```python
# This is finutil.py.
#
# Original version May 7, 2017
# This version 2.9, June 6, 2020
# Maintained by Professor Evans
#
# Copyright 2020 Gary R. Evans
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
# DESCRIPTION
#
# These are the utilities that are used throughout the various financial models
# for such things as calculating options prices and volatilities. These are
# designed to be used with Version 3.7 or higher of Python.
#
# THE SECTION FOR TIME AND CALENDAR COUNTS was moved from here to timeutil.py on
# August 14, 2019. Methods moved include daysto, iso_saysto, iso_daysto_days and
# monthname. Not all programs have been tested for the transfer.
#
import math
import datetime
import scipy.integrate
import numpy as np
#
# Debugger to make sure we are using the right version of finutil
#
def which_finutil():
    return('Version 2.9 of finutil.py in PyFi.')
#
# Method make_bitseq is used to convert any string into sequences of 8 bits.
# NOTE: This was put at the top of the utility as an example of the newer
# formatting for methods.
#
def make_bitseq(s: str) -> str:
    return " ".join(f"{ord(i):08b}" for i in s)
#
# bsm_idv_call is a modification of the call_idv_bsdc model. This is the B-S-M IDV
# model for calls only, using divide and conquer iteration for conversion. This model
# requires stock price, strike price, days (use timeutil.iso_daysto_days if you have
# a calendar date), call price (use peg within this utility if you have bid/ask), and
# interest rate (0 is allowed). This passes out a list of the delta, probability ITM,
# and implied volatility. This is called often from the spread models.
# Added May 2020.
#
def bsm_idv_call(stock: float, strike: float, call_price: float, days: int,
    rate: float) -> list:
    target = call_price
    precision = float(1e-3)
```

```
    low = 0.0
    high = 1.0
    cipd = float((high+low)/2)    # cipd will be the IDV
    temp_cp_tu = copo_pitm(stock,strike,cipd,days,rate)  # passes out a tuple
    tempcp = temp_cp_tu[0]
    while tempcp<=(target-precision) or tempcp>=(target+precision):
        if tempcp >= (target+precision):
            high = cipd
        else:
            low = cipd
        cipd = float((high+low)/2)
        temp_cp_tu = copo_pitm(stock,strike,cipd,days,rate)
        tempcp = temp_cp_tu[0]
    delta = temp_cp_tu[1]
    prob_itm = temp_cp_tu[3]
    idv = cipd
    return [delta,prob_itm,idv]
#
# bsm_idv_put is a modification of the put_idv_bsdc model. This is the B-S-M IDV
# model for puts only, using divide and conquer iteration for conversion. This model
# requires stock price, strike price, days (use timeutil.iso_daysto_days if you have
# a calendar date), put price (use peg within this utility if you have bid/ask), and
# interest rate (0 is allowed). This passes out a tuple of the delta, probability ITM,
# and implied volatility. This is called often from the spread models.
# Added May 2020.
#
def bsm_idv_put(stock: float, strike: float, put_price: float, days: int,
    rate: float) -> list:
    target = put_price
    precision = float(1e-3)
    low = 0.0
    high = 1.0
    pipd = float((high+low)/2)    # pipd will be the IDV
    temp_pp_tu = popo_pitm(stock,strike,pipd,days,rate)  # passes out a tuple
    temppp = temp_pp_tu[0]
    while temppp<=(target-precision) or temppp>=(target+precision):
        if temppp >= (target+precision):
            high = pipd
        else:
            low = pipd
        pipd = float((high+low)/2)
        temp_pp_tu = popo_pitm(stock,strike,pipd,days,rate)
        temppp = temp_pp_tu[0]
    delta = temp_pp_tu[1]
    prob_itm = temp_pp_tu[3]
    idv = pipd
    return [delta,prob_itm,idv]
#
# csnd integrates a standard normal distribution up to some sigma.
#
def csnd(point: float) -> float:
