents will floating point number. Wow, according to our orientation function the points E f and G R. Colin ear again, this cannot possibly be. In fact, we can go through the next 23 successive floating point values with our functions still reporting that the three points are cold linear until we get to this value, at which point things settle down and become well behaved again. What's happening here is that we've run into problems with the finite precision of python floats at points very close to the diagonal line and the mathematical assumptions we make in our formula about how numbers work. Breakdown. Due to rounding problems, we can write a simple program to take a slice through this space, printing the value of our orientation function for all represent a ble points on a vertical line, which extends just above and below the diagonal line, the variable p y es or P wise, the definition of which we've alighted. Here is a list of the 271 nearest represent a ble y coordinate values to 0.5. We haven't included the code to generate the success of float values because it's far from straightforward to do in python and somewhat besides the point, we've included this list for you in the course material. Then the program iterated over these P y values and performs the orientation test. Each time when we look at that output, we see an intricate pattern of results emerge, which isn't even symmetrical around the central 0.5 value. By this point, you should at least be wary of using floating point arithmetic for geometric computation, lest you think this can easily be solved by introducing a tolerance value or some other clunky solution will save you the bother by pointing out that doing so merely moves thes fringing effects to the edge of the tolerance zone. What to do? Fortunately, as we've alluded to at the beginning of this tale, Python gives us a solution in the form of the rational numbers implemented as the fraction type to use. The fraction type instead of float will use the num type argument of orientation. First, we need to import the fraction class from the fractions module. Then we'll rerun our little program this time passing fraction as the fourth argument orientation there by using fraction in the computation. Here's the output we get this time using fractions internally, our orientation function gets the full benefit of exact arithmetic with effectively infinite precision and consequently produces an exact result, with only one position of P being reported as Colin Ear. With Q and R

Visualizing the Results

[Autogenerated] going further, we can map out the behavior of our orientation functions by hooking up our program to the BMP image File Writer recreated in module 11 of our core python getting started course by using our sequence of consecutive floats centered around the 0.5 value to generate two dimensional coordinates in a tiny square region straddling or diagonal line and then evaluating orientation functions. At each point, we can produce a view of how our diagonal line is represented. Using different number types, the code changes straightforward. First, we import our BNP module we created in core python getting started. We've included a copy in the example code for this course, too. Then we replace the code, which operates through our line transect with code to do something similar in two dimensions. Using NECID list comprehension to produce nested lists representing pixel data, we use a dictionary of three entries called color to met from negative 10 and positive one orientations to black, middle, gray and white, respectively. The inner comprehension produces the pixels within each row, and the outer comprehension assembles the rose into the image. We reverse the order of the rose using a call to reversed to get our corded axes to correspond with the image format conventions. The result of this call is that Pixels is now a list of lists of integers where each integer is a value of 01 27 or 2 55 depending on whether the pixel is below, on or above the line. These in turn, will correspond to the shades of black, gray and white in the image. Finally, the pixel data structure is written out as a BMP filed with a call to B m p dot right gray scale. This produces an image that we would intuitively expect. The diagonal is gray representing Colin ear points, while the rest of the images black or white, indicating non Colin E. Already. If we instead use float in the orientation calculation, we get a map of above and below the diagonal for float computations. While graphically interesting, this image shows us that using floats for the orientation calculation produces graze in many places off the diagonal. The lesson is that this geometric calculation benefits greatly from using the correct numeric type. Let's review what we've covered in this module. Computational geometry is a fascinating field in one, which sometimes requires a non intuitive approach to get correct results. Using the float type naively can lead to results, which are misleading or simply incorrect. The fraction type can sometimes help you avoid the shortcomings afloat so often with a substantial increase in run time well done on completing core python numeric types, dates and times. Numerical computation is central to many programs, and understanding the trade offs of the different number types in Python can make the difference between success and failure in such programs. This course has given you a solid overview of the numeric types provided by the Corps Python language as well as the standard library. We looked at the different characteristics of these number types, tried to give some insight into how and when to use them, and work to do a concrete example to see how their differences could be important in practice. Look out for other core python courses here on plural site, which build on the knowledge you've gained here and which explained the many other tools and abstractions provided by python for building powerful programs. Remember to check out our Python Craftsman Book series, which covers these topics in written form. Specifically, you'll find these topics covered in the pipe and journeymen, the second book of the trilogy, We'll be back with more content for the ever growing Python language in Library. Please remember, though, that the most important characteristic a python is that above all else, it's great fun to write Python software Happy programming.

Course authors

Austin Bingham

Robert Smallshire

Course info

Level	Intermediate

Rating	★★★★★
My rating	★★★★★
Duration	0h 55m
Released	10 Mar 2020

Share course