    return (1.0 + math.erf(point/math.sqrt(2.0)))/2.0
#
# cnd integrates a Gaussian distribution up to some value.
#
def cnd(center: float, point: float, stdev: float) -> float:
    return (1.0 + math.erf((point - center)/(stdev*math.sqrt(2.0))))/2.0
#
# COPO: Calculating the BSM CALL option price, traditional model. This requires the
```

```
# user to provide stock price, strike price, daily volatility, risk-free interest
# rate and days to expiry. (To calculate days use method daysto below).
# This returns the call price, the delta, and duration volatility as a tuple
# array. See popo below for puts. NOTE: Since cum2 is itm_prob, expand the tuple
# to pass out itm_prob at some point for this and popo.
#
def copo(stock: float, strike: float, dayvol: float, days: int, rfir: float) -> list:
    d1 = math.log(stock/strike)+((rfir/365)+(dayvol**2)/2)*days
    durvol = dayvol*math.sqrt(days)
    delta = csnd(d1/durvol)
    cumd2 = csnd((d1/durvol) - durvol)
    discount = math.exp(-rfir*days/365)
    callpr = (stock*delta)-(strike*discount*cumd2)
    return [callpr,delta,durvol]
#
# COPO_pitm: Calculating exactly the same as copo, but also passing out one more
# variable in the tuple, probability of being in the money (at expiry). COPO
# above was not changed because too many programs use COPO and this may have broken
# them. Will merge them at a later time.
#
def copo_pitm(stock: float, strike: float, dayvol: float, days: int, rfir: float) -> list:
    d1 = math.log(stock/strike)+((rfir/365)+(dayvol**2)/2)*days
    durvol = dayvol*math.sqrt(days)
    delta = csnd(d1/durvol)
    pitm = csnd((d1/durvol) - durvol)
    discount = math.exp(-rfir*days/365)
    callpr = (stock*delta)-(strike*discount*pitm)
    return [callpr,delta,durvol,pitm]
#
# delta_call: the estimator for the classical call delta.
# This version allows for drift and uses the half-variance ITO
# adjustment. This takes the risk-free rate into account.
# See also delta_put. For documentation ...
#
def delta_call(stock_price: float, strike: float, sigma: float, alpha: float,
    rate: float, days: float) -> float:
    sto_pr_exp = stock_price*math.exp(alpha*days)
    dur_vol = sigma*math.sqrt(days)
    log_spread = math.log(strike/sto_pr_exp)
    norm_ls = log_spread/dur_vol
    norm_ls_adj = norm_ls - (dur_vol/2) - dcount_rfr_opm(rate,sigma,days) # Ito ajustment
    return 1 - csnd(norm_ls_adj)
#
# delta_put: the estimator for the classical put delta.
# This version allows for drift and uses the half-variance ITO
# adjustment. This takes the risk-free rate into account.
# See also delta_call. For documentation ...
#
def delta_put(stock_price: float, strike: float, sigma: float, alpha: float,
    rate: float, days: float) -> float:
    sto_pr_exp = stock_price*math.exp(alpha*days)
    dur_vol = sigma*math.sqrt(days)
    log_spread = math.log(strike/sto_pr_exp)
    norm_ls = log_spread/dur_vol
    norm_ls_adj = norm_ls - (dur_vol/2) - dcount_rfr_opm(rate,sigma,days) # Ito ajustment
    return csnd(norm_ls_adj)
#
# An elementary function for calculating the stock price adjusted for drift.
# Time can be an integer or a float.
```

```
#
def drift(alpha: float, time) -> float:
    return 1.0*math.exp(alpha*time)
#
# An elementary multiplier function for converting daily volatility to duration
# volatility. [Note: it was silly to do it this way rather than the way shown with
# dur_vol below, but the durvol method is retained because it is likely used in
# mutiple older programs].
#
def durvol(time) ->float:
    return 1.0*math.sqrt(time)
#
# The primary method for converting daily volatility to duration volatility. This
# function below requires the sigma argument, unlike the durvol function above.
#
def dur_vol(daily_vol: float, time) ->float:
    return daily_vol*math.sqrt(time)
#
# dcount is a time-discount function to discount the value of a future payment (like an
# option) discounted at the risk-free interest rate. The variable riskfreerate
# is annual and time is in days.
#
def dcount(riskfreerate: float, time) -> float:
    return 1.0*math.exp(-1.0*(riskfreerate/365)*time)

# dcount_rfr_opm is a component of options pricing models and calculators for delta
# and prob ITM. It is usually called as a component of the delta (d1) or prob_itm (d2)
# formula in options pricing models. It is equal to rfr/365 times days divided by
# duration volatility [sqrt(t)/t] reduced to [sqrt(t)].
#
def dcount_rfr_opm(riskfreerate: float, sigma, time) -> float:
    return ((riskfreerate/365)*math.sqrt(time))/sigma
#
# itm_call: the estimator for the probability that the call will be ITM at expiry.
# This version allows for drift and uses the half-variance ITO
# adjustment. See also itm_put. For documentation see Jupiter call_ITM_prob_vX.
#
def itm_call(stock_price: float, strike: float, sigma: float, alpha: float,
    days: float) -> float:
    sto_pr_exp = stock_price*math.exp(alpha*days)
    dur_vol = sigma*math.sqrt(days)
    log_spread = math.log(strike/sto_pr_exp)
    norm_ls = log_spread/dur_vol
    norm_ls_adj = norm_ls + (dur_vol/2)    # This is the Ito ajustment
    return 1 - csnd(norm_ls_adj)
#
# itm_call_rfr: the estimator for the probability that the call will be ITM at expiry.
# This version allows for drift and allows for a risk-free rate. itm_call was left
# unaltered because it is used on a lot of legacy programs single-day trade programs.
# This is for longer-duration trades. Only line 5 is changed.
#
def itm_call_rfr(stock_price: float, strike: float, sigma: float, alpha: float,
    rfrate: float, days: float) -> float:
    sto_pr_exp = stock_price*math.exp(alpha*days)
    dur_vol = sigma*math.sqrt(days)
    log_spread = math.log(strike/sto_pr_exp)
    norm_ls = log_spread/dur_vol
    norm_ls_adj = norm_ls + (dur_vol/2) - dcount_rfr_opm(rfrate,sigma,days) # Ito ajustment
    return 1 - csnd(norm_ls_adj)
```

```
#
# itm_option: itm_call and itm_put rolled into one - indentical except the user
# sets true of false to the question of whether this is a call.
# This providesthe estimator for the probability that the option will be ITM at expiry.
# This version allows for drift and uses the half-variance ITO
# adjustment. For documentation see Jupiter call_ITM_prob_vX.
#
def itm_option(stock_price: float, strike: float, sigma: float, alpha: float,
    days: float, call: bool) -> float:
    sto_pr_exp = stock_price*math.exp(alpha*days)
    dur_vol = sigma*math.sqrt(days)
    log_spread = math.log(strike/sto_pr_exp)
    norm_ls = log_spread/dur_vol
    norm_ls_adj = norm_ls + (dur_vol/2)    # This is the Ito ajustment
    if call:
        prob = 1 - csnd(norm_ls_adj)
    else:
        prob = csnd(norm_ls_adj)
    return prob
#
# itm_option_rfr: itm_call_rfr and itm_put_rfr rolled into one - indentical except
# the user sets true of false to the question of whether this is a call.
# This providesthe estimator for the probability that the option will be ITM at expiry.
# This version allows for drift, takes an interest rate (which is how it differs from
# itm_option) and uses the half-variance ITO adjustment. For documentation see Jupiter
# call_ITM_prob_vX.
#
def itm_option_rfr(stock_price: float, strike: float, sigma: float, alpha: float,
    rfrate: float, days: float, call: bool) -> float:
    sto_pr_exp = stock_price*math.exp(alpha*days)
    dur_vol = sigma*math.sqrt(days)
    log_spread = math.log(strike/sto_pr_exp)
    norm_ls = log_spread/dur_vol
    norm_ls_adj = norm_ls + (dur_vol/2) - dcount_rfr_opm(rfrate,sigma,days) # Ito ajustment
    if call:
        prob = 1 - csnd(norm_ls_adj)
    else:
        prob = csnd(norm_ls_adj)
    return prob
#
# itm_put: the estimator for the probability that the call will be ITM at expiry.
# This version allows for drift and uses the half-variance ITO
# adjustment. For documentation see Jupiter call_ITM_prob_vX.
#
def itm_put(stock_price: float, strike: float, sigma: float, alpha: float,
    days: float) -> float:
    sto_pr_exp = stock_price*math.exp(alpha*days)
    dur_vol = sigma*math.sqrt(days)
    log_spread = math.log(strike/sto_pr_exp)
    norm_ls = log_spread/dur_vol
    norm_ls_adj = norm_ls + (dur_vol/2)    # This is the Ito ajustment
    return csnd(norm_ls_adj)
#
# itm_put_rfr: the estimator for the probability that the put will be ITM at expiry.
# This version allows for drift and allows for a risk-free rate. itm_put was left
# unaltered because it is used on a lot of legacy programs single-day trade programs.
# This is for longer-duration trades. Only line 5 is changed.
#
def itm_put_rfr(stock_price: float, strike: float, sigma: float, alpha: float,
```

```
    rfrate: float, days: float) -> float:
    sto_pr_exp = stock_price*math.exp(alpha*days)
    dur_vol = sigma*math.sqrt(days)
    log_spread = math.log(strike/sto_pr_exp)
    norm_ls = log_spread/dur_vol
    norm_ls_adj = norm_ls + (dur_vol/2) - dcount_rfr_opm(rfrate,sigma,days) # Ito ajustment
    return csnd(norm_ls_adj)
#
# ito_half-var is the adjustment that must be made to drift in the Geometric
# Brownian Motion Model, such as calculating the probability of being ITM
# for an option. Similar to lnmeanshift.
#
def ito_half_var(sigma: float, days: float) -> float:
    return (sigma*sigma*days/2)


#
# def_itm_prob_call was removed and replaced with delta_itm (for both calls and puts). This may
# break some programs!! January 9, 2020
#
# lnmeanshift is an elementary price expected-mean-value adjustment multiplier
# for log distributed prices for 1 day only. The mean of a log-distributed pdf is adjusted by
# minus one-half variance. NOTE: This is similar to half_var. Various models
# used both names and we didn't want to break anything.
#
def lnmeanshift(sigma: float) -> float:
    return 1.0*math.exp(-1.0*(sigma*sigma/2))
#
# norm_dist (normal distribution) accepts a single value for x, mu, and sigma
# and returns a scalar solution for the pdf (the MLE). This is ideal for use with a
# lambda function. See also stan_norm_dist below (for standard normal).
#
def norm_dist(x: float, mu: float, sigma: float) ->float:
    always = 1/math.sqrt(2*math.pi*(sigma**2))
    expo = -((x - mu)**2)/(2*(sigma**2))
    pdf = always*math.exp(expo)
    return pdf
#
# norm_dist_vec (normal distribution) is designed to accept x as a NUMPY
# ARRAY that has been established with a numpy command like np.linspace(-3,3,61),
# along with scalars for mu and sigma. This returns another numpy array of the
# pdf. Note the difference between this and norm_dist. See also stan_nd_vec
# for standard normal array. This can be integrated with a lambda function.
#
def norm_dist_vec(x: np.ndarray, mu: float, sigma: float) ->np.ndarray:
    always = 1/math.sqrt(2*math.pi*(sigma**2))
    length = x.size
    pdf_vec = np.zeros(length)
    pdf_vec = always*np.exp(-((x - mu)**2)/(2*(sigma**2)))
    return pdf_vec
#
# norm_dist_cdf integrates a normal vector using a lambda function and
# scipy's integration function (which can only take a function as an input).
# See the explanation for the stan_norm_cdf function (standard normal) below
# because this is easier to understand after seeing that. The output
# is usually thought of as a probability. Note the use of lambda and its use of
# z.
#
def norm_dist_cdf(mu: float, sigma: float, low_lim: float, point: float) ->tuple:
    nd_function = lambda z: norm_dist(z,mu,sigma)
```

```
    return scipy.integrate.quad(nd_function,low_lim,point)
#
# OTRANCHE is an option tranche value calculator function that assumes you
# have a stock price and strike price, adjusted externally (for example, drift
# is adjusted with drift above). This will calculate the strike-price adjusted
# tranche from either -5 sigma to the strike or from the strike to +5 sigma.
# Sigma used here is duration sigma, adjusted outside using durvol above.
# Main program must set call to true if a call, false if a put.
#
# NOT DIRECTLY RETESTED AS OF JUNE 27, 2019 - HOWEVER, otranche is called
# extensively by oidv in many applications with no problems.
#
def otranche(stock,strike,dursigma,call):
    sspread = (math.log(strike/stock))/dursigma
    if call:
        binborder = np.linspace(sspread, 5.00, num=24, dtype=float)
    else:
        binborder = np.linspace(-5.0, sspread, num=24, dtype=float)
    size = len(binborder)
    binedgeprob = np.zeros(size)
#
    for i in range(0,size):
        binedgeprob[i] = csnd(binborder[i])
    size = size - 1
    binprob = np.zeros(size)
    binmidprice = np.zeros(size)
    binvalue = np.zeros(size)
#
    for i in range(0,size):
        binprob[i] = binedgeprob[i+1] - binedgeprob[i]
        binmidprice[i] = ((stock*math.exp(((binborder[i+1]+binborder[i])/2.0)
        *dursigma))*lnmeanshift(dursigma)) - strike
        binvalue[i] = binmidprice[i]*binprob[i]
#
    if call:
        optionprice = np.sum(binvalue[0:(i+1)])
    else:
        optionprice = (np.sum(binvalue[0:(i+1)]))*-1.0
#
    return optionprice
#
# FTRANCHE is a full tranche value calculator function that assumes you have a
# stock price and reference price, usually a strike price (and still called that
# here). This will calculate the tranche value from either -5 sigma to the
# reference price (left is true) or from the reference to +5 sigma (left
# is false). This is similar to otranche except that it does not subtract the
# strike price and therefore gives the full value of the tranche. It is
# designed to be used primarily by the Aruba Model to calculate the value of
# the remaining stock tranche if the covered call you wrote has been exercised.
#
# NOT RETESTED AS OF JUNE 27, 2019
#
def ftranche(stock,strike,sigma,left):
    sspread = (math.log(strike/stock))/sigma
    if left:
        binborder = np.linspace(-5.0, sspread, num=24, dtype=float)
    else:
        binborder = np.linspace(sspread, 5.00, num=24, dtype=float)
    size = len(binborder)
```

```
    binedgeprob = np.zeros(size)
#
    for i in range(0,size):
        binedgeprob[i] = csnd(binborder[i])
    size = size - 1
    # size += 1
    binprob = np.zeros(size)
    binmidprice = np.zeros(size)
    binvalue = np.zeros(size)
#
    for i in range(0,size):
        binprob[i] = binedgeprob[i+1] - binedgeprob[i]
        binmidprice[i] = ((stock*math.exp(((binborder[i+1]+binborder[i])/2.0)
        *sigma))*lnmeanshift(sigma))
        binvalue[i] = binmidprice[i]*binprob[i]
#
    trancheprice = np.sum(binvalue[0:(i+1)])
    return trancheprice
#
# OIDV calculates implied daily and duration volatility for a call or a put
# using divide and conquer (the default for most models). Also see oidvnm.
# This uses an iterative process that uses otranche (above) to calculate the
# sigma, here an implied sigma, from the existing option value (ovalue).
# The call variable is True for a call, False for a put. The convergence is
# within the while loop. This function returns a tuple of two values, daily
# IDV and duration IDV.
#
def oidv(stock: float, strike: float, ovalue: float, days, call: bool) ->list:
    precision = float(1e-4)
    low = 0.0
    high = 1.0
    daysigma = float((high+low)/2)
    dursigma = daysigma*durvol(days)
    tempop = otranche(stock,strike,dursigma,call)
    while tempop<=(ovalue-precision) or tempop>=(ovalue+precision):
        if tempop >= (ovalue+precision):
            high = daysigma
        else:
            low = daysigma
        daysigma = float((high+low)/2)
        dursigma = daysigma*durvol(days)
        tempop = otranche(stock,strike,dursigma,call)
    # End of Loop!
    return [daysigma,dursigma]
#
# oidvnm calculates implied daily and duration volatility for a call or a put
# using Newton's Method for convergence (default is divide and conquer).
# This uses an iterative process that uses otranche (above) to calculate the
# sigma, here an implied sigma, from the existing option value (ovalue).
# The call variable is True for a call, False for a put. The convergence is
# within the while loop. This function returns a tuple of two values, daily
# IDV and duration IDV.
#
# NO LONGER USING BECAUSE OF EXPLOSIONS ... USE DIVIDE AND CONQUER (oidv). This
# should still work, though, for most applications.
#
def oidvnm(stock,strike,ovalue,days,call):
    seedsigma = 1e-6
    durseed = seedsigma*durvol(days)
```

```
    # NOTE: daysigma is supposed to be set at a reasonable estimate of the
    # actual idv. The Newton method can explode if it is not (especially if you
    # enter option price, strike, and value data that are very unrealistic).
    # Consider setting daysigma at sqrt*time*ln(strike/stock). It was
    # originally set at 0.05
    daysigma = (math.log(strike/stock))*math.sqrt(days)
    dursigma = daysigma*durvol(days)
    cutoff = 1e-4
    tempop = float(0.00)
    #
    # The loop starts here. You start with a test sigma and converge to the
    # actual sigma. The convergence shown here (Newton's method) was designed
    # by Alec Griffith '17
    #
    while np.abs(tempop - ovalue) > cutoff:
        tempop = otranche(stock,strike,dursigma,call)
        price2 = otranche(stock,strike,dursigma+durseed,call)
        deriv = (price2-tempop)/seedsigma
        daysigma -= (tempop-ovalue)/deriv
        dursigma = daysigma*durvol(days)
    # End of Loop!
    return [daysigma,dursigma]
#
# POPO: Calculating the BSM PUT option price, traditional model. This requires the
# user to provide stock price, strike price, daily volatility, risk-free interest
# rate and days to expiry. (To calculate days use method daysto above).
# This returns the put price, the delta, and duration volatility as a tuple
# array. See copo above for calls.
#
def popo(stock: float, strike: float, dayvol: float, days: int, rfir: float) -> list:
    d1 = math.log(stock/strike)+((rfir/365)+(dayvol**2)/2)*days
    durvol = dayvol*math.sqrt(days)
    delta = csnd(-d1/durvol)
    cumd2 = csnd(-(d1/durvol - durvol))
    discount = math.exp(-rfir*days/365)
    putpr = -(stock*delta)+(strike*discount*cumd2)
    return [putpr,delta,durvol]
#
# POPO_pitm: Calculating exactly the same as popo, but also passing out one more
# variable in the tuple, probability of being in the money (at expiry). POPO
# above was not changed because too many programs use POPO and this may have broken
# them. Will merge them at a later time. (Also see COPO_pitm).
#
def popo_pitm(stock: float, strike: float, dayvol: float, days: int, rfir: float) -> list:
    d1 = math.log(stock/strike)+((rfir/365)+(dayvol**2)/2)*days
    durvol = dayvol*math.sqrt(days)
    delta = csnd(-d1/durvol)
    pitm = csnd(-(d1/durvol - durvol))
    discount = math.exp(-rfir*days/365)
    putpr = -(stock*delta)+(strike*discount*pitm)
    return [putpr,delta,durvol,pitm]
#
# PEG: Simply takes the bid and ask and spread, and calculates the peg price.
# Used in combination with programs that also use bsm_idv_X and similar, which
# don't calculate peg. The spread_perc is a value between 0 and 1 and is added
# to the bid floor. Default is 0.50. Many IB algos use 0.70 for buy-to-open,
# 0.30 for sell-to-close as default.
# This is the IEEE754 standard for rounding numbers to nearest even.
#
```

```
def peg(bid: float, ask: float, spread_perc: float) -> float:
    ba_spread = ask - bid
    spread = ba_spread*spread_perc
    single_price_peg = round(bid + spread,2)
    return single_price_peg
#
# st_norm_df maps the standard normal probability density function of a point
# (the maximum likelihood estimator). This is an older method used in older
# programs and has been replaced with stan_norm_dist below, which lends itself
# to more complicated applications like stan_nd_vec.
# The point will be a float between -4.0 and 4.0.
#
def snormdf(point: float) -> float:
    return (1/(math.sqrt(2*math.pi)))*math.pow(math.e,-0.5*point**2)
#
# stan_norm_dist (standard normal distribution) is designed to accept a single
# value for x and returns as scalar solution for the standard normal pdf. This
# gives the same solution as snormdf above (MLE).
# This is ideal for use with a lambda function.
#
def stan_norm_dist(x: float) ->float:
    always = 1/math.sqrt(2*math.pi)
    expo = -(0.5*x**2)
    pdf_sn = always*math.exp(expo)
    return pdf_sn
#
# stan_nd_vec (standard normal distribution) is designed to accept x as a NUMPY
# ARRAY that has been established with a numpy command like np.linspace(-3,3,61).
# Mu is zero and sigma is one. This returns another numpy array of the pdf.
# Note the difference between this and stan_norm_dist. Note that we must use
# np.exp (returns array) rather than math.exp (returns scalar).
#
def stan_nd_vec(x: np.ndarray) ->np.ndarray:
    always = 1/math.sqrt(2*math.pi)
    length = x.size
    pdf_sn_vec = np.zeros(length)
    pdf_sn_vec = always*np.exp(-(0.5*x**2))
    return pdf_sn_vec
#
# stan_norm_cdf integrates a standard normal vector using a lambda function and
# scipy's integration function (which can only take a function as an input).
# Because you can't integrate from minus infinity, use a low_limit of -3.5 or
# similar if you are integrating from minus infinity. You are integrating to
# the point, which will effectively be a value between -3.5 and 3.5. The output
# is usually thought of as a probability. Note the use of lambda and its use of
# z.
#
def stan_norm_cdf(low_lim: float, point: float) ->tuple:
    snd_function = lambda z: stan_norm_dist(z)
    return scipy.integrate.quad(snd_function,low_lim,point)
#
# TDECAY adds an extension to the otranche calculator (Taboga model)
# to calculate one-day time decay.
#
def tdecay(stock: float, strike: float, daysigma: float, oprice: float, days, call: bool) ->float:
    days -= 1.0
    dursigma = daysigma*durvol(days)
    oprice1d = otranche(stock,strike,dursigma,call)
    timedecay = oprice - oprice1d
```

```
    return timedecay
#
if __name__ == "__main__":
    # x = copo(101.24,105.0,0.0160,8,0.00)
    # y = copo_pitm(101.24,105.0,0.0160,8,0.020)
    # x = delta_call(101.24,105.0,0.0160,0.0,0.020,8)
    # y = delta_put(101.24,98.0,0.0160,0.00021,0.020,8)
    # x = bsm_idv_call(101.24,105.0,0.565,8,0.01)
    # x = bsm_idv_put(101.24,98.0,0.610,8,0.01)
    # y = iso_daysto_days(2019, 7, 19)
    # x = peg(2.40, 2.50, 0.5)
    # x = popo(101.24,95.0,0.01820,8,0.00)
    # x = popo_pitm(101.24,98.0,0.0160,8,0.020)
    # x = oidv(101.24,105.0,0.555,8,True) # Note: if oidv works, otranche works.
    # x = tdecay(101.24,105,0.016,0.555,8,True)
    # x = stan_norm_dist(1.0)     # If 1, should be 0.24, if 0, should be slightly less than 0.40
    # x = ito_half_var(0.30,1)
    # x = np.linspace(-4.0,0.0,31)
    # y = stan_nd_vec(x)
    # x = norm_dist(1.00,1.0,0.5)
    # y = norm_dist_vec(x,0.0,1.0)
    # y = stan_norm_cdf(-3.5,0.8)
    # y = norm_dist_cdf(2.0,1.0,-1.5,2.8)
    # x = dur_vol(0.20,9)
    # z = itm_call(101.24,105.0,0.0160,0.000,8)
    # z = itm_call_rfr(101.24,105.0,0.0160,0.00,0.020,8)
    # x = itm_put(100.0,105.0,0.025,0.000794,16)
    # z = itm_put_rfr(101.24,98.0,0.0160,0.00021,0.020,8)
    # x = itm_option(100.0,95.0,0.025,0.000794,16,False)
    # z = itm_option_rfr(101.24,98.0,0.0160,0.00021,0.020,8,False)
    # print (type(x))
    # print (" ",x)
    # print (type(y))
    # print (" ",y)
    # print (" ",z)
    print ("  Completed, no obvious bugs.")
    print (" ",which_finutil())
```